

Guia Workflow Frontend



Uma metodologia para quem
já manja de HTML, CSS e Javascript

Daniel Tapias Morales

Tabela de conteúdos

| | |
|--|-------|
| Início | 1.1 |
| Introdução | 1.2 |
| No começo era tudo TAG | 1.2.1 |
| HTML, CSS e Javascript eram suficientes | 1.2.2 |
| Antes do susto | 1.2.3 |
| Depois do susto | 1.2.4 |
| Para quem foi escrito | 1.2.5 |
| Nodejs, NPM e terminal | 1.3 |
| O que é o nodeJS? | 1.3.1 |
| E o NPM? | 1.3.2 |
| Uma palavra sobre o terminal. | 1.3.3 |
| Versionamento e GIT | 1.3.4 |
| Comandos básicos Unix para manipulação de arquivos | 1.3.5 |
| Configurando e iniciando o Git | 1.3.6 |
| Iniciando um repositório local | 1.3.7 |
| Versionando nosso primeiro arquivo | 1.3.8 |
| Avaliando o histórico | 1.3.9 |
| Fundamentos do Github | 1.4 |
| Turbinando o NodeJS | 1.5 |
| Pensando fora do Browser | 1.5.1 |
| Instalando packages local ou globalmente | 1.5.2 |
| Package.json | 1.5.3 |
| Dependências de produção vs. de desenvolvimento | 1.5.4 |
| Automatização de tarefas | 1.6 |
| Criando a estrutura de diretórios | 1.6.1 |
| Instalando o GruntJS | 1.6.2 |
| Registrando tarefas | 1.6.3 |
| Criando os arquivos estáticos | 1.6.4 |
| Executando Tarefas | 1.6.5 |
| Tarefas em lote | 1.6.6 |

| | |
|---|---------|
| Considerações finais | 1.6.7 |
| Pré-processadores de CSS | 1.7 |
| O que é preciso para começar? | 1.7.1 |
| Aplicando Sass em nosso projeto | 1.7.2 |
| Pondo a mão na massa | 1.7.3 |
| Variáveis | 1.7.3.1 |
| @import | 1.7.3.2 |
| Entendendo o arquivo .map | 1.7.3.3 |
| Referência ao seletor pai | 1.7.3.4 |
| Aninhamento de seletores | 1.7.3.5 |
| Juntando o aninhamento, variáveis e seletor pai | 1.7.3.6 |
| Funções relacionadas a cores | 1.7.3.7 |
| @mixin | 1.7.3.8 |
| Considerações finais | 1.7.4 |
| Até Breve | 1.8 |

workflow front end

Há poucos anos, bastava um programador front-end dominar o html, o css e javascript para ter uma boa colocação no mercado. Logicamente, outras disciplinas eram incluídas nesse pacote, como por exemplo: semântica do HTML, Javascript não bloqueantes, acessibilidade, performance entre outras. Mas tudo girava em torno das três tecnologias principais.

Mas esse cenário mudou. Somou-se às competências acima outras habilidades técnicas como versionamento de arquivos, automatização de tarefas, pré-processadores css, NodeJS e NPM, para citar alguns. Estes são conhecimentos obrigatórios se você quiser entrar/se manter no mercado.

Com essa premissa em mente, fica fácil perceber que em pouquíssimo tempo muitos novos conhecimentos são exigidos aos profissionais da área. Em contrapartida, encontrar material de qualidade e com uma linguagem simples para quem está começando não é uma tarefa muito fácil. Os tutoriais e artigos de agora esperam que você já tenha a base, ou seja, já tenha o NodeJS instalado e saiba mexer no terminal. Não há um "Hello worldflow" (piadinha ruim essa, ne!).

Se você já entrou num artigo da internet e parou de lê-lo no momento que encontrou um texto parecido com "Digite npm install qualquer-coisa" você sabe do que eu estou falando.

Este guia é para você que manja de HTML, CSS e Javascript, mas que precisa conhecer as novas tecnologias que envolvem o universo front-end.

Além disso, este guia ainda está em desenvolvimento. Avisarei pelo meu twitter [@tapmorales](#) quando novos tópicos estiverem disponíveis. Me segue lá se você quiser saber quando tem coisa nova por aqui.

Se quiser bater um papo comigo, dá uma olhada nos [links que disponibilizo na minha página](#) - é só uma página mesmo, literalmente :).

Tenho também um projeto pessoal que você pode conferir em [serfrontend.com](#).

Para ler on-line, [clique aqui](#).

Se encontrou algum erro, pode fazer um [pull request aqui](#). Agradeço muito!

Seja muito bem-vindo. Espero que curta!

Daniel Tapias Morales



Esta obra está licenciado com uma Licença [Creative Commons Atribuição 3.0 Brasil](#).

Introdução

Nesta introdução você conhecerá os motivos que me levaram a escrever esse guia. Saberá para quem ele foi escrito e o que você aprenderá até o final da leitura.

No começo era tudo TAG

Me lembro até hoje, o momento em que criei o meu primeiro arquivo HTML. Isso foi lá pelo final dos anos 90, bem no meio da bolha. Eu abri um simples arquivo no notepad, escrevi 'olá mundo' envolto por uma simples tag `h1` e salvei o arquivo na área de trabalho com uma nova extensão, a tal da .html. E quando eu abri o arquivo, com um duplo clique, magicamente ele não abriu mais no bloco de notas, mas sim num programa muito melhor, o *Netscape navigator*.

Algum tempo depois apareceu uma tecnologia revolucionária para formatação de páginas, chamada CSS. Adeus ``. Apreendi várias novas propriedades de CSS que me ajudavam a colocar características visuais em meus sites montados com tabelas. Uma beleza!

Também aprendi Javascript, para que eu pudesse mostrar a data e a hora, e oferecer um pop-up de boas-vindas a todos os meus visitantes. Aahh! que saudade do `window.open` !

HTML, CSS e Javascript eram suficientes

Por muito tempo isso foi tudo o que era preciso saber para trabalhar como webdesigner. Com o tempo, fui me aprofundando nessas tecnologias e adquiri experiência construindo coisas mais interessantes do que simplesmente mostrar a data e um menu pop-up (Eu estou falando de janelas modais, carrosseis, validação de formulário e menus drop-downs. Talvez no futuro alguém fale desses caras com certa nostalgia. Assim como eu fiz quando lembrei do `window.open`).

A verdade é que até o final de 2013, trabalhando numa grande agência de publicidade, onde os projetos têm prazos curtos e ficam pouco tempo no ar, essas tecnologias eram tudo o que eu precisava. Lá, as vezes eu desenvolvia um projeto em uma semana para uma campanha que iria ficar on-line por 30 dias ou menos.

Nessa época, já havia surgido muita coisa na área de front-end: `node.js`, automatização de tarefas, pré-processadores CSS, etc. Mas como não via chances de usar essas coisas na agência, dei uma lida sobre esses assuntos, mas logo me esqueci de tudo pela falta de utilização prática.

Antes do susto

Mas algo muito positivo aconteceu na minha vida. Saí da agência onde os projetos tinham data de nascimento e de óbito definidas e fui trabalhar numa empresa onde eu tive a oportunidade de desenvolver produtos que teriam vida longa. Aplicativos baseados na web que seriam usados por muitas pessoas, e em diversos dispositivos. E agora? Até dá pra desenvolver aplicativos web usando javascript e jquery, (com um mínimo de padrão de projeto, por favor). Mas e a performance? Minificar e concatenar arquivos? Sprites de imagens? Eu até sabia tudo isso na teoria, mas na prática, nada. Eu precisava mudar isso. E rápido.

Então fui procurar material na internet. Procurei sobre automatização de tarefa e fui levado ao site do [grunt.js](https://gruntjs.com/). O link *"Getting Started"* é onde devemos clicar para aprender algo, não é mesmo? A primeira linha, do primeiro parágrafo, na primeira página que abro, diz o seguinte:

"Grunt and Grunt plugins are installed and managed via npm, the Node.js package manager."

Mas whata? npm? node.js? Ok. Tinha que correr atrás do prejuízo. Achei que seria mais fácil ver como o GruntJs funciona na prática. Googlei novamente e abri um site famoso em português que dizia:

...para instalar, digite:

```
sudo npm install -g grunt-cli
```

Digitar? Onde? Como assim?

Com um pouco de receio, entrei no site nodejs.org para saber mais sobre essa tecnologia. Confesso que o texto a seguir não me esclareceu muito:

"Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world."

Foi a partir desse ponto que eu fechei todas as abas do navegador. Abri meu Sublime e continuei com o velho e bom javascript. Do jeito que sempre foi. Ah, javascript, seu lindo!

Depois do susto

Mas é claro que se eu me propus a escrever todo esse texto que se segue, é porque eu superei esse trauma inicial e, sem a pressão do dia-a-dia, fui estudando aos poucos, à noite e aos finais de semana. Sem pressão, sem prazo.

Fui descobrindo aos poucos sobre como essas novas tecnologias funcionavam e onde me seriam úteis.

Ainda tenho muito o que aprender. Comecei os meus primeiros passos agora (no início de 2015). Mas creio que, mesmo com pouco tempo de experiência trabalhando com esse workflow front-end, já tenho uma base sólida e consiga passar esse conhecimento adiante.

Para quem foi escrito

Com este material pretendo ajudar a galera que está começando agora e que já conhece e consegue se virar com Javascript, HTML e CSS. Se você se identificar com alguns dos aspectos abaixo, esse material é pra você:

- Manjo de javascript mas gostaria de melhorar meu workflow front-end para desenvolvimento de sites ou aplicativos.
- Quando abro um tutorial na internet que começa com uma frase parecida com "Para começar, digite `sudo npm install ...`", ou - pior ainda - "Basta você digitar `npm install ...`", fecho-o imediatamente antes que me contamine.
- Sei da importância da performance de um site ou aplicativo, mas continuo entregando os mesmos javascripts de desenvolvimento em produção.
- Versionamento? Claro! Sempre copio e colo a pasta de projetos colocando a data no nome da pasta. Além disso, nunca apago um trecho de código javascript. Sempre comento código velho que não está sendo mais necessário. Vai que eu precise dele no futuro. Nunca se sabe!
- Eu ainda uso o bom e velho FTP para subir meus arquivos no servidor. Nunca falha.

Se você se identificou com alguns pontos descritos acima, espero que este material seja útil para você.

Se resolver continuar a leitura, você:

- Irá perder o medo do terminal.
- Irá conhecer um dos pré-processadores mais famosinhos do momento: o Sass.
- Concatenará e minificará seus javascripts e css. Acredite: é moleza e vale a pena.
- Nunca mais irá comentar códigos velhos com medo de precisar deles no futuro.
- Nunca mais irá colocar seus arquivos de desenvolvimento em produção.
- Não se assustará se abrir um tutorial na net que começa pedindo para você digitar algo começando com `npm install`. Você saberá o que isso significa e onde deverá digitar esse código maluco.

Ou seja, No final deste livro (se é que posso chamá-lo assim), você terá uma base sólida para iniciar o desenvolvimento de aplicativos para a web. Hoje em dia, saber HTML, CSS e Javascript não é suficiente. Você precisa conhecer versionamento (Git e GitHub). Você precisa instalar uma ferramenta de build (Grunt ou Gulp). Você precisa saber manipular um

gerenciador de dependências (NPM ou NPM + Bower), e se souber também um pouco sobre pre-processadores CSS, terá uma maior produtividade e acrescentará no teu currículo.

O que você não verá aqui

- Detalhes sobre javascript, jquery, css e html. Espero que você já possua esses pré-requisitos.
- Não vou desenvolver nenhum sistema do zero, e você não será um 'ninja' em Node.js. Meu intuito é fazer com que você saiba o básico, que eu adoraria que alguém tivesse me contado quando comecei a desenvolver aplicativos para a web.
- Não vou explorar nenhum framework em particular. Pelo menos não por enquanto.
- Mesmo sobre os assuntos que me proponho a explicar, por vezes não serei 100% fidedigno. Me proponho a explicar de um jeito fácil de entender. Provavelmente, se alguém tivesse feito isso comigo quando eu estava começando a entrar nesse terreno, algumas pedras sairiam do caminho antes de eu tropeçar. Pense assim: Se você tivesse que explicar javascript para alguém que está começando, provavelmente você não se preocuparia em explicar sobre escopo de variáveis. Estou quase certo que as suas variáveis estariam todas no escopo global. E sabemos que isso não é uma boa prática. Contudo, se entrar muito nesse nível de detalhe talvez você assuste o iniciante e o faça desistir antes mesmo de tentar. Nesse caso, não importa que o seu código se pareça com:

```
var texto = "Olá mundo";  
alert(texto);
```

Mesmo que isso não seja uma boa prática é fácil de entender. Lembre-se: os meus exemplos não têm a intensão de serem escritos como na vida real, contudo, espero te mostrar a base para que, depois, você possa avançar com os seus estudos.

Softwares utilizados:

Sistema Operacional:

Eu sei que há na comunidade um certo estrelismo por quem usa o mac ou linux. Não quero entrar nesse mérito. Sei também que há um certo preconceito com o windows. Também não quero discutir isso. Mas o fato é que eu, desde o começo da minha carreira, trabalhei com windows. Portanto, não vou escrever nada que eu não tenha proficiência. Tudo que você

verá aqui é voltado para o windows. Muitas coisas irão servir mesmo que você use o mac ou linux, mas não me comprometo com isso, uma vez que não tenho know-how nesses sistemas. Combinado?

Editor de código:

Há vários editores de código disponíveis no mercado e você provavelmente utiliza o Sublime Text com o package manager instalado e, no mínimo, o emmet. Isso não é primordial, mas sugiro que dê uma olhada no Sublime, se ainda não o conhece. Não faz diferença, nesse momento, se você usa o Notepad++, o Text Edit, o Brackets ou o Dreamweaver. Antes que chova pedra na minha cabeça, quero me explicar: já ví códigos muito mais semânticos e corretos escritos no Dreamweaver do que códigos escritos por pessoas que se achavam cools só porque usavam o Notepad ou Sublime. A ferramenta não é importante. O que importa é o desenvolvedor. É claro que com o tempo os devs com mais experiência tendem a abandonar o Dreamweaver e começar a utilizar um editor de código melhor. Mas isso não importa realmente. Se você usa o Dreamweaver, pense em ao menos experimentar algo diferente. E por falar nisso, eu comecei a escrever HTML usando o programa até então da Macromedia. Foi de lá que vieram os meus primeiros `<h1>` 's. E não há nada de errado nisso.

No próximo capítulo iremos entender de uma vez por todas o que é o nodejs, o NPM e o que é um programa do tipo terminal.

Nodejs, NPM e terminal

Nesta seção você saberá, sem muitas explicações técnicas demais, o que é esse tal de NodeJS. Saberá o que é um terminal e o que é o NPM. Se você nunca mexeu num terminal e isso te assusta um pouco, fique tranquilo. Ao final dessa leitura você não dominará em detalhes a linha de comando, mas conseguirá se virar bem com qualquer tutorial da internet que espera que você já saiba o que é um programa de linha de comando.

Boa leitura!

NodeJS, NPM e terminal

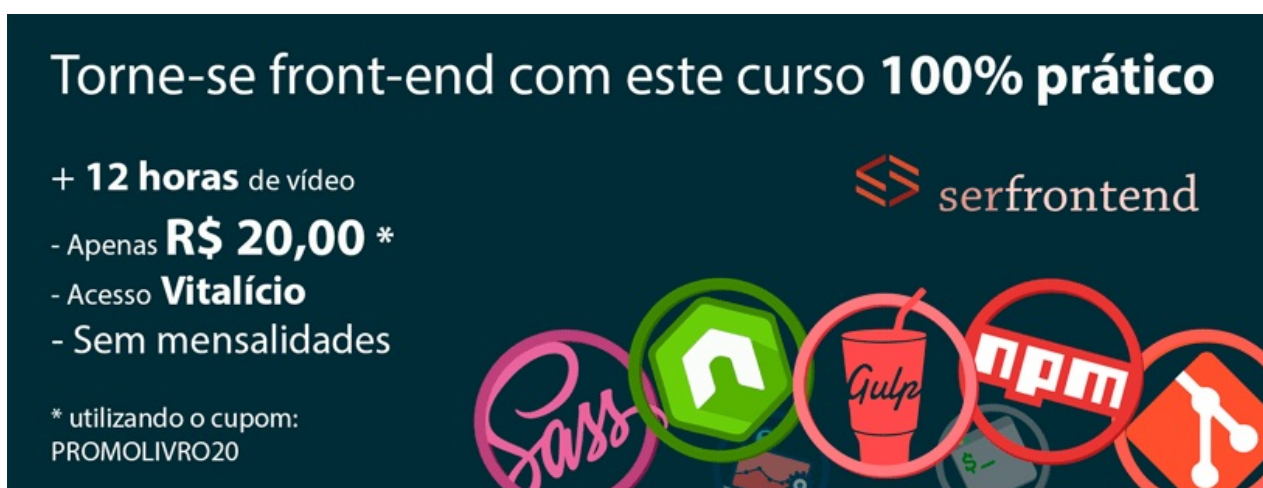
O que é o NodeJS?

É por causa do NodeJS que estamos aqui nesse momento. Foi a partir do NodeJS que o javascript, que antes era vista como uma linguagem inferior, ganhou notoriedade no mercado de desenvolvimento web. Depois do NodeJS, vieram o Express js (framework para criar servidores web com javascript), Grunt e Gulp (para automatização de tarefas) e até mesmo banco de dados NoSQL, como o mongoDB.

Esqueça as definições muito técnicas que você possa ter visto em sites que tratam do assunto. Elas estão corretas, mas são difíceis de entender. A partir de agora, pense no NodeJS como um software que você instala no seu sistema operacional, que permite você rodar códigos javascripts fora do Browser.

Quando percebi isso meu cérebro cresceu alguns milímetros. Vou repetir, dando ênfase no que considero mais importante:

O Node.js é um **software**, que você **instala no seu sistema operacional**, e que permite que você rode códigos **javascripts fora do Browser**.



Torne-se front-end com este curso **100% prático**

+ **12 horas** de vídeo

- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

serfrontend

Sass, Gulp, NPM, Git

Mas como isso foi possível? Simples, desenvolveram o NodeJS usando como base a tal da V8, que é a engine que interpreta o javascript no Google Chrome. Fantástico! Ou seja, com o NodeJS, você não depende mais do browser para ler e interpretar seus arquivos javascripts.

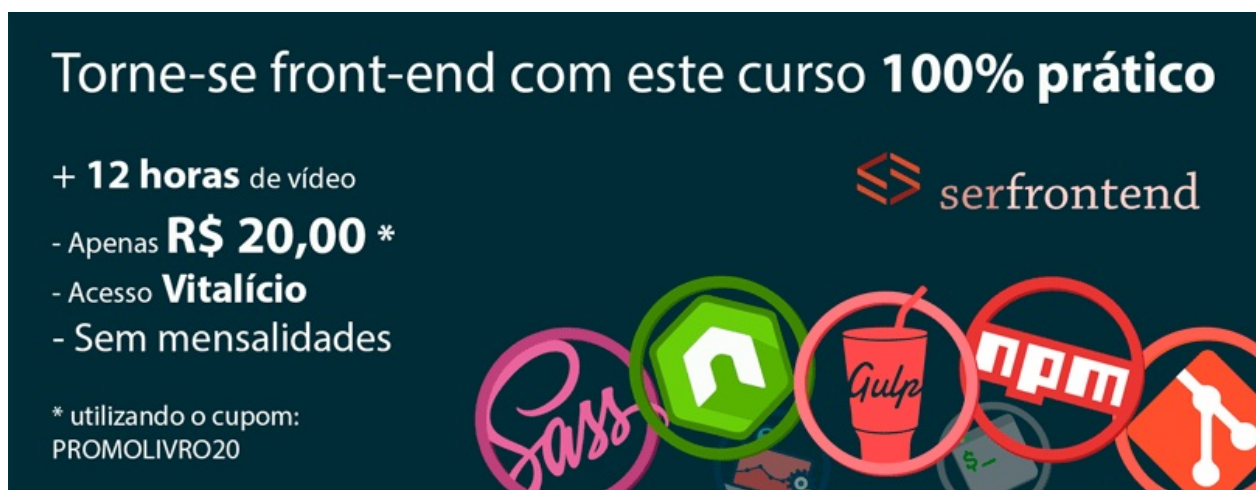
Mas e aquela definição do NodeJS retirada do site nodejs.org? Não se preocupe por enquanto. Lembre-se que não é necessário entender o escopo de variáveis quando estamos começando aprender javascript.

Sabendo disso, é hora de instalar o NodeJS. Não vou detalhar esse processo por ser bem simples. Basta fazer o download e dar um duplo clique no executável. Depois de alguns "Next" clique em "Finish". Depois de instalado, seu sistema poderá rodar arquivos javascripts fora do browser. Vai lá em <https://nodejs.org>. Eu te espero.

E o NPM?

