



Alexandre Afonso
Thiago Faria

FULLSTACK ANGULAR & SPRING

Guia para se tornar um desenvolvedor moderno



Fullstack Angular e Spring - Guia para se tornar um desenvolvedor moderno

por Alexandre Afonso e Thiago Faria

Versão 1.0, 30/05/2018

© 2018 AlgaWorks Softwares, Treinamentos e Serviços Ltda. Todos os direitos reservados.

Nenhuma parte deste livro pode ser reproduzida ou transmitida em qualquer forma, seja por meio eletrônico ou mecânico, sem permissão por escrito da AlgaWorks, exceto para resumos breves em revisões e análises.

AlgaWorks Softwares, Treinamentos e Serviços Ltda

www.algaworks.com

contato@algaworks.com

+55 (11) 2626-9415

Siga-nos nas redes sociais e fique por dentro de tudo!

[Facebook](#)

[YouTube](#)

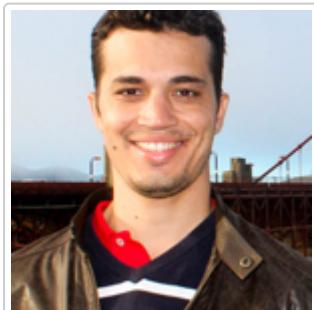
Sobre os autores



Alexandre Afonso

Instrutor Java na AlgaWorks, graduado em Sistemas de Informação, está no mercado de programação Java há mais de 10 anos, principalmente no desenvolvimento de sistemas corporativos.

LinkedIn: <https://www.linkedin.com/in/alexandreadfon>



Thiago Faria de Andrade

Fundador e instrutor na AlgaWorks. Certificado como programador Java pela Sun, autor e co-autor de diversos livros de Java, Java EE, JPA, JSF e Angular. Palestrante da maior conferência de Java do mundo (JavaOne San Francisco). Thiago iniciou seu interesse por programação aos 14 anos de idade (1995), quando desenvolveu o primeiro jogo de truco online e multiplayer do mundo. Já foi sócio e trabalhou em outras empresas de software como programador, gerente e diretor de tecnologia, mas nunca deixou de programar.

LinkedIn: <https://www.linkedin.com/in/thiagofa>

Antes de começar...

Antes que você comece a ler esse livro, nós gostaríamos de combinar algumas coisas com você, para que tenha um excelente aproveitamento do conteúdo. Vamos lá?

O que você precisa saber?

Você só precisa conhecer um pouco de programação para absorver o conteúdo deste livro. Não necessariamente precisa ter trabalhado na área e também não precisa de muita experiência.

Como obter ajuda?

Durante os estudos, é muito comum surgir várias dúvidas. Nós gostaríamos muito de te ajudar pessoalmente nesses problemas, mas infelizmente não conseguimos fazer isso com todos os leitores do livro, afinal, ocupamos grande parte do dia ajudando os alunos de cursos online na AlgaWorks.

Então, quando você tiver alguma dúvida e não conseguir encontrar a solução no Google ou com seu próprio conhecimento, nossa recomendação é que você poste na nossa Comunidade Java no Facebook. É só acessar:

<http://alga.works/comunidadejava/>

Como sugerir melhorias ou reportar erros sobre este livro?

Se você encontrar algum erro no conteúdo desse livro ou se tiver alguma sugestão para melhorar a próxima edição, vamos ficar muito felizes se você puder nos dizer.

Envie um e-mail para livros@algaworks.com.

Onde encontrar uma cópia do livro?

Nós planejamos adotar melhoria contínua nesse livro. Por isso é importante você verificar através do link <http://alga.works/livro-fsas> se está com a versão mais atualizada.

Ajude na continuidade desse trabalho

Escrever um livro (mesmo que pequeno, como esse) dá muito trabalho, por isso, esse projeto só faz sentido se muitas pessoas tiverem acesso a ele.

Ajude a divulgar esse livro para seus amigos que também querem se tornar Desenvolvedores Fullstack Angular e Spring. Compartilhe no Facebook e Twitter!



Sumário

1 Introdução

1.1	Front-end	10
1.2	Back-end	11
1.3	Desenvolvedor Fullstack	12

2 Arquitetura

2.1	Arquitetura tradicional	14
2.2	Arquitetura moderna	16
2.3	Arquitetura tradicional versus moderna	20
2.4	Single-Page Applications	21

3 REST

3.1	Por que usar REST?	26
3.2	Melhores práticas na adoção de REST	27
3.3	É REST ou RESTful?	29
3.4	Protocolo HTTP	30
3.5	URI e recursos	34
3.6	Métodos	36

4 Ecossistema Spring

4.1	Spring Framework	40
4.2	Spring Security	42
4.3	Spring Boot	43

4.4	Spring Tool Suite - STS	45
-----	-------------------------------	----

5 Angular

5.1	Angular vs AngularJS	48
5.2	O versionamento do Angular	50
5.3	Como está o mercado?	51
5.4	A linguagem TypeScript	52
5.5	Elementos de uma aplicação Angular	54
5.6	Bibliotecas de componentes	60
5.7	Ambiente de desenvolvimento	64

6 Conclusão

6.1	Próximos passos	67
-----	-----------------------	----

Capítulo 1

Introdução

No mundo de desenvolvimento de software, existem três termos muito comuns que provavelmente você já se deparou: *front-end*, *back-end* e *fullstack*.

É muito comum vermos vagas de empregos com os títulos: “Programador Front-end”, “Programador Back-end” e “Programador Fullstack.”

Mas o que isso significa?

Vamos responder nos próximos tópicos deste capítulo.

1.1. Front-end

O *front-end* é todo o código da aplicação responsável pela apresentação do software (*client-side*). Em se tratando de aplicações web, é exatamente o código do sistema que roda no navegador.

Um desenvolvedor *front-end*, geralmente, trabalha com linguagens como HTML, CSS e JavaScript, além de frameworks e bibliotecas, como por exemplo Angular (um dos assuntos deste livro).

Existem diversas oportunidades no mercado para desenvolvedores especialistas em *front-end*. Nestes casos, esses programadores não conhecem nada ou conhecem muito pouco de *back-end*.

São pessoas que preferiram se especializar no *front* da aplicação.

Embora esses programadores não precisem conhecer como desenvolver código de *back-end* (vamos falar sobre isso daqui a pouco), é extremamente importante que eles conheçam os fundamentos sobre a arquitetura do software, porque afinal, o código que eles produzem fazem parte de um todo e se comunica com o *back-end*.

Desenvolvedores *front-end* não lidam diretamente com banco de dados, servidores de aplicação complexos e várias outras coisas que só quem trabalha com *back-end* conhece.

Os desafios são outros: criar páginas ou telas com