O NPM é um gerenciador de pacotes do NodeJS. É através dele que instalamos e desinstalamos módulos (os pacotes, ou packages) que expandem as funcionalidades dessa ferramenta. Para facilitar o entendimento, vou utilizar uma analogia. Pense no node como sendo o seu Sublime Text. Se você já programa para a web, provavelmente já instalou algum plugin (como se fosse os módulos do node). Portanto, já deve ter instalado o Package manager do Sublime, certo? O Package manager é gerenciador de pacotes do Sublime, assim como o NPM é o gerenciador de pacotes do NodeJS.

Uma diferença é que através do NPM podemos instalar módulos (não vou mais falar que são plugins, ok?) tanto localmente (em cada projeto) quanto globalmente (no NodeJS instalado em seu sistema operacional).



Torne-se front-end com este curso **100% prático**

+ **12 horas** de vídeo

- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

serfrontend

Sass Home Gulp NPM

As versões mais recentes do NodeJS já vêm com uma versão do NPM, portanto, não há nada o que fazer, é só sair usando. O que significa que já pode instalar ou desinstalar módulos digitando poucos comandos no terminal.

Uma palavra sobre o terminal.

Antigamente, quando o computador era feito de pedra e ainda não existia uma interface gráfica (GUI) para interagirmos com a máquina, a maioria de nós tinha que dizer ao computador o que fazer através de alguns comandos básicos que eram digitados numa interface do tipo terminal, ou, se preferir, uma interface de linha de comando (CLI - command line interface). Criar diretório, apagar, renomear ou criar um arquivo eram tarefas triviais e executadas por estas linhas de comando, e não por cliques do mouse. Vivíamos muito bem com isso. Porém, quando o Windows chegou (vale lembrar que o Windows não foi o primeiro sistema operacional com interface gráfica, mas foi o que dominou o mercado de computadores pessoais), a facilidade de executar as mesmas tarefas com poucos cliques do mouse fez com que rapidamente esquecêssemos de tais comandos.

Ainda hoje é possível digitar os mesmos comandos pelo prompt de comando do Windows (No mac ou linux, não há prompt de comando, e sim o terminal, que é quase a mesma coisa). Faça um teste, se você já instalou o NodeJS, abra o prompt de comando (Aperte a tecla Windows + R, digite `cmd` e aperte Enter) e digite `node -v`. Você deverá ver no output do terminal a versão do node instalado em sua máquina.

Agora que você já sabe como digitar comandos no prompt do Windows, devo alertá-lo: Não o faça mais. Nós nunca mais iremos usar o prompt de comando do windows (cmd) para esta tarefa.

A princípio parece que usar o prompt de comandos do windows é uma boa alternativa, uma vez que já está instalado na máquina, prontinho para uso. Mas o fato é que o CMD é um tipo de terminal que aceita apenas os comandos usados no antigo sistema operacional MSDOS, e não aceita os comandos usados em sistemas Unix.

Há muitos desenvolvedores que não utilizam o Windows como sistema operacional. Portanto, uma boa prática é usar um terminal que converse a mesma língua de outros sistemas operacionais. Isso vai fazer você evitar problemas de compatibilidade e ser um profissional mais preparado para trabalhar com outros sistemas além do Windows. Convenci?

Existem várias interfaces do tipo terminal, mas nós vamos usar uma que é instalado no momento que começarmos a versionar nossos arquivos usando o GIT. Nós usaremos um cara chamado Git Bash. O motivo é que com o Git Bash nós podemos simular comandos Unix no Windows. Acredite, você irá querer isso.

Torne-se front-end com este curso 100% prático


+ **12 horas** de vídeo


- Apenas **R\$ 20,00** *

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

 serfrontend



No próximo capítulo, conheceremos o básico sobre versionamento, instalaremos o git para Windows, conheceremos alguns comandos básicos no git bash e criaremos nosso primeiro repositório no GIT.

Até lá!

Versionamento e GIT

Tanto no desenvolvimento de sistemas para web quanto na criação de websites, é muito importante que você saiba versionar os seus arquivos. Mesmo que você seja uma pessoa regrada e que faça backup do seu site/aplicativo todos os dias em Pendrivers ou deixe-os na nuvem, como o DropBox, o gerenciamento dessas versões é impraticável. Você nunca se lembra o que foi feito na semana passada e se algo quebrar, você precisa comparar os arquivos manualmente para descobrir o código que gerou o bug.

Versionar arquivos é uma tarefa básica e trivial que todos nós devemos ser capazes de executar sem pestanejar.

Se você usa Windows, eu tenho uma notícia boa e uma má para você. A boa: há uma ferramenta que faz o versionamento através de uma interface gráfica muito elegante. É bem simples mexer e você adoraria trabalhar com ela. A má: nós não vamos nem chegar perto dessa ferramenta. Motivo: no começo, eu também me senti atraído por poder versionar meus arquivos sem a necessidade de digitar linhas de comandos. O problema é que com o tempo você percebe que precisa ter um maior controle sobre esse versionamento, e esta ferramenta que deveria facilitar a tua vida acaba se tornando um problema. Além do mais, trabalhar com o Git por linha de comando é moleza, você vai ver.

Dito isto, quero explicar a base sobre os sistemas de versão.

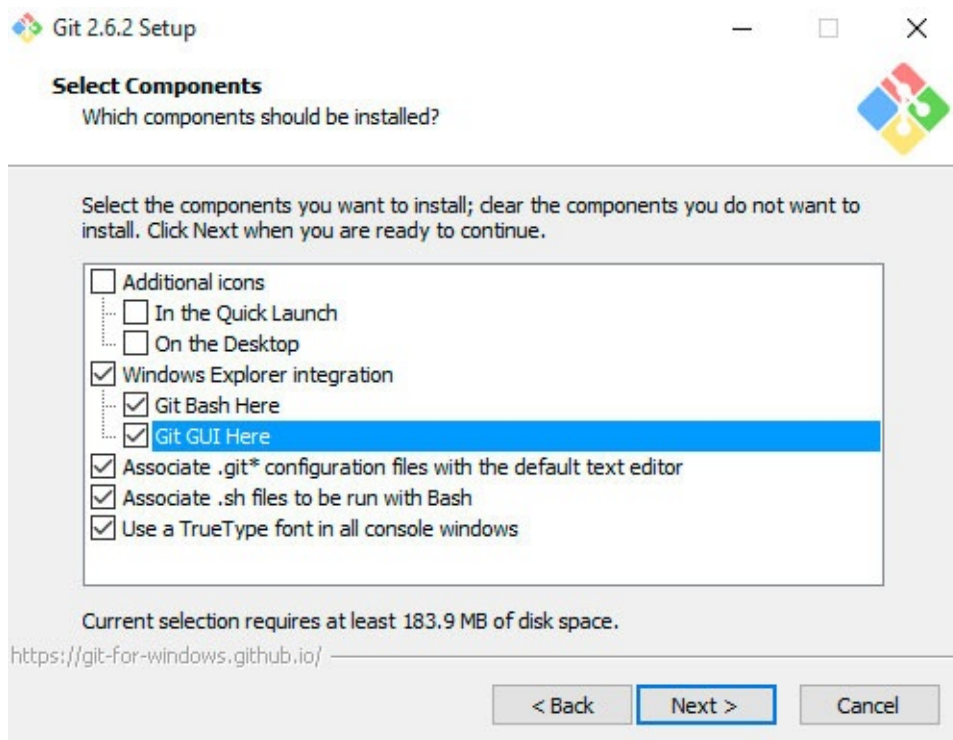
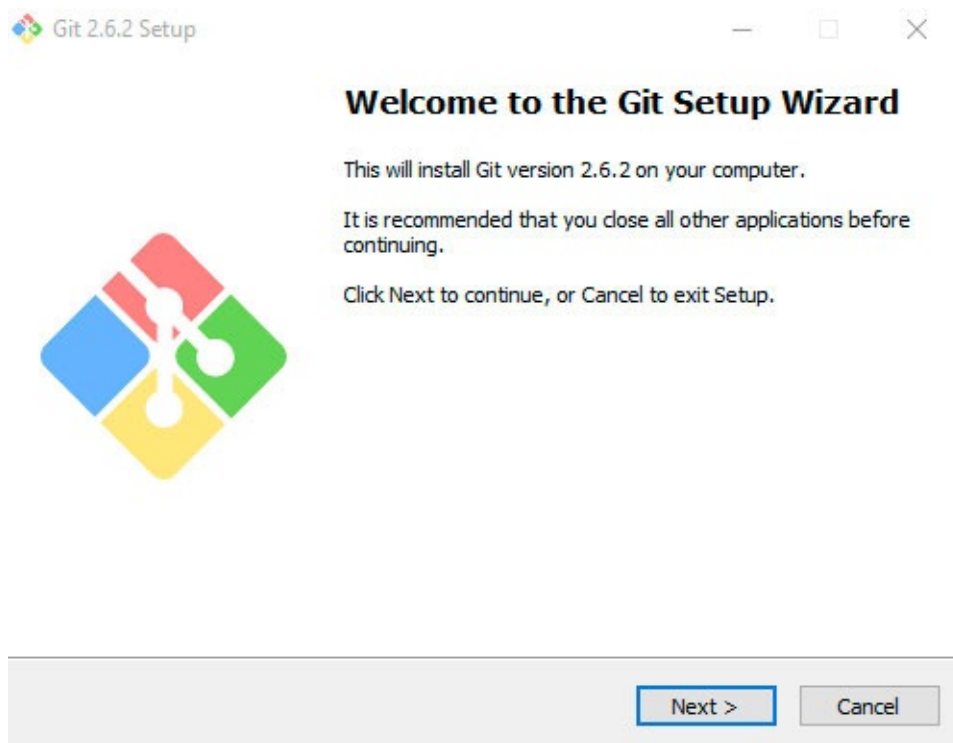
Um sistema de versionamento que se preze precisa efetuar bem duas tarefas:

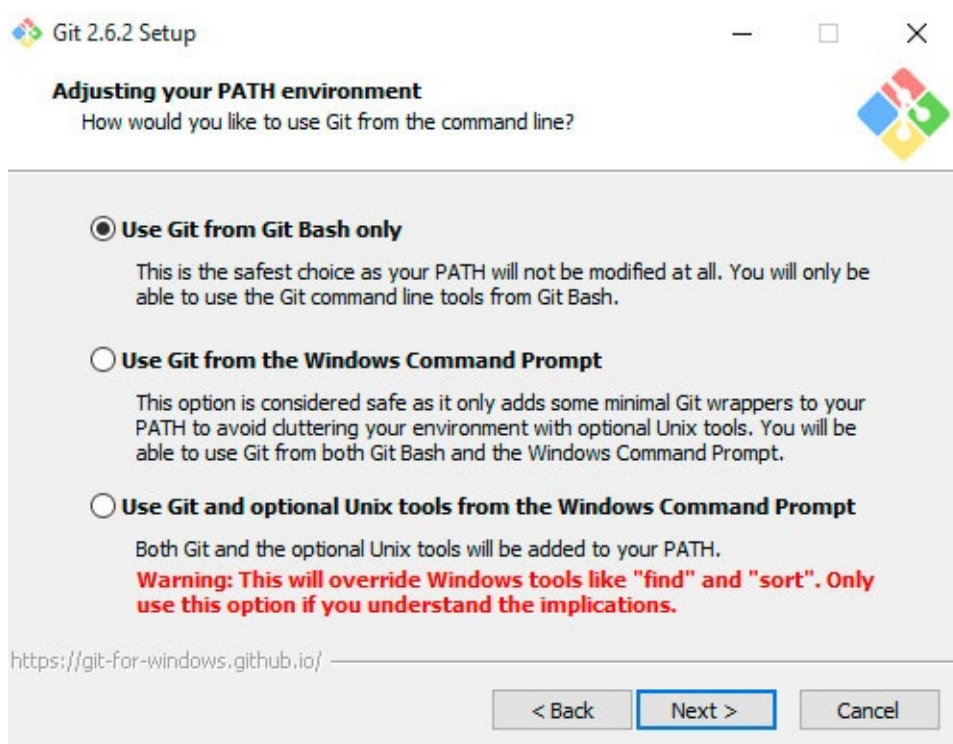
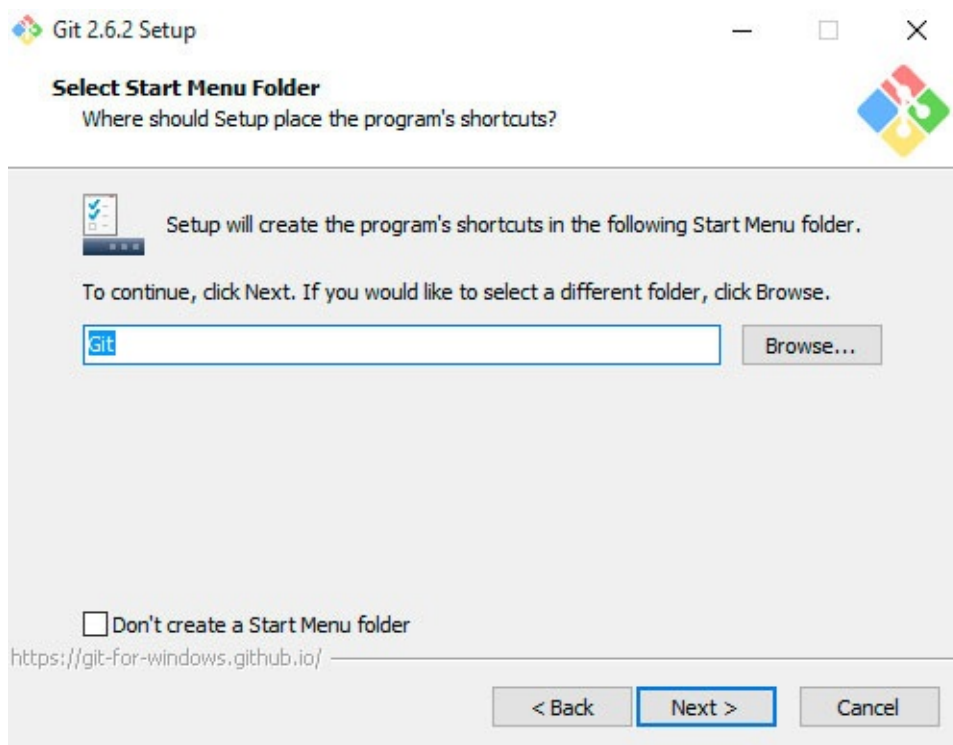
1. Armazenar registros temporais da evolução de seu sistema. É como se você conseguisse tirar uma fotografia do seu projeto num determinado instante e inserir uma legenda nessa fotografia. Essa 'legenda' é chamada de commit. Cada commit é um registro do seu projeto numa linha de tempo.
2. Gerenciar conflitos em um mesmo arquivo se este for editado por mais de uma pessoa ao mesmo tempo. Muitas vezes o Git é capaz de lidar com esses conflitos de maneira espetacular. Outras vezes o Git não sabe qual é a versão correta, e então os desenvolvedores envolvidos devem decidir qual é a versão que deve ficar para a história do projeto.

Instalando

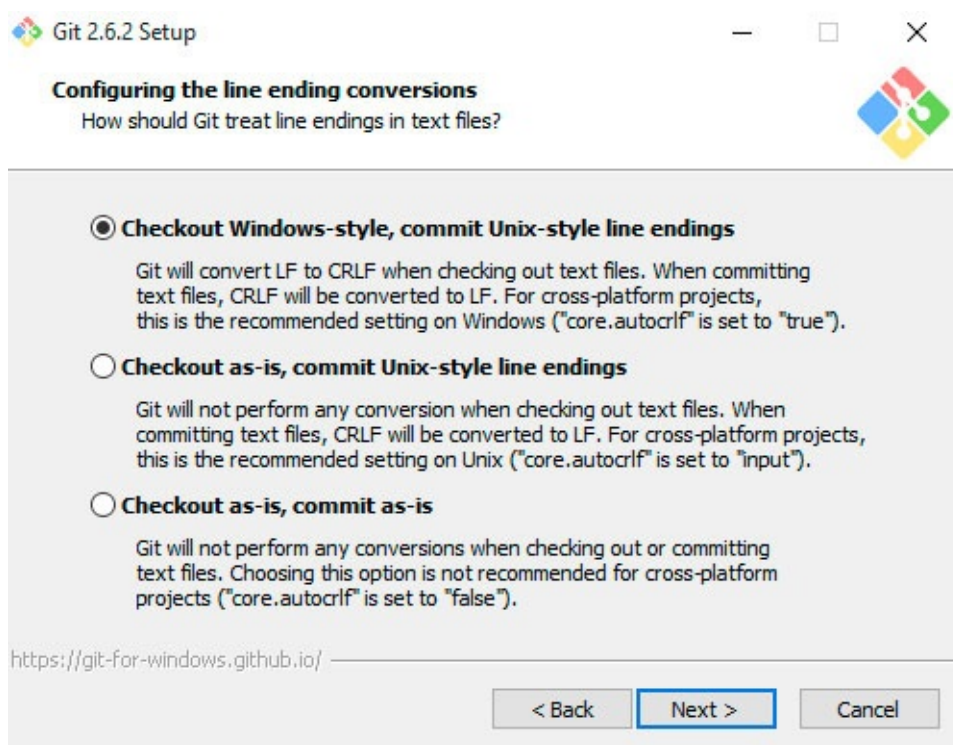
Para instalar o Git para windows, acesse <https://git-for-windows.github.io/> e faça do download do executável. No momento da escrita deste texto, o Git para windows estava na versão 2.6.2.

Siga as imagens abaixo para terminar sua instalação

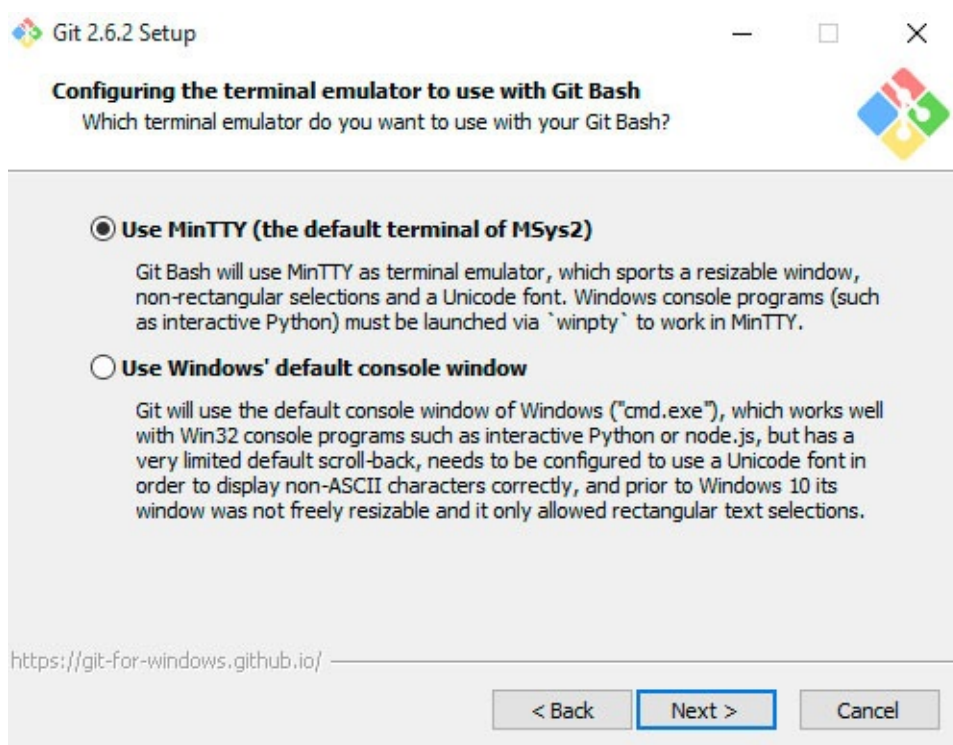


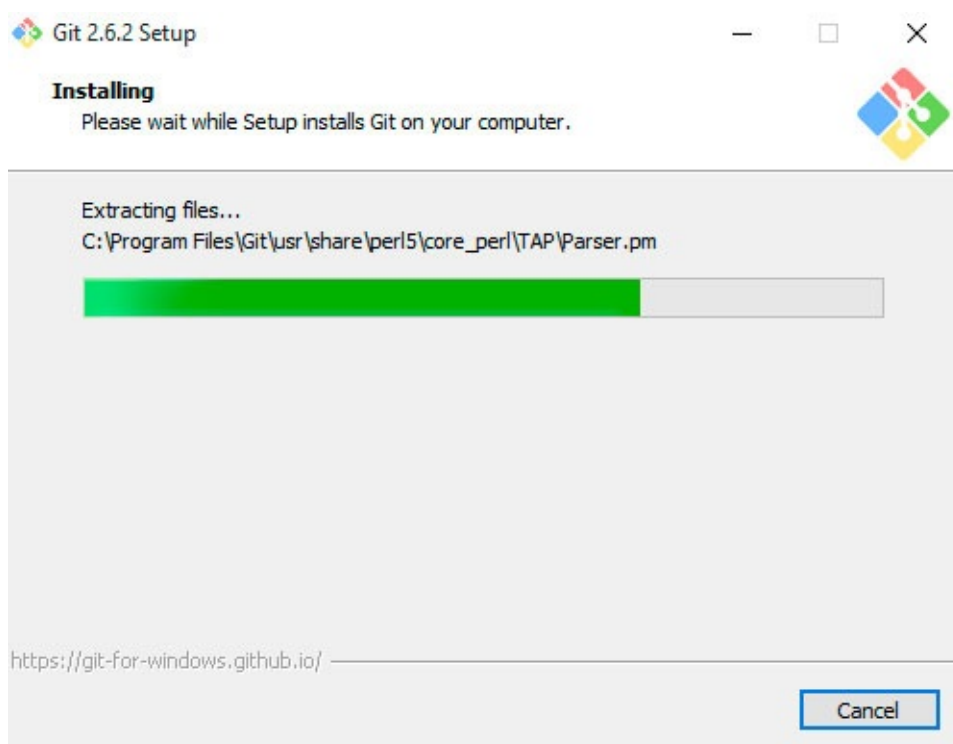
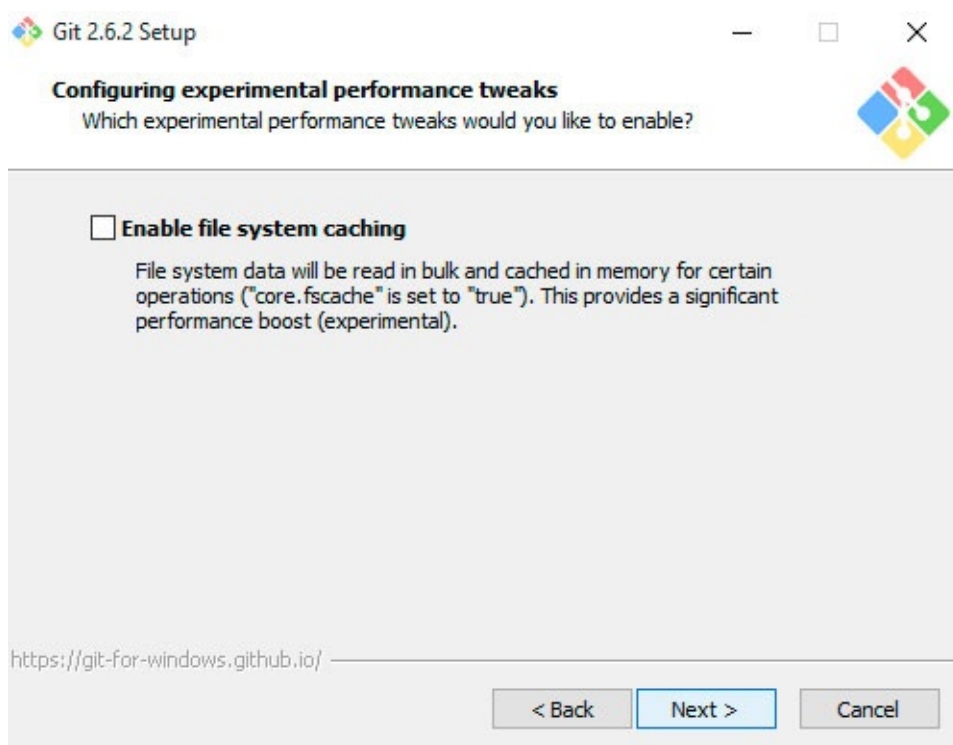


Na tela acima deixe marcada somente a primeira opção. Isto é mais seguro, pois fará que o git seja usado somente a partir do Git Bash.



Já a tela acima é sobre quebra de linhas. Como os sistemas operacionais possuem formatos diferentes de quebra de linhas em arquivos de texto, deixar marcada a primeira opção normaliza isto. Explicando: a primeira opção converte para o padrão Windows quando os arquivos chegam até você. Quando você efetuar um commit será convertido para o formato Unix. A segunda não faz nenhuma conversão quando o arquivo chega, apenas converte para Unix quando comitamos. Já a terceira opção não faz nenhum tipo de conversão. Deixe marcada a primeira opção.





Agora que temos o git instalado e configurado, na próxima seção vamos dar uma olhada rapidamente em alguns comandos Unix para manipulação de arquivos.

Torne-se front-end com este curso **100% prático**

+ **12 horas** de vídeo

- Apenas **R\$ 20,00** *

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20



Comandos básicos Unix para manipulação de arquivos

Você já efetuou várias tarefas de manipulação de arquivos usando um interface gráfica. Já criou, deletou, renomeou arquivos e pastas e já moveu arquivos dentro do seu HD. Você está tão acostumado a fazer isso com o mouse que parece uma idiotice ter que aprender a fazer isso via linha de comando. A primeira impressão é que você perderá muito mais tempo digitando comandos extensos do que simplesmente dando alguns cliques no mouse e/ou alguns atalhos do teclado. Talvez você esteja certo. Mas talvez não. Mesmo que se no final deste capítulo você continuar usando a interface gráfica para efetuar estas tarefas, é bom conhecer outras possibilidades. O fato é que saber operar o terminal é primordial se você quiser evoluir na área de desenvolvimento front-end, e conhecer o mínimo de manipulação de arquivos é um excelente ponto de partida.

Uma vez que você instalou o Git for Windows na sua máquina da maneira como eu descrevi na seção anterior, você ganhou de presente o "Git Bash", que é um programa do tipo CLI (command line interface) que simula um terminal do Unix no Windows, além de permitir que você digite os comandos do Git.

Abra o Git Bash de maneira trivial, procurando-o no menu iniciar ou em algum atalho criado na instalação.

A primeira observação importante é que, para cada usuario presente na máquina há uma pasta chamada "pasta pessoal" (também conhecida como pasta "Home"). No Windows, essa pasta fica em `c:/Users/Seu_usuario`. Ao abrir o Git Bash pela primeira vez você estará "visualizando" essa pasta, muito provavelmente.

Uma outra maneira de abrir o Git Bash a partir do Windows é clicar com o botão direito do mouse dentro de um diretório qualquer e selecionar a opção "Git Bash Here". Nesse caso você não estará visualizando a sua pasta pessoal, mas sim a pasta a partir da qual você abriu o Git Bash.

A segunda observação importante é que você provavelmente está vendo algo parecido com:

```
seu_usuario@host MINGW64 ~ $
```

Vamos entender o que é cada informação acima.

- Antes do arroba é o seu usuário logado no sistema.
- Depois do arroba significa o nome da sua máquina, ou host.

- MINGW64 é uma variável de sistema. Não se preocupe com ela.
- ~ é um atalho para sua pasta "Home". (é o mesmo que sua pasta pessoal, mas o nome "Home" é mais correto, pois é assim que é chamado em ambientes Linux). Se você não estiver vendo esse sinal de til, provavelmente você abriu o terminal com o botão direito do mouse sobre um diretório qualquer. Para acessar sua pasta Home, digite `$ cd ~` e aperte o enter.
- O sinal de cifrão significa que você é um usuário padrão da máquina.

Para ter certeza de qual diretório está sendo visualizado, digite no terminal (A partir desse ponto, sempre que eu me referir ao terminal, ou linha de comando, estarei falando do Git Bash, ok?):

```
$ pwd
```

Esse comando significa "print working directory" e serve para visualizar a pasta corrente

Se você quiser visualizar os arquivos e pastas não ocultas no diretório corrente, basta digitar:

```
$ ls
```

Esse comando é parecido com o comando 'dir' do MSDOS, porém, estamos simulando um ambiente Unix e o comando dir não existe nesse ambiente.

Se você quiser entrar em alguma pasta, basta digitar o comando `cd` (change directory). Por exemplo:

```
$ cd Desktop
```

A partir desse ponto você está visualizando a sua área de trabalho, experimente efetuar novamente os comandos `ls` e `pwd` nessa pasta.

Se você quiser voltar um nível, basta digitar

```
$ cd ..
```

O sinal ".." significa "um nível acima". Nesse ponto estamos novamente acessando a pasta pessoal. Tente voltar dois níveis de uma vez:

```
$ cd ../../
```

Agora estamos visualizando o C:. Para voltar para a sua pasta pessoal, você poderia digitar `cd Users/seu nome` ou simplesmente usar um atalho, como a seguir:

```
$ cd ~
```

o sinal "til" é um atalho que aponta para o seu diretório Home, lembra?

Se você quiser descobrir a pasta pessoal de qualquer diretório ou partição no seu computador, basta digitar :

\$ echo ~

Esse comando te mostrará qual é a pasta pessoal mas não "navega" até ela

Criar um diretório onde salvaremos todos os exemplos desse livro.

Eu recomendo que para os exemplos desse livro, você salve seus arquivos no seu diretório Home. No futuro, nos projetos da vida real, fique a vontade para trabalhar na estrutura de arquivos que você preferir. O motivo é que ao trabalharmos no diretório Home, nós temos a mesma estrutura de pastas e arquivos, sendo assim ficará mais fácil e mais didático acompanhar a leitura deste guia.

Antes de criarmos o diretório principal, tenha certeza de que você está no lugar certo, digite

```
$ cd ~
```

Como vimos, esse comando fará você acessar a sua pasta pessoal, independente de qual diretório estiver sendo acessado anteriormente.

Confirme isso digitando `$ pwd`

Nesse ponto, vamos criar um diretório que conterá todos os exemplos deste livro. Digite: `$ mkdir Projects`

Esse comando criará um diretório chamado Projects. Para ter certeza que foi criado corretamente. `$ ls`

Você deverá visualizar a pasta "Projects" na lista que aparecer no terminal. Criamos o diretório, mas ainda não estamos "dentro" dele. Como fazemos mesmo para trocar o diretório corrente? `$ cd Projects`. Isso mesmo. Mas nesse ponto eu quero te dar uma dica. O Git bash possui um autocomplete que facilita muito a nossa vida. Para vê-lo em ação, tente digitar: `$ cd Pro` (e sem continuar a palavra, aperte TAB do teclado. Faça uns testes com `$ cd P` (TAB), `$ cd Pr` (TAB). Veja o que acontece. Muito útil.

Será que está tudo certo? Se estiver no diretório Projects, digite `$ pwd` para vermos o diretório corrente. Observe que, como este diretório está vazio, digitar `$ ls` não produzirá nenhum resultado.

Já temos o básico necessário para começarmos a versionar nossos arquivos com o Git. Mas é claro que há muito mais comandos Unix para aprender. Se você quiser estudar por conta própria, sugiro alguns comandos iniciais: cp, mv, rm e rmdir. Com esses comandos você consegue fazer tudo o que fazia com o mouse e o Windows, só que agora via terminal. Não é só uma questão de se sentir cool. Mais cedo ou mais tarde você precisará editar um

arquivo diretamente no servidor, e para fazer isso, você precisará de um editor de código do tipo terminal, como o VIM ou o Emacs (dê tchauzinho ao seu Sublime Text nessas situações).

Configurando e Iniciando o Git

Os passos a seguir precisam ser efetuados somente uma vez a cada instalação do Git.

No terminal, digite:

```
$ git config --global user.name "Seu nome"
```

```
$ git config --global user.email "seu e-mail"
```

Como o Git é um programa de versionamento, normalmente usado para trabalhos em equipe, é necessário se identificar para que ele saiba quem é o dono de cada commit (lembre-se que um commit é um registro temporal - uma fotografia - do site/aplicativo).

Pronto, isso basta para começarmos a versionar nossos arquivos.

Iniciando um repositório local

Estando no diretório Projects, vamos criar mais um diretório chamado "PrimeirosPassosWorkflow" e acessá-lo em seguida. (importante: nunca coloque espaços ou caracteres especiais, como acentos ou cedilhas em nomes de arquivos ou pastas)

```
$ mkdir PrimeirosPassosWorkflow
```

```
$ cd primeirosP (TAB + ENTER)
```

Agora, nós precisamos pedir ao Git para tomar conta dessa pasta pra gente, e ele gentilmente cuida para que nenhum arquivo ou versão se perca no decorrer do tempo. A não ser que a sua máquina pegue fogo ou o seu HD resolva fritar. Mas ainda assim há serviços que permitem você colocar suas versões na nuvem. Tem um serviço desses que um dia ficará famosinho. É um tal de GitHub. conhece? :-P

Voltando à nossa realidade, precisamos dizer ao Git que cuide no nosso diretório corrente (`$ pwd` , ok?).

Estando no diretório que queremos versionar, vamos digitar:

```
$ git init
```

Ao fazermos isso, aparece uma mensagem no terminal informando que um repositório vazio foi criado. Isso significa que agora uma pasta `.git` foi criada no diretório PrimeirosPassosWorkflow.

Ao digitarmos `$ ls` esperaríamos ver o diretório mencionado acima. Mas isso não acontece. Por quê? O motivo é que o diretório `.git` é oculto (esse "." inicial significa que é um arquivo ou diretório oculto).

Porém, ao digitarmos `$ ls`, podemos perceber que os arquivos ocultos não estão listados, mas eu garanto que ele está lá. Quer ver? Então digite: `$ ls -a`

Com isso aparece no terminal algo como: `. .. .git`

O primeiro "." significa o diretório corrente. O ".." significa diretório acima (nível acima, diretório pai, como você preferir). Nós já sabemos do que se trata o último ".git".

Dica 1: o nome correto para esse símbolo "-a" que digitamos acima é flag a. As flags servem para passar informações adicionais aos comandos via terminal.

Dica 2: não altere nada dentro da pasta ".git". Dizem que ao fazer isso um urso coala morre.

Versionando nosso primeiro arquivo

Nesse ponto já temos o Git cuidando do nosso diretório, ou seja, versionando nossos arquivos.

Vamos então criar nosso html simples com um 'Olá mundo git'.

Para isso, vou abrir a minha pasta diretamente no Sublime para que eu consiga visualizar meus arquivos diretamente no painel lateral deste editor.

Vou criar um novo arquivo nesse diretório clicando com o botão direito do mouse e salvar como exemplo01.html. Segue o código:

```
<!doctype html>
<html lang="pt-br">
  <head>
    <meta charset="utf-8">
    <title>exemplo 01</title>
  </head>
  <body>
    <h1>Primeiro exemplo com o Git</h1>
  </body>
</html>
```

Agora vamos pensar. Eu tenho um diretório que está sendo monitorado pelo Git chamado PrimeirosPassosWorkflow . Eu incluí um arquivo dentro desse diretório, logo, o Git começou a versioná-lo. Certo? Errado. Essa é a primeira pegadinha do Git. O fato de termos um repositório git, ou seja, um diretório sendo monitorado, não versiona arquivos criados automaticamente. É preciso que para cada arquivo incluído seja informado ao git para monitorar também esse arquivo. Mais para frente vamos ver uma maneira de incluir vários arquivos de uma vez, mas por hora, vamos provar isso que eu acabei de dizer. Digite no seu terminal:

```
$ git status.
```

Você verá uma mensagem informando que há um arquivo "untracked", ou seja, este arquivo está num repositório do git, mas não está sendo "rastreado". Vamos corrigir isso incluindo esse arquivo numa área especial dentro do git responsável por "fotografar" seus arquivos a cada commit. A essa área especial chamamos de "stage". Digite:

```
$ git add exemplo01.html e tecla ENTER
```

Novamente, digite `$ git status` para entendermos o que aconteceu.

Agora podemos ver que esse arquivo está pronto para ser fotografado, isto é, comitado (O verbo comitar, assim como googlar ou twittar, não existem de fato, mas é muito comum no nosso meio).

Legal! Temos uma pasta que transformamos num repositório git (`git init`), informamos que queremos rastrear as mudanças feitas no nosso primeiro arquivo, colocando-o na área de stage (`git add`), mas até agora este arquivo não foi comitado de fato. Para resolver isso, digite no seu terminal:

```
$ git commit -m "primeiro comite lindo"
```

a flag -m é usada para informar uma mensagem de commit. É muito importante que as mensagens de seus commits sejam descritivas e específicas das tarefas executadas naquele momento. Por exemplo, "alteração de endereço na página de contato", "inclusão da funcionalidade buscar cep" ou "alterar banner da home - campanha natal 2016" são bons nomes de commits pois permitem facilmente identificar o momento em que as alterações foram efetuadas. Ao passo que: "correção de bugs diversos", "nova funcionalidade na página de compras" ou "incluir banner" são poucos específicos e de pouco valor. Fique atento às suas mensagens de commits.

Prontinho. Nosso primeiro commit foi feito. Vamos verificar:

```
$ git status
```

 (dica: você pode executar comandos escritos recentemente no terminal apertando as teclas para cima e para baixo do teclado).

Na saída do terminal, a última linha é a que nos interessa. "nothing to commit, working directory clean". Ou seja, tudo está atualizado.

Vamos agora voltar ao Sublime, criar um paragrafo qualquer no nosso arquivo html e salvar.

```
<!doctype html>
<html lang="pt-br">
  <head>
    <meta charset="utf-8">
    <title>exemplo 01</title>
  </head>
  <body>
    <h1>Primeiro exemplo com o Git</h1>

    <p>um texto qualquer</p>
  </body>
</html>
```

Vamos investigar como está o nosso repositório git nesse momento (depois de alterar e salvar, não se esqueça, salvar é muito importante).

```
$ git status
```

O git nos dá duas informações muito importantes. A primeira é que há mudanças que não estão na área de stage, portanto, ainda não estão prontas para o próximo commit. A segunda é uma consequência da primeira, ou seja, não há mudanças para serem comitadas.

Essa é a segunda pegadinha do Git. Ao alterarmos um arquivo que foi comitado previamente, precisamos incluí-lo novamente na área de stage e só depois podemos comitá-lo. Então vamos lá:

```
$ git add exemplo01.html
```

```
$ git commit -m "inclusao de paragrafo".
```

Agora que nós temos o nosso primeiro html versionado, quero que você execute algumas tarefas:

1. criar um arquivo de estilo css/estilo.css
2. criar uma arquivo de javascript: js/app.js
3. criar um conteúdo qualquer dentro desses arquivos
4. vincular os dois arquivos à sua página html.

Para referência, segue um exemplo bem bobinho que fiz apenas para mostrar:

estilo.css:

```
body{  
    background: #ccc;  
}
```

app.js

```
var body = document.body;  
body.style.color = 'red';
```

Nesse ponto vale uma dica: com o comando `ls` é muito difícil identificar o que é arquivo e o que é diretório. Faça um experimento, digite no terminal `$ ls --color` e veja o que acontece.

Depois de vinculá-los à sua página, digite `$ git status` e analise o output no terminal. Se tudo estiver bem, você deverá ver que um arquivo já comitado anteriormente foi modificado, porém, os dois arquivos criados não estão sendo rastreados pelo Git, ou seja, não estão sendo versionados. Como são apenas dois arquivos, você poderia incluí-los um a um com o comando `$ git add`. Porém, pense no caso de termos muitos arquivos, como fontes, imagens, audios, vídeos etc. Nesse caso, incluí-los um a um faria você desistir do versionamento nos primeiros dez arquivos.

Para incluir todos os arquivos de uma vez, você poderia digitar o seguinte comando: `$ git add .`

Observe o ponto no final. Esse sinal é o mesmo que dizer "todos os arquivos".

Faça o teste: digite o comando acima e em seguida veja o que acontece quando você verifica o status novamente.

Com tudo adicionado à área do stage, que informa que os arquivos estão preparados para serem comitados, basta você efetuar o comite normalmente

```
$ git commit -m "Inclusão de javascript e css"
```

Fácil, né? Incluímos todos os arquivos que não estavam na área de stage de uma única vez e depois comitamos normalmente. Contudo, digamos que depois do comite efetuado, o cliente pediu algumas alterações que precisam ser efetuadas tanto no arquivo de estilo quanto no arquivo de javascript. Como queremos manter o histórico, vamos alterar esses arquivos e efetuar um novo comite. Então vai lá. Altere os seus arquivos (pode ser qualquer alteração mesmo, ok).

Pronto? agora digite `$ git status` novamente. Repare que Git nos mostra no output do terminal que dois arquivos (previamente adicionados à área do stage) foram modificados. E também nos dá uma dica do que podemos fazer: podemos digitar `git add` ou `git commit -a` para resolver a questão.

O comando `git commit -a` mata dois coelhos com uma cajadada: acrescenta as modificações à área de stage e comita de uma só vez, porém, o jeito certo de usar esse comando é um pouco diferente. As três linhas de comando abaixo são equivalentes. Fica a seu gosto

```
$ git commit -a -m "mensagem do comite".
```

(a flag `-a` significa "todos os arquivos" e `-m` significa que você quer digitar uma mensagem no comite).

```
$ git commit -am "mensagem do comite".
```

(é uma abreviação da linha acima, juntando as duas flags)

```
$ git commit -all -m "mensagem do comite".
```



(um pouco mais verboso, mas é exatamente a mesma coisa que as duas linhas acima).

Mas atenção: a flag -a, ou uma das variações, só funciona se os arquivos alterados foram versionados previamente (com o `$ git add` ou `$ git add .`). Se você nunca adicionou os arquivos à área de stage, a flag -a não produzirá resultado algum.

Torne-se front-end com este curso **100% prático**

- + **12 horas** de vídeo
- Apenas **R\$ 20,00 ***
- Acesso **Vitalício**
- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20



Avaliando o histórico

Se tudo correu bem, já temos algumas versões de nosso arquivo. Vamos verificar digitando no terminal:

```
$ git log
```

Com esse comando, podemos ver o histórico de commits já efetuados. Poderíamos também ver uma versão resumida, passando mais uma flag `--oneline`

```
$ git log --oneline
```

Se nosso projeto tivesse muitos comites efetuados, o comando acima nos traria uma lista enorme. Uma opção seria limitar o número de itens mostrado pelo comando `$ git log`. Para isso, basta passar mais uma flag, informando quantos comites queremos ver. Até o momento temos apenas dois comites em nosso repositório, portanto, para ver isso funcionando, digite em seu terminal:

```
$ git log -n 1
```

Esse comando nos traz apenas o último commit. Se quisermos, por exemplo ver os últimos três, basta digitar:

```
$ git log -n 3
```

E também dá para unir as duas flags. Experimente:

```
$ git log --oneline -n 1
```

Dica: Se você tiver vários comites, é provável que veja um sinal de dois-pontos na última linha do terminal ao digitar `$ git log`. Esse sinal é um indicativo que há mais linhas para mostrar no output. Para vê-las uma a uma, basta apertar o ENTER do teclado. Para sair dessa visualização, basta digitar a letra "q".

Fácil, não? Vimos o básico sobre o Git. Daria pra falar muito mais sobre ele, como por exemplo criar branches, criar releases, fazer merge entre branches, resolver possíveis conflitos entre arquivos. Mas vamos ter que parar por aqui. Espero que com essa base você possa continuar seus estudos sem se assustar com o terminal e podendo dar os primeiros comandos do Git.

Há várias maneiras de comparar as modificações de um arquivo usando o Git. A que mais me agrada é utilizar uma interface gráfica para isso. Para abrir esse programa, digite no terminal `$ gitk`. O Gitk é um programa a parte específico para visualizar as alterações nos

arquivos. Dê uma fuçada nele. É bem tranquilo.

No próximo tópico vou falar um pouco sobre o GitHub. Vamos criar um repositório na nuvem e mandar nossos arquivos pra lá, deixando nossos arquivos visíveis para o mundo inteiro. O cloud é o limite.

Fundamentos do GitHub

Ainda me lembro como se você ontem. Foi no final de 2012 que desenvolvi meu primeiro plugin jQuery. O plugin era muito simples e resolvia um problema que frequentemente eu enfrentava: igualava as alturas de vários elementos flutuantes de acordo com a altura do maior elemento na mesma linha. Hoje há formas de resolver este problema apenas com o CSS, mas naquela época, me facilitou muito a vida e resolvi compartilhar o meu feito com o mundo inteiro.

E onde deveria "hospedar" o meu código? Já havia usado o GitHub para "baixar" javascripts de terceiros, portanto, "subir" meu próprio código não deveria ser difícil.

Depois de me cadastrar no site, procurei imediatamente um botão para upload, sem sucesso. Não havia (e não há um botão upload de arquivos no GitHub). Tudo bem, talvez eu tenha sido o único idiota que tenha procurado esse botão no GitHub, mas naquele momento, a minha vida teria sido mais fácil se alguém mais experiente tivesse me explicado:

"O GitHub não é um lugar para você hospedar seus arquivos lá. Ele é um serviço que possibilita você deixar o seu repositório GIT na nuvem."

Assim no Git como no GitHub.

Até o capítulo passado, todo o versionamento (`git init` , `git add` , `git commit` etc) era efetuado na sua máquina. Isso resolve um problema: se algo der errado, é fácil voltar uma versão ou até mesmo comparar arquivos antes e depois de cada commit. Mas isso ainda não resolve alguns problemas:

- Se você trabalhar em equipe, cada desenvolvedor terá a sua própria versão. O que não é o ideal.
- Mesmo trabalhando sozinho, o ideal é que você não mantenha o seu versionamento de seu sistema somente na sua máquina. Assim, mesmo que a sua máquina pegue fogo, você poderá continuar trabalhando de qualquer outro lugar.

O GitHub não é a única solução para estes problemas. Você poderia, por exemplo, manter um repositório central no seu servidor e fazer todos os colaboradores clonarem os projetos de lá. É uma solução válida, contudo, pela facilidade, mostrarei aqui como manter seu repositório no GitHub. Entretanto, vale alertar que o GitHub é gratuito para contas com projetos públicos. Se você precisar de um projeto privado, precisará de uma conta paga.

Dito isto, o primeiro passo é criar sua conta em <https://github.com/join>.

Preencha o formulário com seus dados e depois escolha um dos planos. Pode criar uma conta gratuita para testarmos.

Depois da conta criada, você precisará criar um repositório de teste. Do lado direito, procure uma caixa "Your repositories" e clique no botão "upload". Brincadeira!. Clique em "new repository". Dê um nome bonito para o seu repositório e lembre-se de deixá-lo como público. As demais configurações você pode deixar como está. Clique em "create repository".

Prontinho. Você tem seu primeiro repositório (ainda vazio) criado. Agora vamos fazer a ligação entre o seu repositório local e o remoto.

O primeiro passo é configurar o repositório corretamente usando o terminal.

Acesse a pasta do nosso exemplo digitando no Git Bash:

```
$ cd ~/Projects/PrimeirosPassosWorkflow
```

Lembre-se que o "~" é um atalho para a sua pasta pessoal.

Estando na pasta correta, digite o seguinte comando no seu terminal:

```
$ git remote add origin https://github.com/Seu_usuario/seu_repositorio.git
```

Repare que você precisa trocar o seu usuário e o nome do seu repositório no endereço acima. Se você não tiver certeza de ter escrito corretamente, procure na página inicial do seu repositório o lado direito onde está escrito: HTTPS clone URL

É essa URL que você precisa escrever no seu terminal (comando `$ git remote`)

Com o comando anterior, você criou um vínculo do nome "origin" para o seu repositório que está lá no GitHub.

Git Push e Pull

Uma vez criado o nosso apontamento do "origin" para o GitHub, o próximo passo é "subir" seus arquivos para a nuvem. Tendo certeza que você está na pasta correta, digite no terminal:

```
$ git push origin master
```

Na verdade, nós não estamos apenas "subindo" nossos arquivos. Nós estamos enviando todo o histórico do versionamento. Explicando o comando acima: `push` é o comando que submete o versionamento para o seu repositório na nuvem. `origin` é o seu repositório

remoto que, como dito antes, aponta lá para o GitHub. `master` é a sua branch principal. Não cabe aqui explicar em detalhes o que é uma branch, mas saiba que o nome "master" é um nome padrão criado pelo próprio Git. Recomendo deixar esse nome por enquanto.

Digite seu usuário e senha do GitHub e tecele ENTER. Se tudo correu bem, você já consegue ver sua versão na internet. https://github.com/Seu_usuario/seu_repositorio

Digamos agora que a sua máquina subiu no telhado e você precisa continuar o desenvolvimento da sua página exemplo01.html. Se você fez tudo certinho (`git commit` e `git push` nos momentos adequados) basta você "baixar" seus arquivos de outra máquina (pode até ser emprestada) que contenha o Git instalado.

Para isso, crie uma pasta nesse novo computador que será o seu repositório local de todos os projetos (similar à nossa pasta "Projects") e a acesse no terminal com o comando `cd` . Depois, digite:

```
$ git clone https://Seu_usuario/seu_repositorio.git
```

Esse comando fará uma cópia dos arquivos que estão no seu repositório lá no GitHub num diretório com o mesmo nome de seu repositório (veja com o comando `ls`). Agora não tem mais desculpa. Pode continuar o desenvolvimento do projeto. Quando tudo tiver feito (ou no final do dia de trabalho), faça um `commit` e suba tudo novamente no GitHub (`$ git push origin master`). Assim você garante que não haverá problema de perda de arquivos (e horas de trabalho) se a sua nova máquina resolver subir no telhado também.

Maravilha. Você continuou o trabalho usando uma máquina emprestada sem prejuízo algum. Mas a sua outra máquina foi para o concerto e ficou pronta. Tudo funcionando perfeitamente. Então você agradece e devolve a máquina emprestada e volta para a sua própria máquina. Como continuar com o desenvolvimento? Será preciso clonar novamente o repositório? Não. Se este ainda estiver configurado no computador, basta solicitar ao Git para atualizar os arquivos locais de acordo com o último `commit` que foi feito no GitHub. Para isso, digite no terminal (depois de acessar a sua pasta com o comando `cd`).

```
$ git pull origin master .
```

Prontinho, não só os arquivos na sua máquina estão todos atualizados como você ainda mantém todo o histórico. Faça um teste digitando `$ git log` .

Por hora é isso. Ainda há muito o que aprender sobre o Git mas o que eu apresentei aqui é o básico necessário para você continuar seus estudos.

Atualmente, saber manusear bem os comandos do Git (ou outro sistema de controle de versão) é requisito obrigatório em qualquer área de desenvolvimento (web, app, mobile, front-end, back-end etc).

Se você não conhecia nada sobre o assunto, eu espero que tenha conseguido te dar uma luz. Se já conhecia, espero que tenha acrescentado algo novo.

No próximo capítulo vou mudar um pouco de assunto e falar sobre npm e automatização de tarefas. Espero que continue a leitura. Até lá!

Turbinando o NodeJS

Se você acompanhou a leitura desde o princípio, você está apto a dar continuidade nos estudos e entender melhor o que é o [NodeJS](#) e o [npm](#). Para isso, vamos resumir o que aprendemos até esse ponto:

- Já temos instalado o NodeJS em nossa máquina, por conseguinte, ganhamos de presente o npm, que é o gerenciador de pacotes do node.
- Também vimos alguns comandos básicos do Unix e até testamos estes comandos no Git Bash, que é um programa do tipo terminal que simula comandos Unix no Windows.
- Instalamos o Git na nossa máquina, que é um software de versionamento, responsável por armazenar o histórico de desenvolvimento do nosso site/aplicativo. Até já criamos alguns commits usando o terminal.
- Criamos um repositório no github e efetuamos alguns pushes para lá.

Nas próximas seções, vamos avançar com nossos estudos instalando novos módulos no NodeJS, ganhando assim novos comandos na linha de comando.

Pensando fora do Browser

O NodeJS é um programa escrito em C++ e baseado na engine V8 cuja principal função é poder ler e interpretar códigos javascripts fora do browser.

A princípio, o NodeJS foi pensado para poder criar servidores web baseados em eventos javascripts. E isso ainda é perfeitamente possível. Mas o fato é que você pode instalar e utilizar o node sem nunca necessariamente criar ou rodar um servidor baseado nele. Um exemplo: há um módulo do nodejs chamado uglify. (Um módulo é um arquivo javascript que você instala através do npm. Veremos isso detalhadamente em instantes). Uma vez instalado o uglify, você ganha de presente um novo comando no seu terminal. Esse comando faz com que o NodeJS seja capaz de ler um arquivo js e minificá-lo, automaticamente. Lindo né?

Uma coisa que devemos ter em mente é que um módulo do node pode ser instalado globalmente (no sistema, podendo ser usado em qualquer projeto) ou localmente (apenas no projeto atual).

Instalando módulos no NodeJS

Assim que instalamos o node já temos o nosso primeiro módulo disponível para uso, que é o próprio node package management, ou seja, ganhamos um comando novo no terminal:

`npm`. Há uma flag que mostra um texto de ajuda sobre determinado comando: `-h`. Então, digite no seu terminal: `$ npm install -h`

Esse comando irá mostrar o texto de ajuda do comando install.

Sempre que você tiver dúvida sobre algum comando, habitue-se a sempre ver o texto de ajuda antes de procurar no stackoverflow. É uma excelente maneira de evoluir seus conhecimentos sobre determinada ferramenta.

Voltando: repare que no começo da ajuda há a seguinte linha: `npm install <pkg>`, onde `<pkg>` é o pacote a ser instalado.

Queremos instalar em nosso sistema o uglify, portanto, digite no terminal: `$ npm install uglify-js -g`

a flag `-g` informa que você está instalando o pacote globalmente no seu sistema.

Você deve estar se perguntando de onde o npm baixou os arquivos necessários para a instalação. A resposta: <https://www.npmjs.com/package/uglify-js>. O site [npmjs.com](https://www.npmjs.com) é um repositório de pacotes desenvolvidos pela comunidade prontos para uso. Se o pacote estiver neste repositório, basta digitar seu nome logo após o comando `install` que o npm já sabe o que fazer.

Se a instalação ocorreu com sucesso, você deverá ver no terminal o local onde o módulo foi instalado (lembre-se que o instalamos globalmente no sistema com a flag `-g`) e as dependências necessárias para este módulo funcionar.

Agora que temos um comando novo na linha de comando, vamos experimentar o seguinte comando: `$ uglifyjs -h`

Eu confesso que a ajuda do uglify é um pouco extensa, mas não se preocupe, por hora, vamos simplesmente ver o uglify em ação.

A ideia é minificarmos o nosso arquivo `app.js`. Mas para isso, vamos editá-lo um pouco, como a seguir:

```
var body = document.body;
var cor = 'red';

var mudaCor = function(alvo, cor){
  alvo.style.color = cor;
}

mudaCor(body, cor);
```

Tudo deve continuar funcionando, entretanto, vamos minificar esse arquivo. Para isso, tenha certeza de que está na pasta `PrimeirosPassosWorkflow` e digite no terminal:

```
$ uglifyjs js/app.js --output js/app.min.js
```

Se tudo correu bem, um novo arquivo foi criado. O NodeJS leu o seu arquivo `app.js` e deu saída (`--output`) em outro arquivo, mas desta vez minificado. Abra-o no seu editor e veja o que aconteceu. Podemos ver que a minificação não foi lá essas coisas. Vamos corrigir isso. Digite no seu terminal:

```
$ uglifyjs js/app.js --output js/app.min.js --mangle
```

A flag `--mangle` irá analisar o seu arquivo e irá substituir, onde possível, alguns nomes de variáveis para deixar o arquivo mais enxuto, reduzindo assim alguns kbytes de transferência de dados.

Se você olhar o arquivo minificado, verá que os nomes dos parâmetros da função `mudaCor` foram alterados para um caracter apenas. Bem melhor, não? Há outras opções disponíveis no uglify. E há também uma infinidade de outros pacotes disponíveis no npmjs.org. Mas o mais importante aqui é perceber que usamos o NodeJS e o npm para instalar um pacote globalmente e utilizamos esse pacote sem necessariamente termos configurado um servidor.

Precisamos entender agora a diferença entre instalar um pacote global (com a flag `-g`) e um pacote local em nosso projeto.

Maneiras de instalar pacotes do npm

Sabemos que para instalar um novo pacote através do npm, devemos digitar em nosso terminal o seguinte comando:

```
$ npm install <package> , para instalações locais em cada diretório do projeto
```

OU:

```
$ npm install <package> -g , para instalações globais no sistema.
```

Contudo, falta-nos entender, de fato, quais as principais diferenças entre essas duas abordagens:

A vantagem mais óbvia de uma instalação global é com relação à utilização do espaço físico na sua HD. Os pacotes instalados globalmente precisam ser instalados apenas uma vez e podem ser utilizados em quantos projetos forem necessários. Se você usa com frequência um determinado módulo do NodeJS, vale a pena pensar em instalá-lo globalmente. A propósito, o caminho físico dessa instalação em seu sistema Windows é: `C:\Users\\AppData\Roaming\npm\node_modules`

Se você instalar um módulo localmente na sua pasta do projeto os arquivos e todas as suas dependências ficarão dentro de um diretório chamado "node_modules" pertencente ao projeto. Os módulos do npm são conhecidos por serem, algumas vezes, um tanto pesados. Portanto, o espaço físico disponível na sua HD deve ser levado em consideração.

Dito isto, você deve estar se perguntando por qual motivo alguém deveria instalar um módulo localmente no diretório do projeto. Eu creio que este não seja o momento de explicar isso, mas eu quero que você tome nota do parágrafo abaixo e leia-o no futuro (eu anotaria no evernote):

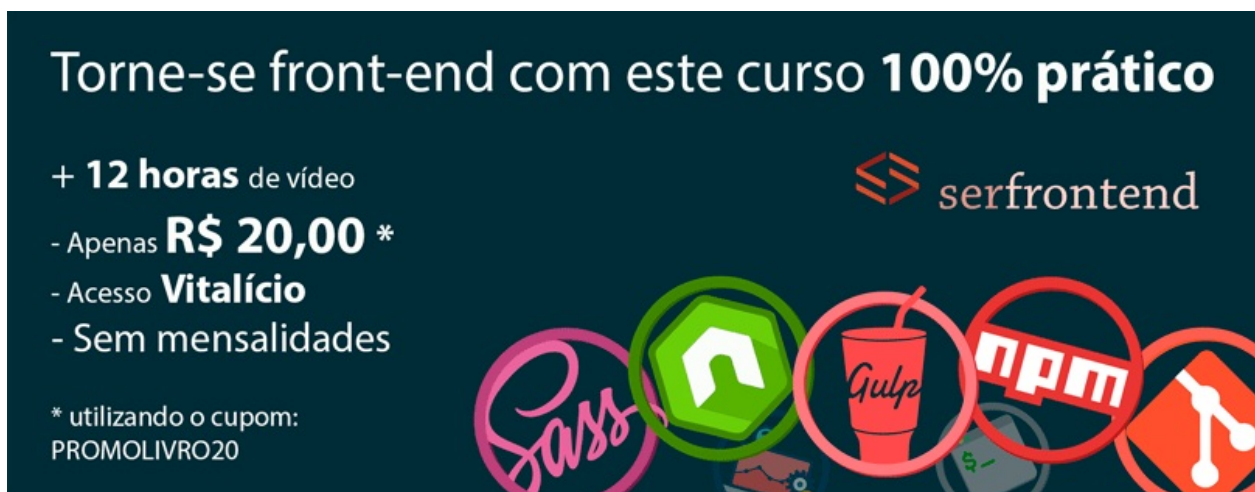
Se você quiser usar o módulo no seu projeto, através do terminal (linha de comando, ou CLI, como preferir), você poderá instalar um módulo globalmente. Entretanto, se você estiver desenvolvendo um módulo para servir de dependência de outro módulo - usando `require("seu_modulo")` -, você terá que instalá-lo localmente, na raiz do seu projeto.

Na prática, eu instalo os módulos que utilizo com mais frequência globalmente. Contudo, instalo os módulos usados em projetos específicos apenas localmente. Estes pacotes serão descartados da minha máquina quando o projeto estiver entregue em produção, liberando espaço físico da minha HD. Não há problema algum em apagar um módulo, desde que você consiga nova instalação rapidamente, sem esforço algum, caso necessário na manutenção do projeto. E isso é tema do próximo tópico.

Package.json

Ora, há uma série de módulos prontos para serem utilizados em <https://www.npmjs.com/>. Tem módulo para tudo que você imaginar. Mas você não pode sair instalando tudo globalmente no seu sistema como se não houvesse o amanhã. Você irá primeiramente trabalhar com esse módulo localmente, e além disso, como dito no tópico anterior, você pode querer trabalhar com este módulo apenas num projeto específico e depois excluí-lo da sua máquina. Ou seja, para que você não tenha problemas em manutenções futuras, você deverá anotar os módulos utilizados em algum lugar, para que no futuro você mesmo ou qualquer outra pessoa possa instalar corretamente todos estes módulos e continuar trabalhando no projeto. Talvez você tenha pensado num LEIAME.txt, mas não, não faça isso. Uma melhor maneira de resolver esse problema é criar um arquivo chamado package.json contendo, entre outras coisas, os módulos necessários para que o projeto continue funcionando. Além de informar o nome do módulo você também pode incluir informação sobre a versão utilizada, o que pode ser muito útil no futuro.

Os projetos baseados em NodeJS usam um arquivo de configuração (manifest) chamado package.json



Torne-se front-end com este curso 100% prático

+ **12 horas** de vídeo

- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

serfrontend

Sass, Home, Gulp, NPM, Git

Um outro cenário é quando trabalhamos em equipe. Se eu estou desenvolvendo um projeto node, posso versionar o arquivo de configuração package.json, ignorando toda a pasta node_modules (que, como vimos, são os módulos instalados localmente). Se uma pessoa do meu time precisar desenvolver uma funcionalidade nova ou continuar o meu trabalho,

ele deverá puxar do Git/GitHub (`$ pull origin` , lembra?) somente o package.json, não se importando com os módulos que instalei localmente. Pois estes módulos serão instalados automaticamente na sua máquina com apenas um comando na linha de comando:

```
$ npm install
```

Esse comando irá ler o arquivo package.json e instalar todas as dependências locais do projeto. Isso não é lindo?

Esse arquivo possui a seguinte aparência:

```
{
  "name": "PrimeirosPassos",
  "version": "0.0.1",
  "description": "Uma descrição do seu projeto",
  "main": "main.js",
  "repository": {
    "type": "git",
    "url": "http://github.com/tapmorales/meu_repositorio"
  },
  "keywords": [
    "array", "palavras chaves"
  ],
  "author" : {
    "name" : "Daniel Tapias Morales",
    "email" : "daniel@email",
    "url" : "http://www.meudominio.com/"
  },
  "homepage": "http://www.meu-projeto.org"
  "license": "ISC",
  "private" : true,
  "dependencies" : {
    "nome_do_modulo" : "x.x.x"
  }
}
```

Acredito que a maior parte desse arquivo seja autoexplicativo, portanto, não entrarei em maiores detalhes.

O que merece maior atenção é com relação à última propriedade chamada "dependencies", que é um objeto cuja chave é o nome do módulo dependência e o valor é a versão utilizada em seu projeto. Ao digitar em seu terminal `$ npm install` esse objeto será lido pelo npm e todos os módulos listados serão instalados automaticamente. Eu já falei que isso é lindo?

Mas aqui eu preciso chamar a sua atenção para um detalhe importante. Os projetos instalados globalmente não são listados como dependência. Essa é uma grande desvantagem da instalação global. Dito isso, vamos criar o nosso arquivo package.json.

Mas como criar esse arquivo? É na unha?

Não precisa ser. Há um comando do npm que te ajuda a criar esse arquivo por meio de um assistente maroto. Tudo o que você precisa fazer é responder algumas perguntas e o arquivo de configuração será criado. Para abrir este assistente, digite no seu terminal:

```
$ npm init
```

Feito isso, a primeira informação que você precisará fornecer é o nome do seu projeto. Vá digitando e confirmando com o ENTER. Quando você não tiver certeza, pode apertar o ENTER mesmo sem preencher nada. Há qualquer momento, CTRL + C para sair do assistente.

No final, você verá como o seu arquivo json será criado. Aperte Enter para confirmar.

A partir de agora, sempre que você quiser instalar um novo módulo para utilizar em seu projeto e quiser mencioná-lo como dependência no arquivo package.json, basta você passar uma flag no momento da instalação: `--save`.

A flag `--save` acrescenta uma nova configuração no package.json, informando qual o módulo e versão está sendo utilizado no projeto.

Mas antes de entrar em maiores detalhes, quero explicar para você que há dois tipos de dependências: as de produção e as de desenvolvimento.

Mas isso é assunto para a próxima seção.

Dependências de produção vs. de desenvolvimento

Há dois tipos de dependências em projetos NodeJS. Vamos estudá-las.

Dependências de produção

São aquelas dependências que precisam estar em produção para que seu projeto funcione. Por exemplo: jquery, angular, bootstrap etc. Se você deseja instalar um módulo de produção, basta digitar em seu terminal:

```
$ npm install <pacote> --save
```

Espero que você entenda porque não faz sentido digitar o seguinte:

```
$ npm install <pacote> -g --save
```

Dependências de desenvolvimento

São as dependências que não precisam ser levadas para produção. Estes módulos são usados apenas em ambiente de desenvolvimento. Alguns exemplos são os módulos que você instala para minificar, concatenar arquivos e até mesmo rodar alguns pré-processadores de css.

Para incluir no package.json uma dependência de desenvolvimento, basta digitar a flag --save-dev, como no exemplo:

```
$ npm install <pacote> --save-dev
```

Três informações importantes:

-
1. Na prática, os comandos acima criam duas novas propriedades no arquivo package.json, cada uma contendo um objeto, são elas: dependencies e devDependencies, usadas para listar as dependências de produção e de desenvolvimento, respectivamente.
 2. Alguns devs gostam de diferenciar o gerenciador de pacotes de produção e desenvolvimento. Eles utilizam um gerenciador de dependências exclusivo para o ambiente de produção chamado bower. Eu acho isso uma boa prática. Mas como não

vou incluir nenhum arquivo voltado para o front-end, não vou me preocupar com as dependências de produção. Contudo, tenha em mente que utilizar o bower em projetos na vida real pode ser uma boa estratégia.

3. Se você instalou todas as dependências de seu projeto usando a flag `--save`, estas estão listadas no seu arquivo `package.json`. Isso significa que você não precisa versionar seu diretório `node_modules`, o que é uma boa prática. Se você ou outra pessoa precisar rodar seu projeto em outra máquina, basta digitar no terminal `$ npm install` que todas as dependências serão instaladas de uma única vez. É possível dizer ao Git que você não quer versionar seu diretório de módulos criando um arquivo `.gitignore` contendo simplesmente o nome da pasta `"node_modules/"`. Lembre-se de incluir esse arquivo `.gitignore` no versionamento com o `$ git add .gitignore` OU `$ git add .`

Pois bem. Até o momento aprendemos a diferenciar os pacotes de desenvolvimento/produção e instalações globais/locais. A sua lição de casa é dar uma olhada no [bower](#). Você verá que, com o conteúdo que aprendeu até aqui, será capaz de instalar e utilizar essa ferramenta para gerenciar suas dependências de produção.

Atenção: Spoiler à frente

Se você chegou até aqui e deu uma olhada no [bower](#), você deve ter percebido que ao instalar uma dependência de produção a mesma fica armazenada numa pasta chamada `bower_components`.

Como bower é um gerenciador de pacotes específicos para front-end, você encontrará frameworks para ser entregues em produção, como por exemplo:

- Bootstrap
- jQuery
- jQuery UI
- Underscore
- AngularJS
- Backbone
- Modernizr
- Normalize.css
- Font Awesome

Estes são só alguns exemplos. Você pode ver os pacotes disponíveis na ferramenta [clcando aqui](#).

De certa maneira, esse processo agiliza muito o processo de iniciar um projeto novo. Você não precisa mais baixar todas as bibliotecas que precisará num diretório no seu projeto e depois vinculá-los um a um.

Você pode listar todas as dependências do front-end num arquivo chamado `bower.json` e depois baixá-las todas de uma única vez usando o comando `$ bower install` .

Fim do Spoiler

Automatização de tarefas

Se você chegou até aqui, meus parabéns! Até esse ponto, você já deve ter conhecimento para:

- Criar um repositório no Git e no GitHub contendo os arquivos de seu projeto versionados.
- Utilizar o terminal e o npm para instalar pacotes do NodeJS para facilitar o desenvolvimento de seu site/aplicação.
- Usar o npm para criar o package.json e listar, nesse mesmo arquivo, as dependências do seu projeto no momento da instalação.

Nós vimos o básico do NodeJS e do npm instalando o pacote [uglify-js](#). Quando fizemos isso, ganhamos de presente um comando novo de linha de comando que podemos usar para minificar nossos arquivos javascript. Por favor, nunca mais entregue em produção seus arquivos javascripts de desenvolvimento.

Além do uglify, há uma série de outros pacotes super úteis no desenvolvimento front-end. Listarei alguns:

- [Autoprefixer](#). Com este módulo instalado você escreve seus arquivos css sem se preocupar com os prefixos dos vendors (-webkit, -moz, -o, -ms). Ou seja, você escreve seu CSS normalmente e o autoprefixer escreve os prefixos pra você.
- [Clean CSS](#). É um minificador para arquivos css.
- [Usemín](#). Procura no arquivo html as chamadas para arquivos externos (javascript e css) e as substitui por uma única chamada a um arquivo concatenado.
- [Imagemin](#). Responsável por minificar suas imagens.

Há uma série de outros módulos úteis disponíveis no <https://www.npmjs.com>. Mas o fato é que, mesmo sendo uma mão na roda, executar estes comandos, um a um no terminal, não me parece uma boa prática. Imagine que toda vez que você efetuar uma alteração no seu arquivo fonte (arquivo de edição), você tenha que executar uma série de comandos no terminal. Um comando para concatenar os seus javascripts num arquivo só. Outro comando para minificá-lo. Outro ainda para minificar as suas imagens e outro para substituir as chamadas aos seus arquivos externos no seu html. Não seria produtivo e esse workflow seria abandonado muito rapidamente. E é aí que entram os automatizadores de tarefas.

Há dois principais automatizadores de tarefas, o [Grunt](#) e o [Gulp](#). Vou focar somente no primeiro pois, quando eu comecei meus estudos, tinha muito mais plugins disponíveis para a instalação.

O Grunt funciona da seguinte forma: você escreve uma lista de tarefas e atribui à essa lista um nome, por exemplo, 'deploy'. Toda vez que você chamar no terminal esse 'deploy', uma série de comandos será executada, ou seja, você poderá, de uma única vez, executar as seguintes tarefas:

1. Autoprefixar os seus arquivos CSS.
2. Minificar o seu arquivo CSS autoprefixado.
3. Concatenar todos os seus arquivos javascripts num arquivo só.
4. Minificar esse arquivo, removendo quebras de linha, reduzindo nomes de variáveis e removendo os comentários.
5. Copiar alguns arquivos estáticos (como arquivos de vendors e imagens) para a sua pasta de deploy (que não é a mesma pasta de desenvolvimento).
6. Atualizar as chamadas nos arquivos htmls aos arquivos externos.

Existem outras possibilidades ao trabalharmos com o Grunt, mas por hora, vamos focar no workflow descrito acima.

Te vejo no próximo tópico

Criando a estrutura de diretórios

A partir desse momento vamos trabalhar com uma estrutura de diretórios um pouco diferente de como estávamos trabalhando até o tópico anterior. A ideia é termos uma estrutura que lembre um pouco um projeto real. Para isso, vou pedir que você execute algumas tarefas:

- Crie um novo diretório chamado "projetoWorkflow" no seu diretório home. (fora da pasta que estávamos trabalhando, a "PrimeirosPassosWorkflow").
- Nesse diretório recém-criado, acrescente mais dois diretórios: "source" e "deploy". Os arquivos de "source" serão os arquivos fontes (arquivos de edição). Os arquivos que estiverem em "deploy" serão os arquivos que irão para produção. Estes arquivos serão gerados pelo nosso GruntJS.
- Dentro da pasta "source" crie mais três diretórios, a saber: "javascript", "sass" e "vendor". Crie também um index.html com a estrutura inicial de qualquer arquivo html.
- Na raiz do projeto, crie um arquivo chamado .gitignore (mantenha o 'ponto' inicial no nome do arquivo).
- Crie um arquivo package.json através de um comando no npm. Lembra qual é o comando? `$ npm init`. Responda as perguntas feitas pelo assistente e vá dando ENTER.
- Transforme o diretório raiz (PrimeirosPassosWorkflow) num repositório Git. Lembra-se do comando que faz isso? Não poderei contar que é o `$ git init` ;)
- Crie um novo repositório na sua conta do GitHub e mande tudo para lá. Se você não se lembra como faz isso, é uma boa hora para revisar [Fundamentos do GitHub](#). Essa etapa não é obrigatória, mas é um bom momento para revisarmos.

No final das etapas descritas acima, você terá a seguinte estrutura:

```
projetoWorkflow
|-> .git
|-> deploy
|-> source
    |-> javascript
    |-> vendor
    |-> sass
    |- index.html
|- .gitignore
|- package.json
```

Faça uma pequena edição no seu arquivo package.json para ficar semelhante à:

```
{
  "name": "projetoWorkflow",
  "version": "1.0.0",
  "description": "Exemplo de um projeto",
  "author": "Daniel Tapias Morales"
}
```



Explicação sobre o processo de build

O build, ou deploy, nada mais é do que preparar os arquivos para serem enviados para produção (subir no servidor e torná-lo disponível na internet). Esse processo pode envolver uma série de tarefas, e pode variar de projeto para projeto. Os exemplos mais comuns e simples de entender são:

- aut prefixar nos arquivos css, ou seja, incluir os prefixos dos browsers, quando necessário.
- concatenar arquivos javascript.
- minificar arquivos, aumentando a performance do site/aplicação.
- Copiar arquivos estáticos (como mídias ou fontes) no diretório de deploy.

Há muitos outros processos legais que poderíamos ver nesse guia, mas focarei no mais fundamental para seu entendimento.

Dito isto, quero detalhar alguns pontos sobre a estrutura de diretórios criada acima:

- O arquivo package.json, como sabemos, é onde listamos as nossas dependências do projeto.
- O arquivo .gitignore é usado para dizer ao Git quais arquivos não devem ser versionados.

- Não deveremos mexer dentro da pasta "deploy". Os arquivos desta pasta serão criados pelo nosso *builder*, no nosso caso, será o GruntJS.
- A pasta source/javascript terá mais de um arquivo .js. Porém, esses arquivos serão concatenados num único arquivo e minificados no momento do build.
- Por enquanto, a pasta source/sass conterà um arquivo css simples. Este arquivo será autoprefixado apenas. Nos próximos tópicos faremos as alterações necessárias para podermos trabalhar com o pre-processador SASS.
- A pasta source/vendor conterà, para exemplificar, um arquivo de terceiro, o modernizr. Se este arquivo estiver minificado (normalmente está) ele simplesmente será copiado para o devido lugar dentro de "deploy". Esse mesmo processo poderia ser criado para uma pasta "images", por exemplo. Não criarei imagens em nosso projeto para manter as coisas simples.
- Antes de cada build, iremos apagar (automaticamente, claro) todos os arquivos que estiverem dentro de "deploy". Esse processo é importante para evitarmos "sujeiras" de builds anteriores.

Já temos o entendimento necessário para instalarmos o GruntJS, mas isso é assunto para o próximo tópico.

Instalando o Grunt JS

Para que o Grunt funcione adequadamente, precisamos primeiramente instalar a sua interface CLI (*command line interface*). Dessa forma, você consegue rodar os comandos do grunt via terminal. Para isso, digite o comando abaixo

```
$ npm install -g grunt-cli
```

Em seguida, precisamos instalar o Grunt e incluí-lo na nossa lista de dependência de desenvolvimento no arquivo package.json. Para isso, tenha certeza que você está visualizando o diretório projetoWorkflow no seu terminal e digite:

```
$ npm install grunt --save-dev
```

Feito isso, deverá acontecer duas coisas: A primeira é que foi criado um diretório chamado node_modules contendo o grunt. A segunda é que foi acrescentado no seu package.json as seguintes linhas:

```
"devDependencies": {
  "grunt": "^0.4.5"
}
```

Instalando plugins do Grunt

O Grunt sozinho não faz nada sem os plugins (mentira, faz sim, mas por enquanto, aceite isso como uma verdade). Ou seja, precisamos instalar os plugins para rodar determinadas tarefas que você precisar. Há plugin pronto para quase tudo, e você poderá checar o site <http://gruntjs.com/plugins> para pesquisar coisas novas.

Num dos tópicos passado, instalamos o uglify para minificar nosso js, lembra? Apenas para recordar, segue o comando para instalar o uglify:

```
$ npm install uglify-js
```

Contudo, ao trabalharmos com o Grunt, não precisamos instalar o pacote acima para minificar nossos arquivos. Em contrapartida, precisamos instalar um ‘wrapper’ do próprio Grunt que envolve o pacote uglify e assim possibilita que executemos o comando uglify não mais pela linha de comando, mas sim programaticamente. É isso mesmo. Nós iremos

escrever códigos javascript que rode comandos que antes só rodavam no terminal. Ou seja, vamos criar um arquivo javascript que contenha, por exemplo, uma chamada para “uglify”. Mas eu não estou falando de qualquer arquivo. Eu estou falando do Gruntfile.js.

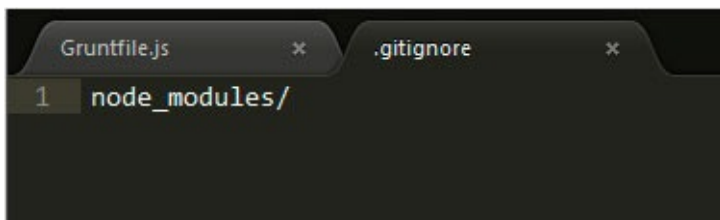
arquivo Gruntfile.js

Temos quase tudo de que precisamos. Resta agora criarmos, na raiz do nosso projeto, um arquivo chamado Gruntfile.js (mantenha a primeira letra maiúscula). Esse arquivo conterá as definições das tarefas que você criar. É a partir desse arquivo que o Grunt saberá quais tarefas executar.

Mas antes de continuarmos, eu preciso que você instale todos os plugins necessários para executar as tarefas mencionadas anteriormente. Para isso, você irá digitar no terminal as seguintes linhas:

```
$ npm install grunt-autoprefixer --save-dev
$ npm install grunt-contrib-cssmin --save-dev
$ npm install grunt-contrib-uglify --save-dev
$ npm install grunt-contrib-copy --save-dev
$ npm install grunt-contrib-clean --save-dev
```

Da mesma forma que os pacotes instalados diretamente pelo terminal, os plugins do Grunt são copiados para dentro de node_modules. Dependendo do numero de plugins instalados, o diretório node_modules fica com um peso considerável. Por isso, acho uma boa prática não incluí-lo no versionamento. Lembre-se que o .gitignore está aí para isso.



Até aqui já temos o GruntJS e também alguns plugins instalados em nosso projeto. Vamos aprender no próximo tópico como configurar no Gruntfile.js.

Registrando tarefas no Gruntfile.js

Depois que tivermos todos os plugins instalados, precisamos registrá-los (carregá-los) no nosso Gruntfile. Para isso, abra esse arquivo no seu editor de código e insira o trecho a seguir:

```
module.exports = function(grunt) {  
  
    grunt.loadNpmTasks('grunt-autoprefixer');  
    grunt.loadNpmTasks('grunt-contrib-cssmin');  
    grunt.loadNpmTasks('grunt-contrib-uglify');  
    grunt.loadNpmTasks('grunt-contrib-copy');  
    grunt.loadNpmTasks('grunt-contrib-clean');  
}
```

Isso faz com que os plugins instalados anteriormente estejam disponíveis para serem usados via Gruntfile.js, e não mais via terminal.

O próximo passo é configurar cada um dos plugins. Cada um deles possui sua maneira peculiar de configuração. Eu farei aqui somente o básico, mas no final, você poderá ler diretamente a documentação de cada um dos plugins para se aprofundar.

Toda a configuração precisa estar inserida num método do Grunt. Veja o código:

```
grunt.initConfig({  
  
});
```

Esse objeto que é passado por parâmetro para a função `initConfig` conterá as opções de cada um dos plugins.

```
grunt.initConfig({
  autoprefixer: {

  },

  copy: {

  },

  clean: {

  },

  cssmin: {

  },

  uglify: {

  },

});
```

Por hora, essa é a aparência de nosso arquivo.


```
Gruntfile.js
1  module.exports = function(grunt) {
2      'use strict';
3
4      grunt.initConfig({
5          autoprefixer: {
6
7          },
8
9          copy: {
10
11          },
12
13          clean: {
14
15          },
16
17          cssmin: {
18
19          },
20
21          uglify: {
22
23          },
24      });
25
26
27
28      grunt.loadNpmTasks('grunt-autoprefixer');
29      grunt.loadNpmTasks('grunt-contrib-cssmin');
30      grunt.loadNpmTasks('grunt-contrib-uglify');
31      grunt.loadNpmTasks('grunt-contrib-copy');
32      grunt.loadNpmTasks('grunt-contrib-clean');
33
34  }
```

Para começarmos os nossos testes efetivamente, precisamos criar os nossos arquivos css, javascript e html. Mas para isso, você terá que clicar no link do próximo tópico (ou ir para a próxima página, se tiver feito o download deste gruaia).

Criando os arquivos estáticos

Esse tópico não apresentará nada de novo. Vamos criar os arquivos necessários para rodar as tarefas do Grunt. Então, simplesmente crie os seus arquivos no diretório 'source'. Não se esqueça que precisaremos de um arquivo modernizr.

Eu quis criar dois arquivos javascripts pra vermos como a concatenação deles funciona. Eu até que me esforcei para pensar em alguma funcionalidade que fosse simples, mas que não fosse banal. Depois de alguns minutos, desisti. Resolvi criar duas funcionalidades ~~idiotas~~ banais mesmo. Para os propósitos deste guia, satisfaz.

Os arquivos javascripts estão divididos em módulos. Um mostra a data de hoje na tela (UAU!) e o outro mostra na tela quantas vezes o usuário clicou num botão (WOOWW).

O CSS foi criado com alguns detalhes de gradiente para vermos o autoprefixer em ação.

O meu HTML e CSS segue a metodologia BEM (Block, element, modifier) para estruturar minhas classes. Existem várias outras metodologias já criadas e discutidas. E você pode usar alguma delas, alterá-la ao seu modo ou simplesmente criar a sua própria metodologia. A ideia é criar um padrão, uma maneira de escrever códigos CSS que sejam reutilizáveis, escaláveis e elegantes. Não entrarei em detalhes sobre o BEM, mas garanto que é bem simples e que há vários materiais na internet para você consultar.

Se você é um profundo conhecedor de BEM, você perceberá uma pequena falha nos meus CSS's. Não se preocupe com isso pois no momento oportuno eu retomarei esse assunto para corrigirmos isso. Saiba que esse "erro" é proposital e tem finalidades didáticas

Além disso, há mais uma coisa que preciso dizer: se você fizer o seu CSS exatamente igual ao meu, você verá que eu não me preocupei com o design da página. (Mentira. A verdade é que não sou um bom designer, e por isso sempre escolho tons de cinza. É mais difícil de errar).

Agora sim, segue a estrutura final e os códigos abaixo

index.html

```
<!doctype html>
<html lang="pt-br">
  <head>
    <meta charset="utf-8">
    <title>Exemplo de Task Runner</title>
    <script src="vendor/modernizr.min.js"></script>
    <link rel="stylesheet" href="sass/main.css">
  </head>
  <body>
    <header class="mainHeader">
      <h1 class="mainHeader__title">Exemplo de utilização do GruntJS</h1>
    </header>

    <main role="main">

      <div class="dateContainer">
        <p class="dateContainer__content" id="containerDate"></p>
      </div>

      <div class="buttonsContainer">
        <button id="incrementButton" class="buttonsContainer__button">Clica pa
ra incrementar</button>
        <div id="resultIncrementButton" class="buttonsContainer__result">Clica
do <span class="buttonsContainer__result--number js-result">0</span> vezes</div>
      </div>

    </main>

    <script src="javascript/date.js"></script>
    <script src="javascript/incrementButton.js"></script>
  </body>
</html>
```

sass/main.css

```
body{
  background: #ccc;
  font-family: arial, sans-serif;
}

.mainHeader{
  color: #bbb;
  text-shadow: 2px 2px 1px rgba(0,0,0,.6);
}

.buttonsContainer{
  border-top: 2px solid #A5A5A5;
}

.buttonsContainer .buttonsContainer__button{
  background-image: linear-gradient(to top, #848484, #AFAFAF);
  border: none;
  padding: 1em;
  margin: 1em 0;
  cursor: pointer;
  outline: none;
}

.buttonsContainer .buttonsContainer__button:hover{
  background-image: linear-gradient(to top, #AFAFAF, #848484);
}

.buttonsContainer .buttonsContainer__button:active{
  background-image: linear-gradient(to top, #868686, #565656);
}

.buttonsContainer .buttonsContainer__result--number{
  font-weight: bold;
}
```

javascript/date.css

```
/*
Mostra a data na tela
*/

;(function(){
    'use strict';
    var dateContainer = document.getElementById('containerDate');

    /* recupera a data e retorna na variavel dateInfo */
    var dateInfo = (function(){
        var today    = new Date()
        ,    mounth   = parseInt(today.getMonth() + 1).length < 2 ? parseInt(today.getMo
nth() + 1) : '0' + parseInt(today.getMonth() + 1)
        ,    date     = today.getDate() + '/' + mounth + '/' + today.getFullYear();

        console.log('a data que irá retornar é: ' + date);

        return date;
    })();

    // mostra a data na tela

    dateContainer.appendChild(document.createTextNode(dateInfo));
})();

,
```

javascript/incrementButton.css

```
/*
Conta quantos cliques no botão
*/

;(function(){
  'use strict';
  var botao      = document.getElementById('incrementButton')
  ,   container  = document.querySelector('.buttonsContainer__result .js-result')
  ,   count      = 0;

  var showQuant  = function(n){
    console.log('showQuant chamado recebendo: ' + n);
    // mostra a quantidade na tela
    container.firstChild.nodeValue = n;
  }

  showQuant(count);

  botao.addEventListener('click', function(e){
    showQuant(++count);
  })
})();
```

Segue o link para download do modernizr: <http://modernizr.com/download/?-eventlistener-queryselector-rgba-textshadow-setclasses>

Segue a estrutura de arquivos do meu projeto para sua referência.

```
▼ projetoWorkflow
  ► deploy
  ► node_modules
  ▼ source
    ▼ javascript
      date.js
      incrementButton.js
    ▼ sass
      main.css
    ▼ vendor
      modernizr.min.js
  index.html
  .gitignore
  Gruntfile.js
  package.json
```

No próximo tópico, vamos iniciar a configuração de nosso gruntfile para executar as tarefas que havíamos planejado.

Te vejo no futuro!

Executando Tarefas

Autoprefixer

A nossa primeira tarefa será incluir, automaticamente, os vendors prefixers no nosso arquivo css. Vale saber que este plugin utiliza as informações disponíveis no [caniuse](#) para decidir qual vendor incluir no arquivo css. Para isso, o autoprefixer se baseia sempre nas duas últimas versões de cada browser (mas isso é configurável).

Abra o seu Gruntfile no editor de código e substitua:

```
autoprefixer: {  
  
},
```

por:

```
autoprefixer: {  
  dist: {  
    files: {  
      'deploy/css/main.css': 'source/sass/main.css',  
    },  
  },  
},
```

A configuração desse arquivo é tão simples que acredito ser auto-explicativa.

Você pode verificar outras opções de configurações na própria página do plugin.

<https://www.npmjs.com/package/grunt-autoprefixer>

Você pode testar se o autoprefixer está funcionando digitando no terminal:

```
$ grunt autoprefixer
```

Você deverá ver uma mensagem "Done, without errors", indicando que correu tudo bem.

Abra o seu css gerado na pasta de deploy e veja os prefixos do linear gradiente. Veja que eu os escrevi como manda a especificação (to top) e o autoprefixer incluiu o gradiente como fazíamos antes da especificação (bottom).

Repare também que nada mudou com o text-shadow, isso porque o autoprefixer não cria fallbacks. Ele simplesmente adiciona os prefixos, se estes existirem.

Minificando o CSS

Depois que o CSS foi gerado, nós podemos reduzir um pouco o tamanho desse arquivo, minificando-o. Na prática, nesse exemplo simples o resultado será imperceptível (alguns bytes), mas se você estiver trabalhando num projeto maior, mesmo que consiga reduzir alguns kbytes valerá a pena devido à simplicidade desse processo.

Para esta tarefa, usaremos o plugin [cssmin](#).

Substitua no lugar adequado no seu Gruntfile

```
cssmin: {
  dist: {
    files: {
      'deploy/css/main.css': 'deploy/css/main.css'
    }
  }
},
```

Repare que nós estamos minificando o arquivo que foi prefixado anteriormente, já na pasta deploy. OK!

Rode no terminal `$ grunt cssmin` e veja o que acontece

Minificando e concatenando seus javascripts

A nossa próxima tarefa é minificarmos e concatenarmos nossos arquivos javascript. Para isso, vamos usar o [uglify](#).

Da mesma forma que fizemos anteriormente, altere o seu Gruntfile, incluindo no lugar adequado o seguinte código:

```
uglify: {
  options: {
    mangle: true
  },

  dist: {
    files: {
      'deploy/javascript/app.min.js': [
        'source/javascript/incrementButton.js',
        'source/javascript/date.js'
      ]
    }
  },
}
```

Moleza. A propriedade do objeto files é o arquivo que será gerado. O valor dessa propriedade é um array de arquivos, ou seja, o uglify irá concatenar cada um dos arquivos descritos na array num único arquivo.

O opção true para o mangle faz com que a minificação seja mais pesada, alterando, quando possível, o nome de algumas variáveis para reduzir ainda mais o tamanho do arquivo.

Para ver o uglify em ação, você pode digitar no terminal:

```
$ grunt uglify
```

Sempre abra o arquivo gerado no seu editor de código para ver o resultado.

Um detalhe importante: boa prática não minificar arquivos já minificados, como bibliotecas de terceiros.

Copiando arquivos estáticos

Uma tarefa bastante trivial é ter que copiar alguns arquivos estáticos para a pasta de deploy, como por exemplo: arquivos de imagens, arquivos de fontes, javascripts de terceiros, etc. Para essa tarefa, não usaremos o CTRL+C e CTRL+V, mas sim o plugin [copy](#).

Da mesma forma que os plugins anteriores, substitua no lugar apropriado de seu Gruntfile.js

```
copy: {
  dist: {
    files: [
      {src: 'source/vendor/*', dest: 'deploy/vendor/'}
    ]
  }
},
```

A diferença é que agora a propriedade files não é mais uma string, mas sim um array de objetos.

Na verdade, o exemplo anterior funcionaria sem precisarmos de um array, ou seja, se fizéssemos dessa forma:

```
copy: {
  dist: {
    src: 'source/vendor/*',
    dest: 'deploy/'
  },
},
```

Mas a questão é que dificilmente temos que copiar somente um diretório, ou seja, escrevendo com array você poderá copiar mais de um diretório sem grandes problemas.

Por exemplo:

```
copy: {
  dist: {
    files: [
      {src: 'source/vendor/*', dest: 'deploy/vendor/'},
      {src: 'source/images/*', dest: 'deploy/images/'},
      {src: 'source/fonts/*', dest: 'deploy/fonts/'},
    ]
  }
}
```

É hora de verificarmos se o que fizemos deu certo. No terminal, digite `$ grunt copy` e em seguida abra o seu diretório deploy e veja como ficou. Os arquivos foram copiados? A estrutura de pasta está certa? Ficou como esperávamos? Eu acho que não.

A questão é que o Grunt copiou os arquivos, mas não da maneira como estávamos imaginando. Quando configuramos o Grunt dessa forma: `{src: 'source/vendor/*', dest: 'deploy/'}` o que esperávamos era que todos os arquivos inseridos na pasta vendor

fossem copiados para deploy/vendor. Mas na prática, o que o Grunt fez foi copiar o caminho completo, desde source, para dentro de deploy. O resultado foi o seguinte caminho: deploy\vendor\source\vendor. Bem estranho, não é mesmo?

Para resolver, precisamos dizer ao Grunt para, no momento da cópia, se basear relativamente ao diretório source, mas não incluí-lo na cópia. Para isso:

```
expand: true, //habilita o cwd
cwd: 'source/', //relativo ao diretório source, mas não o inclui na cópia
```

A configuração final de nosso copy ficou assim:

```
copy: {
  dist: {
    expand: true, //habilita o cwd
    cwd: 'source/', //relativo ao source, mas não o inclui na cópia
    src: 'vendor/*',
    dest: 'deploy/',
  }
},
```

Para finalizar, falta configurarmos a cópia da nossa index.html. Veja como deve ficar.

```
copy: {
  dist: {
    files: [
      {
        expand: true, //habilita o cwd
        cwd: 'source/', //relativo à source, mas não a inclui na cópia
        src: 'vendor/*',
        dest: 'deploy/'
      },
      {
        expand: true, //habilita o cwd
        cwd: 'source/',
        src: 'index.html',
        dest: 'deploy/'
      }
    ]
  }
},
```

Tudo copiado conforme esperávamos, mas ainda resta alguns problemas: O arquivo index.html faz as chamadas às folhas de estilo e ao javascript da seguinte forma:

```
<link rel="stylesheet" href="sass/main.css">

<script src="javascript/date.js"></script>
<script src="javascript/incrementButton.js"></script>
```

Precisamos atualizar a inclusão do javascript apontando para o arquivo concatenado e minificado e também arrumar o caminho do css, apontando para a pasta correta (nós trocamos o nome do diretório de "sass" para "css", e isso causou um problema de referência)

Eu já trabalhei com dois plugins do GruntJS para resolver esse tipo de problema:

- [grunt-useref](#).
- [grunt-usemin](#).

Para o nosso guia, eu vou simplesmente pedir que você altere a chamada do main.css e app.min.js em "source" e execute o copy novamente.

Se você quiser deixar mais profissional, tente por conta própria usar um dos plugins descritos acima. Senão, basta alterar o caminho do css e do javascript.

Limpando o diretório antes do deploy

A cereja do bolo é fazer com que, antes de cada deploy, o grunt apague todos os arquivos da pasta de destino. Isso garante que não sobre arquivos indevidos no nosso diretório de deploy, como por exemplo imagens ou fontes que não são mais utilizadas.

Para isso, vamos usar o [grunt-contrib-clean](#).

```
clean: {
  dist: {
    src: ["deploy"]
  }
}
```

Esse comando irá deletar o nosso diretório deploy. Não é difícil perceber que o comando `$ grunt clean` deve ser executado antes de qualquer outro comando.

Ou seja, no nosso terminal, devemos executar os seguintes comandos, nessa ordem:

```
$ grunt clean  
$ grunt autoprefixer  
$ grunt cssmin  
$ grunt uglify  
$ grunt copy
```

Eu acho que seria interessante se tivéssemos como executar apenas um comando no terminal e o próprio grunt executasse quantas tarefas fossem necessárias. E adivinha: isso é possível. É o que veremos a seguir.

Executando tarefas em lote

Estamos quase lá.

O que nos resta agora é fazer com que, digitando apenas um comando no nosso terminal, sejamos capazes de executar todas as tarefas de uma única vez. Para isso, nós iremos criar um alias, ou seja, um atalho para mais de uma tarefa criada no Gruntfile.

Veja um exemplo abaixo:

```
grunt.registerTask(minhaTarefa, [tarefa1, tarefa2, tarefa3])
```

Dessa forma, se você digitar no terminal

```
$ grunt minhaTarefa
```

As tarefas tarefa1, tarefa2 e tarefa3 serão executadas nessa mesma ordem.

Um detalhe importante é que você pode registrar uma tarefa chamada 'default'. Nesse caso, lá no terminal basta digitar `$ grunt` que esse "default" será chamado.

Para esse exercício, vamos criar uma tarefa chamada "deploy", que conterà todas as tarefas criadas até o momento:

```
grunt.registerTask("deploy", ["clean", "autoprefixer", "cssmin", "uglify", "copy"])
```

Agora basta digitarmos no terminal `$ grunt deploy` que as cinco tarefas serão executadas, uma a uma.

Isso não é demais!?

Para referência, segue meu Gruntfile.js final:

```
module.exports = function(grunt) {  
  'use strict';  
  
  grunt.initConfig({  
    autoprefixer: {  
      dist: {  
        files: {  
          'deploy/css/main.css': 'source/sass/main.css',  
        },  
      },  
    },  
  },  
};
```

```
    },

    copy: {
      dist: {
        files: [
          {
            expand: true, //habilita o cwd
            cwd: 'source/', //relativo à source, mas não a inclui na cópi
a
            src: 'vendor/*',
            dest: 'deploy/'
          },
          {
            expand: true, //habilita o cwd
            cwd: 'source/',
            src: 'index.html',
            dest: 'deploy/'
          }
        ]
      }
    },

    clean: {
      dist: {
        src: ["deploy"]
      }
    },

    cssmin: {
      dist: {
        files: {
          'deploy/css/main.css': 'deploy/css/main.css'
        }
      }
    },

    uglify: {
      options: {
        mangle: true
      },

      dist: {
        files: {
          'deploy/javascript/app.min.js': [
            'source/javascript/incrementButton.js',
            'source/javascript/date.js'
          ]
        }
      }
    }
  }

});
```



```
grunt.loadNpmTasks('grunt-autoprefixer');
grunt.loadNpmTasks('grunt-contrib-cssmin');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-copy');
grunt.loadNpmTasks('grunt-contrib-clean');

grunt.registerTask('deploy', ['clean', 'autoprefixer', 'cssmin', 'uglify', 'copy']
)

}
```

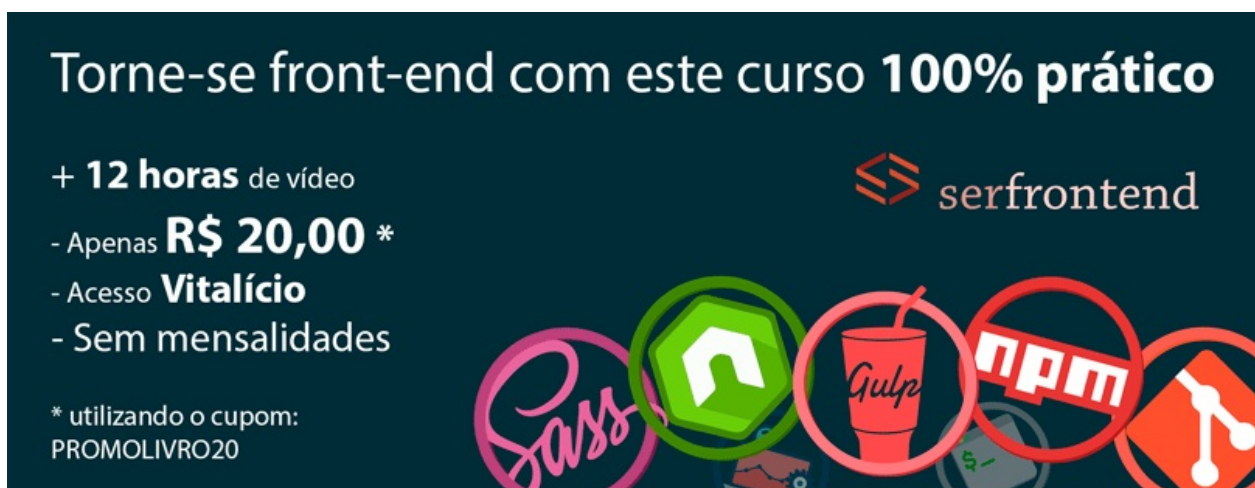
Considerações finais.

Nessa seção vimos um workflow bem básico de como trabalhar com o Grunt. Há muito mais coisa para ver com essa ferramenta, mas vou parando por aqui com a certeza de que, a partir da leitura dessa seção, você será capaz de brincar a vontade com os plugins prontos ou, quem sabe num futuro próximo, criar os seus próprios.

Deixarei aqui uma lista de plugins que já instalei e que já me salvaram algumas horas do meu dia, juntamente com o link e uma breve descrição. Guarde-os na manga.

- [watch](#). Um dos mais importantes. Como este plugin, você consegue dizer ao Grunt para monitorar seus arquivos fonte e rodar determinadas tarefas quando mudanças acontecerem. Por exemplo, você pode monitorar o seu index.html e, toda vez que esse arquivo for salvo, automaticamente rodar o copy para a pasta de deploy. É uma mão na roda.
- [jshint](#). Ferramenta automatizada para garantir a qualidade de seus códigos javascripts.
- [useref](#). Atualiza as referências de seus arquivos externos.
- [uncss](#). Se você, assim como eu, se incomoda em usar frameworks CSS por ter que fazer o usuário baixar um monte de classes css que não irá utilizar, não se preocupe. Seus problemas acabaram. O uncss limpa o seu CSS de acordo com o que está sendo usado em seu HTML. Bootstrap pesado nunca mais.

Na próxima seção, vamos ver como funciona o SASS, um pré-processador CSS famozinho. Até lá!



Torne-se front-end com este curso 100% prático

+ **12 horas** de vídeo

- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

serfrontend

Sass, Gulp, npm, Git

Pré-processadores de CSS

Durante muito tempo eu relutei em trabalhar com um pré-processador de CSS. Escrever CSS é tão simples que achava não precisar de uma ferramenta para deixá-lo mais simples ainda. Mas a questão aqui não é se escrever folhas de estilo é simples ou não. A questão é deixar a escrita do CSS mais rápida e de fácil manutenção. Eu estava errado.

Pense quantas vezes você precisou trocar um código hexadecimal em vários elementos. Provavelmente você fez uma busca em todo o seu arquivo CSS por uma determinada cor e a substituiu por uma nova cor, certo? E quando você precisou encontrar um trecho de CSS naquele seu arquivo com um milhão de linhas? Estas tarefas não são difíceis de serem executadas, é verdade. O 'F12' do Chrome te ajuda a encontrar a linha específica do CSS que você precisa editar e você sempre tem a opção de um 'replace all'. Mas digamos que esse processo, apesar de ser simples, é um pouco chato e deselegante.

Por isso está na hora de dar um up na maneira de escrever suas regras de folhas de estilos. Vamos nessa!

O que é preciso para começar?

Antes de tudo, precisamos escolher um pré-processador de CSS para dar início à esta etapa do nosso workflow. Há três que dominam o mercado, o [Stylus](#), o [Less](#) e o [Sass](#).

O Sass é, atualmente, o mais usado e é o que eu vou utilizar nesse guia, mas tenha em mente que os conceitos mostrados aqui são válidos para qualquer pré-processador que você venha a escolher.

Pense num pré-processador de CSS como sendo um software que recupera um código escrito em um formato diferente e o transforma na boa e velha folha de estilo em cascata, ou seja, você escreve num arquivo contendo outra extensão e o compilador gera o CSS comum. A vantagem é que com um pré-processador você ganha features que não estão disponíveis nativamente num CSS, como variáveis (mentira: <http://caniuse.com/#feat=css-variables>), loops, if e else, entre outras coisas legais.

É importante salientar que há duas formas de escrevermos com o Sass. Há a sintaxe `sass` (um pouco diferente do CSS tradicional. É baseado em indentações e quebras de linha) e o `.scss` (parecido com o CSS que estamos acostumados, por isso mesmo, é a minha sintaxe preferida. Esta sintaxe é baseada em chaves e ponto-e-vírgulas).

Além de escolhermos nosso pré-processador e também a sintaxe, precisamos decidir qual a ferramenta que vamos utilizar para transformar nossos arquivos `.scss` em `.css`. Há o [Ruby](#) e o [Libsass](#).

Originalmente, o Sass foi criado baseado em Ruby, porém, posteriormente surgiu o Libsass, escrito em C. O fato é que o compilador escrito em Ruby possui um melhor suporte às novas features, e por esse motivo será a minha escolha nesse guia (atualmente, início de 2016, acho que isto está mudando. Me senti tentado a escrever esse guia usando Libsass pela facilidade de não termos que instalar o Ruby, mas como tenho trabalhado com esta ferramenta de maneira satisfatória, darei preferência à ela).

Antes de qualquer coisa, como estamos num ambiente windows, precisamos instalar o Ruby. Vá em <http://rubyinstaller.org/> e faça o download. Depois... Ah!, você já sabe o que fazer.



[About](#) [Download](#) [Help](#) [Contribute](#)

The easy way to install Ruby on Windows

This is a **self-contained Windows-based installer** that includes the **Ruby language**, an execution environment, important **documentation**, and more.

[Download](#)

[Add-ons](#)



Latest News

RubyInstaller 2.0.0-p648, 2.1.8 and 2.2.4 released

These new releases of Ruby address a security issue (CVE-2015-7551). 2.1.8 and 2.2.4 also address some bugs and fixes. Upgrading to those versions is recommended. You can find the links to those archives in the download section.

February 04, 2016 [Read full article](#)

RubyInstaller 2.0.0-p645, 2.1.6 and 2.2.2 released

These new releases of Ruby address a security issue (CVE-2015-1855). 2.1.6 and 2.2.2 also address some bugs and fixes. Upgrading to those versions is recommended. You can find the links to those archives in the download section.

April 16, 2015 [Read full article](#)

Extras

Online Ruby Programming Course

If you're new to Ruby, check out this online course from The Pragmatic Studio to learn all the fundamentals of object-oriented programming with Ruby.

Online Rails Programming Course

If you're looking to create Ruby on Rails web apps, you'll learn how to build a complete Rails 4 app step-by-step in this online course also from The Pragmatic Studio.

15 Setup - Ruby 2.2.4-p230-x64

Ruby 2.2.4-p230-x64 License Agreement



Please read the following License Agreement and accept the terms before continuing the installation.

Copyright (c) 2007-2014 RubyInstaller Team.
All rights reserved.

Except:

Ruby is copyrighted free software by Yukihiro Matsumoto.
<http://www.ruby-lang.org/en/LICENSE.txt>

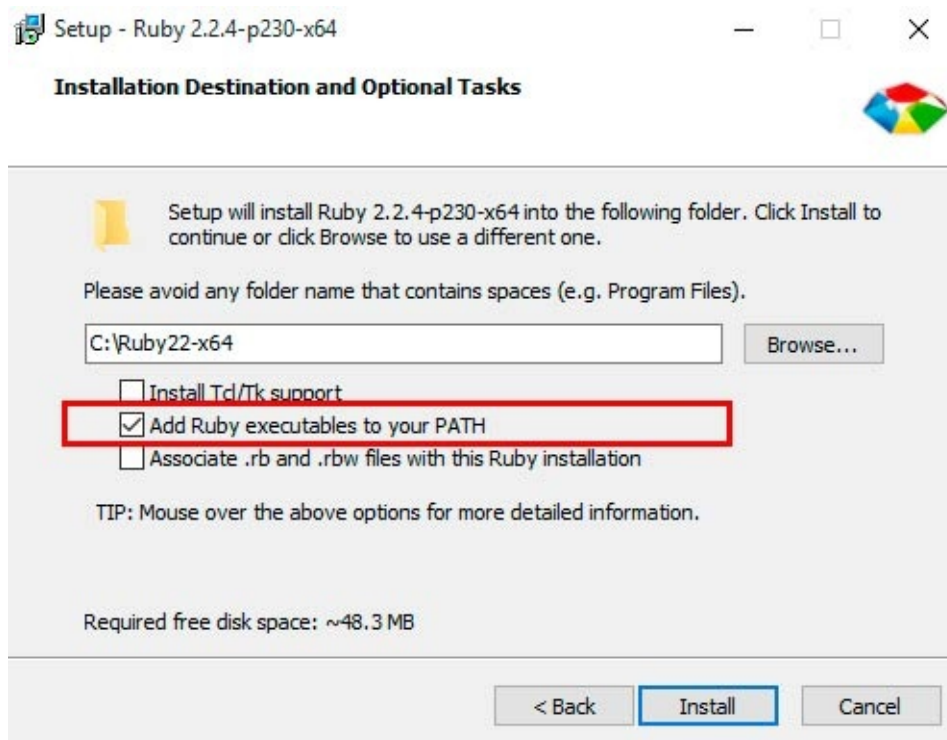
The Book of Ruby is copyrighted by Huw Collingbourne.
<http://www.sapphiresteel.com/The-Book-Of-Ruby>

- ☒ I accept the License
☐ I decline the License

[Next >](#)

[Cancel](#)

Mas atenção: Habilite a opção para adicionar uma variável PATH. Sem essa opção habilitada, você não conseguirá rodar o Sass da maneira como queremos.



Depois de instalado, abra o terminal do Ruby (isso é importante). É a partir dele que iremos instalar o Sass. Digite:

```
gem install sass
```

Talvez seja necessário digitar o comando acima com permissão de administrador usando a palavra `sudo`.

```
sudo gem install sass
```

Veja se tudo correu bem digitando

```
sass -v
```

Você deve visualizar a versão do sass instalado na sua máquina.

Se você optar por compilar seus arquivos usando o Libsass, não precisará instalar o Ruby. Nesse caso, como estamos usando o node, você precisará instalar um wrapper chamado [node-sass](#). Veja mais detalhes [nesse link](#).

Se você, assim como eu, instalou o Ruby e o Sass da maneira como descrevi acima, você será capaz de transformar seus arquivos .scss em .css através do próprio terminal do ruby. É algo muito simples. Basta digitar no terminal do ruby:

```
sass arquivo-fonte.scss arquivo-compilado.css
```

Porém, não entrarei em maiores detalhes sobre isso. A ideia é conseguirmos compilar nossos arquivos através do GruntJS e não através do terminal. Para isso, feche o seu terminal do ruby e abra o Git Bash. Como você já deve saber, precisamos instalar um wrapper do sass para o GruntJS, então...

```
$ npm install grunt-contrib-sass --save-dev
```

Depois disso, precisamos registrar essa tarefa no nosso Gruntfile.js

```
grunt.loadNpmTasks('grunt-contrib-sass')
```

Para configurar o Sass dentro de nosso Gruntfile, temos duas opções:

Opção 1 (simplificada):

```
sass: {  
  dist: {  
    files: {  
      'arquivo-compilado.css': 'arquivo-original.scss'  
    }  
  }  
}
```

Opção 2 (mais parruda):

```
sass: {
  dist: {
    files: [{
      expand: true, //você já sabe o que isso faz
      cwd: 'source/sass', // você também já sabe o que isso faz.
      src: ['**/*.scss'], // onde estão os arquivos fontes
      dest: 'deploy/css', // pasta de destino.
      ext: '.css' // extensão do arquivo final.
    }]
  }
}
```

Além disso, podemos passar algumas opções ao compilador. Veja:

Opção 1 (simplificada - com opções):

```
sass: {
  dist: {
    options: {
      //opções de configuração
    },
    files: {
      'arquivo-compilado.css': 'arquivo-original.scss'
    }
  }
}
```

Opção 2 (mais parruda - com opções):


```
sass: {
  dist: {
    options: {
      //opções de configuração
    },
    files: [{
      expand: true, //você já sabe o que isso faz
      cwd: 'source/sass', // Você também já sabe o que isso faz.
      src: ['**/*.scss'], // onde estão os arquivos fontes
      dest: 'deploy/css', // pasta de destino.
      ext: '.css' // extensão do arquivo fina.
    }]
  }
}
```

No próximo tópico vamos aplicar todo esse conhecimento no nosso projeto. Até lá!

Aplicando Sass em nosso projeto

Uma das principais vantagens em usar algum pré-processador de CSS é podermos modularizar a nossa folha de estilo em diversos arquivos independentes. A metodologia BEM, aplicada em nosso projeto, é meio caminho andado para separar responsabilidades de cada um dos componentes, deixando-os isolados e com formatações visuais bem definidas.

Em nosso projeto, tenho apenas dois componentes:

- mainHeader
- buttonsContainer

Em termos de estrutura de arquivos dentro de uma metodologia de desenvolvimento, eu poderia ter estes componentes inseridos em um diretório chamado `_modules`.

Existem várias maneiras de organizar seus arquivos sass. Vale dar uma pesquisada na internet sobre como organizar estes arquivos ou inventar uma maneira própria. Não há regras. Mas o que se vê muito por aí é termos, basicamente, três principais diretórios:

- `_modules/` - armazena a formatação de componentes independentes, como por exemplo, carousel, modal, accordion etc. Por serem independentes, estes componentes não devem sofrer alterações visuais se você alterar a posição deles no seu HTML. Tanto faz se o seu caroussel estiver numa section, num article ou numa div. O resultado deve ser o mesmo.
- `_partials/` - aqui é onde armazenamos as informações da estrutura do site. Os arquivos dessa pasta devem ser responsáveis pela formatação da grid, do header, do footer etc.
- `_base/` - aqui é onde criamos as configurações principais do site. Definimos as cores, as fontes, os css resets etc.

Como eu falei, não há uma regra. Sinta-se livre para modificar essa estrutura à vontade.

Irei criar dentro da pasta sass de nosso projeto apenas dois diretórios: `_base` e `_modules`. dentro de `_base` irei criar dois arquivos: `_variables.scss` e `_base.scss`. Dentro de `_modules` criarei também dois arquivos: `_mainHeader.scss` e `_buttonsContainer.scss`. Na raiz, criarei o `main.scss` (sem o underline).

Arquivos que contenham o underline inicial não serão compilados em arquivos CSS, ou seja, eles só podem ser importados para outros arquivos.

Dentro de sass/ há um arquivo `main.css`. Não vamos mais utilizar esse arquivo, mas vou deixá-lo aí apenas para referência. No final desse tópico podemos apagá-lo sem problemas.

Torne-se front-end com este curso 100% prático

+ **12 horas** de vídeo


- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

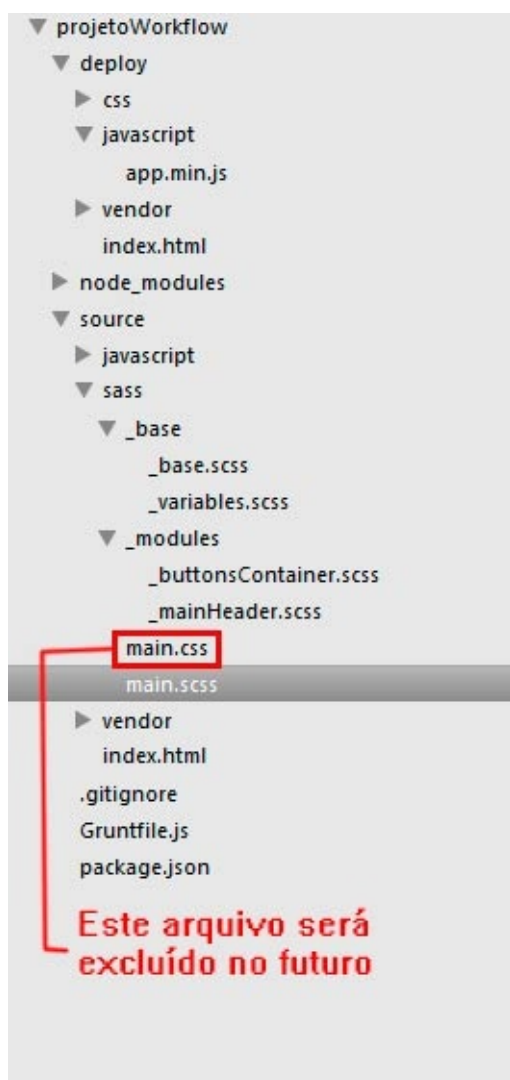
- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

 serfrontend



Segue uma imagem de como deve ficar a estrutura de arquivos.



Antes de começarmos a ver o Sass em ação, vou fazer uma pequena alteração em nosso Gruntfile.

Anteriormente, o autoprefixer usava o arquivo source/sass/main.css como base para salvar o novo deploy/css/main.css. Veja como era o nosso Gruntfile:

```
autoprefixer: {
  dist: {
    files: {
      'deploy/css/main.css': 'source/sass/main.css',
    },
  },
},
```

Só que agora não há mais arquivo .css, ou pelo menos não haverá no futuro. Por isso, farei a seguinte alteração:

```
autoprefixer: {
  dist: {
    files: {
      'deploy/css/main.css': 'deploy/css/main.css',
    },
  },
},
```

Repare que o arquivo de origem e destino é o mesmo. Quem irá gerar o main.css será o Sass antes da tarefa autoprefixer ser executada.

Precisamos, portanto, incluir a tarefa do sass.

```
sass: {
  dist: {
    files: {
      'deploy/css/main.css': 'source/sass/main.scss',
    },
  },
},
```

Além disso, se você ainda não o fez, vamos carregar o grunt-contrib-sass:

```
grunt.loadNpmTasks('grunt-contrib-sass');
```

E registrá-lo em nossa lista de tarefas:

```
grunt.registerTask('deploy', ['clean', 'sass', 'autoprefixer', 'cssmin', 'uglify', 'copy'])
```

No final, o nosso Gruntfile deve ficar semelhante à:

```
module.exports = function(grunt) {
  'use strict';

  grunt.initConfig({
    sass: {
      dist: {
        files: {
          'deploy/css/main.css': 'source/sass/main.scss',
        },
      },
    },
    autoprefixer: {
      dist: {
        files: {
          'deploy/css/main.css': 'deploy/css/main.css',
        },
      },
    },
    copy: {
      dist: {
        files: [
          {
            expand: true, //habilita o cwd
            cwd: 'source/', //relativo à source, mas não a inclui na cópi
a
            src: 'vendor/*',
            dest: 'deploy/'
          },
          {
            expand: true, //habilita o cwd
            cwd: 'source/',
            src: 'index.html',
            dest: 'deploy/'
          }
        ]
      },
    },
    clean: {
      dist: {
        src: ["deploy"]
      },
    },
    cssmin: {
      dist: {
        files: {
          'deploy/css/main.css': 'deploy/css/main.css'
        },
      },
    },
  },
```

```
    uglify: {
      options: {
        mangle: true
      },

      dist: {
        files: {
          'deploy/javascript/app.min.js': [
            'source/javascript/incrementButton.js',
            'source/javascript/date.js'
          ]
        }
      },
    },
  }

});

grunt.loadNpmTasks('grunt-autoprefixer');
grunt.loadNpmTasks('grunt-contrib-sass');
grunt.loadNpmTasks('grunt-contrib-cssmin');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-copy');
grunt.loadNpmTasks('grunt-contrib-clean');

grunt.registerTask('deploy', ['clean', 'sass', 'autoprefixer', 'cssmin', 'uglify',
'copy'])

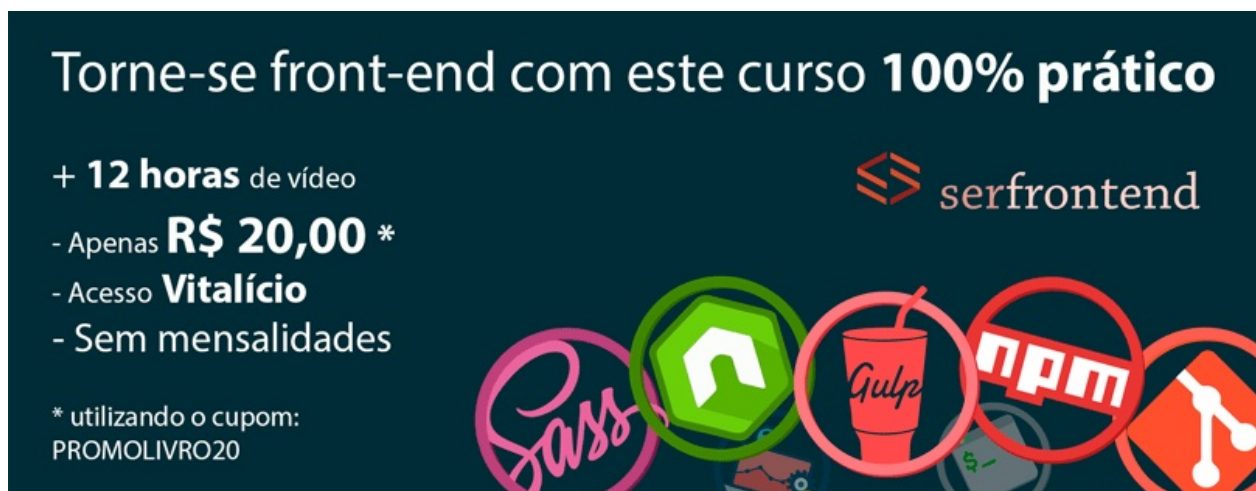
}
```

Pondo a mão na massa

A partir desse ponto temos o nosso Gruntfile configurado para compilar os nossos arquivos e também temos uma estrutura muito simples de diretório que os armazenará.

Tentarei aqui exemplificar algumas funcionalidades do Sass aplicado ao nosso mini-minuscúlo-pequeno projeto e como utilizá-lo para resolver problemas em projetos reais. Não falarei de todas as funcionalidades, mas como darei exemplos práticos, acredito que o aprendizado será muito mais proveitoso.

Uma grande vantagem do uso de qualquer pré-processador de css é poder trabalhar com variáveis. Em sites grandes, é muito comum ter que mudar uma propriedade como uma cor, um valor de opacidade ou o tamanho de uma fonte em mais de um lugar no código. É nesses cenários que as variáveis fazem sentido.



Torne-se front-end com este curso 100% prático


+ **12 horas** de vídeo




- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

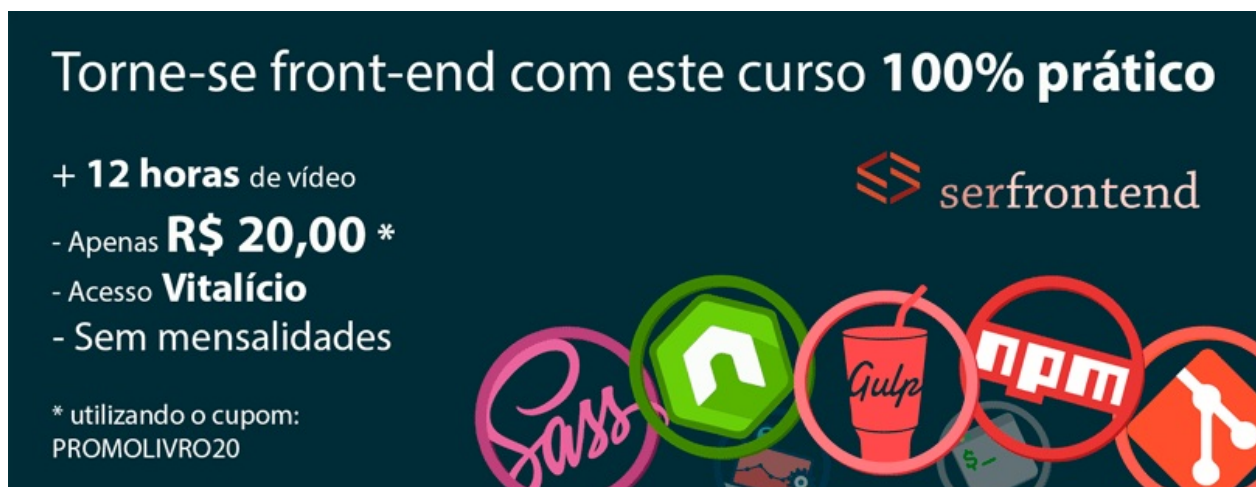
- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

 serfrontend

Variáveis



Torne-se front-end com este curso **100% prático**

+ **12 horas** de vídeo

- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

serfrontend

Sass Home Gulp NPM Git

Sabemos que o arquivo `_base/_variables.scss` deve conter as principais configurações visuais de todo o site/aplicativo. Portanto, vamos editá-lo com o seguinte código:

```
$bg-color: #ccc;  
$font: arial, sans-serif;
```

Repare que as variáveis em sass começam com um sinal de '\$'.

Agora, em `_base/_base.scss`, digite o seguinte código:

```
body{  
  background: $bg-color;  
  font-family: $font;  
}
```

Muito simples! A variável `$bg-color` armazena o valor `#ccc` e esta é usada na declaração CSS `background: $bg-color;` Ou seja, no arquivo compilado teremos o resultado:

```
background: #ccc;
```

Pode parecer bem bobinho neste exemplo, mas num código extenso, saber utilizar bem as variáveis pode te poupar muito retrabalho.

Se você se adiantou e rodou o comando no seu terminal para compilar os seus arquivos, você deve ter percebido que o `main.css` ficou sem nenhuma regra visual. Isso ocorre porque a tarefa do `grunt` compila o arquivo `main.scss` em `main.css`. Ou seja, para vermos o `sass` trabalhando precisamos importar nossos arquivos recém criados.

Declaração @import

Uma coisa bem legal no sass é que podemos separar os nossos códigos em vários arquivos e importá-los num arquivo "pai". Apesar da semelhança de escrita de um import do css, a importação do scss é totalmente diferente. Se em css puro o import não é bem visto por ter problemas com o cache e com múltiplas requisições http, em sass ele é quase que obrigatório. O import do sass faz com que os blocos de códigos sejam inseridos num único arquivo. Logo, o resultado será um arquivo contendo todos os blocos, um a um, na ordem em que foram inseridos.

Abra o seu main.scss e digite o seguinte:

```
@import "_base/variables",  
       "_base/base";
```

Repare que não precisamos incluir o underline nem a extensão. O Sass já sabe do que se trata. Se você incluir o underline ou a extensão, tudo continuará funcionando da mesma forma.

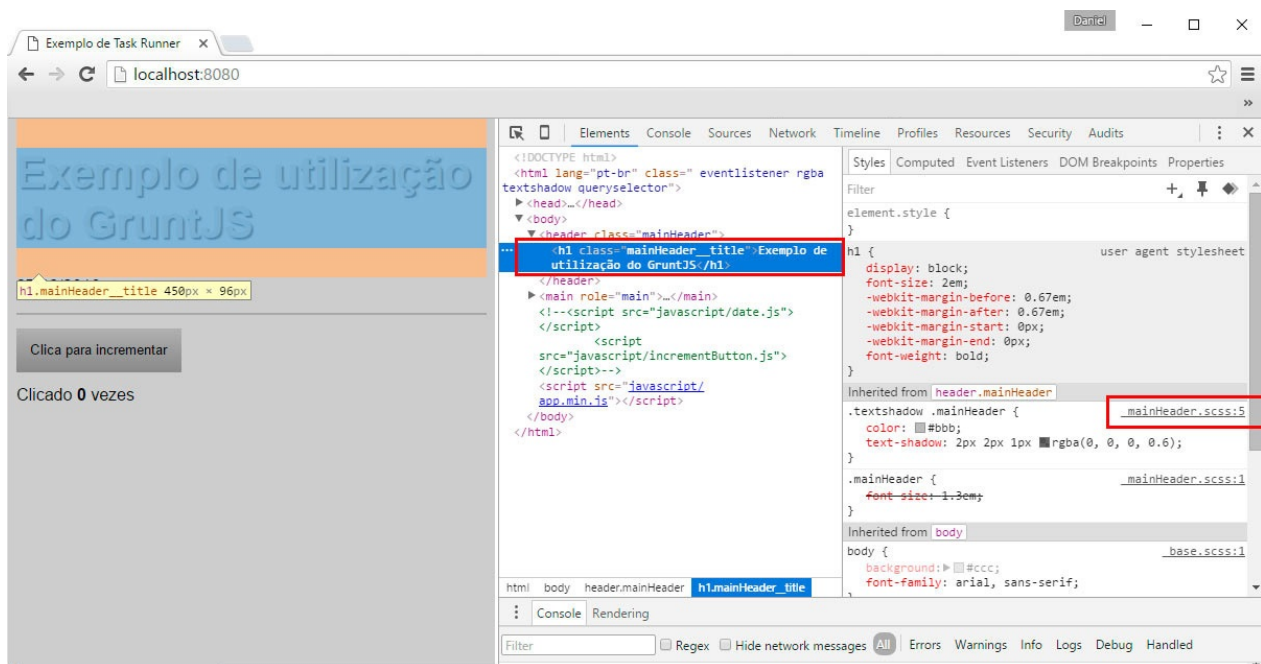
Agora sim, rode o seu grunt sass no terminal e veja o resultado.

Entendendo o arquivo .map

Coisa linda. Se tudo correu bem, temos um main.css compilado e um arquivo novo chamado main.css.map. Este arquivo é usado para debugar o código (você já deve ter se deparado com esse arquivo em bibliotecas como o jquery). Ele não deve ser levado para produção. Além disso, a referência `!# sourceMappingURL=main.css.map` deve estar presente apenas no ambiente de desenvolvimento.

Por enquanto, vou deixar o arquivo .map sendo criado normalmente. Em breve iremos configurar o sass para não gerar esse sourceMap.

Se você inspecionar o seu html pelo Chrome (em um servidor, pois o sourceMap não funciona em `file://...`) você verá que o Browser mostra a referência ao arquivo .scss e não ao .css, como era esperado. Assim, quando você tiver muitos arquivos, muitos componentes, muitas partials etc, ficará fácil saber quem é o sass responsável por aquela formatação. Se não existisse o .map, você teria que avaliar o arquivo compilado e "caçar" nos arquivos fontes. Isso seria uma grande perda de tempo.



O arquivo de extensão .map serve para nos ajudar no ambiente de desenvolvimento. Conforme podemos observar, ao inspecionar um elemento html, o web inspector do google chrome nos mostra o arquivo fonte .scss, e não o .css compilado. Isso é muito útil quando tivermos muitos componentes gerando um único e extenso arquivo final.

Referência ao seletor pai

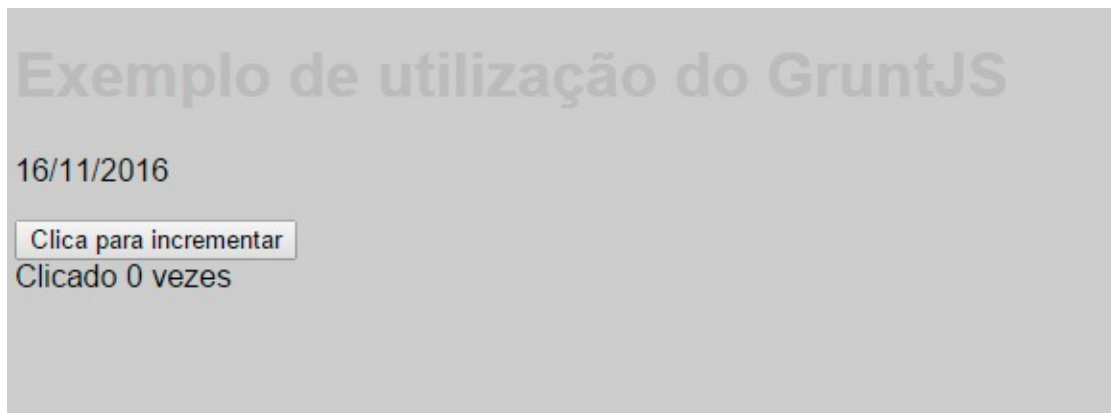
O nosso próximo passo é incluirmos o componente mainHeader. Para isso, crie o seguinte código em `_modules/_mainHeader.scss`

```
.mainHeader{  
  color: #bbb;  
  text-shadow: 2px 2px 1px rgba(0,0,0,.6);  
}
```

Em seguida, importe-o no main.scss e rode novamente `$ grunt sass`.

Nada de novo aqui. Porém, a grande sacada é que, mesmo que você não goste do IE, precisa gostar dos usuários que o utilizam, assim como Deus abomina o pecado, mas ama o pecador.

Sabemos que text-shadow [não funciona em IE antigo](#), e sinceramente, eu nem ligo. Mas o que eu não posso permitir é que os usuários do IE9 ou mais velhos tenham acesso prejudicado ao conteúdo simplesmente por utilizarem browsers antigos.



Conforme podemos observar, a legibilidade do título é prejudicada quando o browser não tiver suporte ao text-shadow. Ou seja, nestes casos, precisamos mostrar o texto com uma fonte um pouco mais escura.

Para resolver essa questão, vou utilizar o símbolo `'&'` que representa um seletor pai, juntamente com o modernizr que inserimos lá no começo do nosso guia.

Edite o arquivo da seguinte forma:

```
.mainHeader{  
  
  font-size: 1.2em;  
  
  .textshadow &{  
    color: #bbb;  
    text-shadow: 2px 2px 1px rgba(0,0,0,.6);  
  }  
  
  .no-textshadow &{  
    color: #717171;  
  }  
}
```

Incluí propositalmente um font-size para você avaliar o resultado. Abra o arquivo main.css em deploy e veja o que representa o sinal '&'. Perceba que fica muito mais fácil dar manutenção. Você pode até não se dar conta disso agora, mas tente implementar esse conceito no próximo código que você precisar escrever.

Repare que no exemplo acima eu também aninhei seletores do CSS, algo muito usado (mas que também pode se tornar um pesadelo se não for usado com cautela). Veremos isso daqui a dois segundos.

Aninhamento de seletores

Torne-se front-end com este curso **100% prático**





+ **12 horas** de vídeo

- Apenas **R\$ 20,00 ***

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20



Entender como funciona o aninhamento de seletores no Sass é algo que não exige maiores explicações por ser bastante auto-explicativo. Veja um exemplo:

O .scss ...

```
#menu{
  ul{
    margin: 0px;
    padding: 0;
    li{
      list-style-type: none;
      a{
        text-decoration: none;
      }
    }
  }
}
```

... gera o.css:

```
#menu {  
  background: #ddd;  
}  
#menu ul {  
  margin: 0px;  
  padding: 0;  
}  
#menu ul li {  
  list-style-type: none;  
}  
#menu ul li a {  
  text-decoration: none;  
}
```

O problema do aninhamento é a sua facilidade. É tão fácil que se não tomarmos cuidado, podemos acabar com seletores do tipo:

```
main #mainContent #header #menu ul li a span.icon
```

Nós dois sabemos que seletores desse tipo são péssimos em termos de performance na renderização, os arquivos ficam mais pesados, ficam deselegantes e denigrem sua imagem como desenvolvedor. Nunca faça seletores desse tipo. (Para mim, isso é pior que usar o Dreamweaver ;)

Mas nós podemos usar o aninhamento para gerar o efeito hover em um link, juntamente com a referência ao seletor pai '&'. Veja:

o arquivo .scss ...

```
a{  
  text-decoration: none;  
  &:hover{  
    text-decoration: underline;  
  }  
}
```

... gera o seguinte .css:

```
a{  
  text-decoration: none;  
}  
a:hover{  
  text-decoration: underline;  
}
```


Juntando o aninhamento, variáveis e seletor pai

O próximo passo é criarmos o nosso componente `buttonsContainer`. Abra esse arquivo e edite-o incluindo os aninhamentos necessários. Lembre-se que ele está em `_modules/`.

.SCSS

```
.buttonsContainer{
  border-top: 2px solid #A5A5A5;

  .buttonsContainer .buttonsContainer__button{
    background-image: linear-gradient(to top, #848484, #AFAFAF);
    border: none;
    padding: 1em;
    margin: 1em 0;
    cursor: pointer;
    outline: none;

    &:hover{
      background-image: linear-gradient(to top, #AFAFAF, #848484);
    }
    &:active{
      background-image: linear-gradient(to top, #868686, #565656);
    }
  }

  .buttonsContainer .buttonsContainer__result--number{
    font-weight: bold;
  }
}
```

Se você olhar bem, verá que há cores que se repetem. Parece interessante criarmos variáveis, não acha? Um detalhe importante é que o sass também possui escopo de variáveis. Veja um exemplo:

```
.main{
  $cor: red;
  h1{
    color: $cor;
  }
}
div {
  background: $cor;
}
```

O código acima irá gerar um erro pois a variável \$cor não existe no contexto da div.

Sabendo disso, vamos editar nosso componente

```
.buttonsContainer{
  $cor1: #848484;
  $cor2: #AFAFAF;
  border-top: 2px solid #A5A5A5;

  .buttonsContainer__button{
    background-image: linear-gradient(to top, $cor1, $cor2);
    border: none;
    padding: 1em;
    margin: 1em 0;
    cursor: pointer;
    outline: none;

    &:hover{
      background-image: linear-gradient(to top, $cor2, $cor1);
    }
    &:active{
      background-image: linear-gradient(to top, #868686, #565656);
    }
  }
}

.buttonsContainer__result--number{
  font-weight: bold;
}
```

Não há nada de errado com o código acima. Mas ele ainda pode ser melhorado. Se prestarmos atenção, todas as cores são variações de cinza. E se tivéssemos uma função que consiga clarear ou escurecer um cor? Ah! sim. [Nós temos esse tipo de coisa no sass.](#)

Funções relacionadas a cores

Para facilitar minha vida, criarei uma variável para a cor cinza mais clara dentro do contexto de nosso componente.

Onde eu precisar escurecer minha cor, chamarei uma função do Sass chamada `darken($color, $amount)`. O segundo parâmetro é referente à porcentagem de escurecimento.

```
.buttonsContainer{  
  
  $cor: #AFAFAF;  
  
  border-top: 2px solid #A5A5A5;  
  .buttonsContainer__button{  
    background-image: linear-gradient(to top, darken($cor, 15), $cor);  
    border: none;  
    padding: 1em;  
    margin: 1em 0;  
    cursor: pointer;  
    outline: none;  
  
    &:hover{  
      background-image: linear-gradient(to top, $cor, darken($cor, 15));  
    }  
    &:active{  
      background-image: linear-gradient(to top, darken($cor, 14), darken($cor, 34  
));  
  }  
}  
  
  .buttonsContainer__result--number{  
    font-weight: bold;  
  }  
}
```

Eu usei [este site](#) para me ajudar a escolher o valor do segundo parâmetro da função `darken()`.

Está ficando bom, mas ainda pode ser melhorado. Você deve ter percebido que incluir a variável da cor dentro do contexto do container não é uma boa ideia. Se no futuro aparecer no layout um botão com destaque, por exemplo, teríamos que duplicar código (e é justamente para não ter que fazer isso que estamos usando o Sass).

@mixin

Torne-se front-end com este curso **100% prático**

+ 12 horas de vídeo

- Apenas **R\$ 20,00** *

- Acesso **Vitalício**

- Sem mensalidades

* utilizando o cupom:
PROMOLIVRO20

serfrontend

Para resolver o problema de reutilização de código comentado no tópico anterior, criei um @mixin que gere todo o código do botão e que receba por parâmetro a cor base. Farei isso da seguinte maneira:

```
@mixin gerarBotao($cor){
  background-image: linear-gradient(to top, darken($cor, 15), $cor);
  border: none;
  padding: 1em;
  margin: 1em 0;
  cursor: pointer;
  outline: none;

  &:hover{
    background-image: linear-gradient(to top, $cor, darken($cor, 15));
  }
  &:active{
    background-image: linear-gradient(to top, darken($cor, 14), darken($cor, 34));
  }
}
```

O @mixin é o responsável por criar dinamicamente todo o código do botão, incluindo o hover e o active. Há no sass a opção de criarmos uma @function, porém, em nosso exemplo, uma @function não nos ajudaria, uma vez que esta retorna apenas um valor, enquanto um @mixin retorna diversos valores.

Tenha em mente que, ao criarmos um @mixin, precisamos utilizá-lo usando a palavra @include. Se usarmos uma @function, temos que retornar o valor com a palavra @return e chamar a função do local que queremos usar aquele valor.

Apenas para exemplificar o uso de uma @function, vou criar uma função que recebe dois valores e calcula a porcentagem:

```
@function calc-percent($target, $container) {  
  @return ($target / $container) * 100%;  
}
```

Quando precisarmos utilizar essa função, teremos que chamá-la de dentro da regra css.

```
.modulo {  
  width: calc-percent(500px, 1000px);  
}
```

Voltando ao nosso exemplo, para utilizar o @mixin criado, precisamos da palavra @include. Veja como deve ficar:

```
@mixin gerarBotao($cor){  
  background-image: linear-gradient(to top, darken($cor, 15), $cor);  
  border: none;  
  padding: 1em;  
  margin: 1em 0;  
  cursor: pointer;  
  outline: none;  
  
  &:hover{  
    background-image: linear-gradient(to top, $cor, darken($cor, 15));  
  }  
  &:active{  
    background-image: linear-gradient(to top, darken($cor, 14), darken($cor, 34));  
  }  
}  
  
.buttonsContainer{  
  
  border-top: 2px solid #A5A5A5;  
  
  .buttonsContainer__button{  
    @include gerarBotao(#AFAFAF)  
  }  
  
  .buttonsContainer__result--number{  
    font-weight: bold;  
  }  
  
}
```

E se no futuro precisarmos de uma variação desse mesmo botão, poderemos simplesmente passar uma nova cor como parâmetro:

```
.buttonsContainer__button{
  @include gerarBotao(#AFAFAF)
}

.buttonsContainer__button--highlight{
  @include gerarBotao(red)
}
```


Considerações Finais

A abordagem apresentada até este ponto não é a única possível, mas o meu intuito não é te apresentar tudo que está disponível no Sass, mas sim te mostrar os primeiros passos para que você consiga desenvolver os seus estudos. [Nesse site](#) há mais alguns exemplos básicos sobre a utilização do Sass, em especial, veja com atenção sobre `@extend` e operadores.

Todo esse conhecimento passado até aqui foi bem básico, mas se você quiser conhecer o Sass pra valer, vá até a [documentação](#).

Antes de finalizarmos, devemos remover as referencias aos arquivos `.map`.

Removendo o sourceMap de produção

Lembra que eu falei que não devemos colocar em produção os arquivos `.map` e que devemos remover as referências à estes arquivos? Pois então, para fazer isso, basta incluir uma opção nas configurações do Sass em nosso `Gruntfile.js`. Veja:

```
sass: {
  options: {
    sourceMap: false
  },
  dist: {
    files: {
      'deploy/css/main.css': 'source/sass/main.scss',
    },
  },
}
```

automação da automação

Para compilar seus arquivos, você pode digitar no terminal o nome da sua tarefa ou fazer com que, toda vez que um arquivo `.scss` seja alterado e salvo, o GruntJS rode automaticamente a tarefa necessária. Para isso você precisará instalar o [watch do grunt](#). Este wrapper é responsável por monitorar alterações nos seus arquivos (`.scss`, por exemplo) e rodar determinadas tarefas quando estes arquivos forem salvos. Dessa forma,

configurando corretamente, você consegue fazer com que o grunt rode a sua tarefa sass toda vez que um arquivo .scss for salvo, sem que você tenha digitar o tempo todo comandos no terminal para esta tarefa.

Fallback com o Modernizr

No componente buttonsContainer, você pode utilizar o sinal de & para criar um fallback de css para quando o browser não tiver suporte à todas as features utilizadas em nossa folha de estilo.

Uma palavra sobre o BEM.

Nesse ponto nosso arquivo está funcionando perfeitamente. Vimos como trabalhar com variáveis, @mixins/@include, seletor & e aninhamento de seletores. Dei uma rápida passada sobre as @function/@return e pedi para que você visse por conta própria sobre @extend e os operadores. Isso cobre uma parte de qualquer pré-processador, mas fique ciente que ainda há muito que não foi discutido.

Para encerrar, gostaria de sugerir que você use com cautela o aninhamento de seletores. O meu exemplo foi para uso didático, mas creio que possa ser melhorado.

Se olharmos com atenção para a metodologia BEM, vemos que não faz muito sentido termos seletores desse tipos:

```
.buttonsContainer .buttonsContainer__button
```

O seletor acima poderia ser facilmente substituído por:

```
.buttonsContainer__button
```

Concorda?

Se achar válido, altere seus arquivos .scss para deixar os seletores do arquivo compilado mais simples, quando possível.

Até breve

Espero que este guia tenha te ajudado a dar os primeiros passos com as novas tecnologias (nem tão novas assim, mas são novas para mim) que norteiam o universo de desenvolvimento front-end.

Não estou encerrando esse guia definitivamente, entretanto, terei que colocá-lo em stand-by por um tempo pois, apesar de não ter falado tudo o que eu queria, apareceu um outro projeto pessoal que me tomará muito tempo.

Vimos o básico necessário para que você, se não conhecia nada a respeito, possa dar passos mais largos do que estes que demos juntos. O meu objetivo foi criar um material para iniciantes pois, quando eu precisei conhecer todas essas coisas que falei aqui - início de 2015 - tive muita dificuldade para encontrar material didático para um noob como eu que ficou durante muitos anos escrevendo apenas HTML, CSS e Javascript.

2015 foi o ano em que desenvolvi bastante meus conhecimentos, mas ainda tenho muito o que aprender (TDD e integração contínua, estou olhando pra vocês). Quem sabe eu não venha por aqui no início de 2017 para contar as coisas que aprendi nesse ano.

Abração

Quer bater um papo comigo?

Dá uma olhada no meu [twitter](#)

Também tenho um projeto pessoal onde ministro aulas sobre front-end. Dá uma conferida em [serfrontend.com](#).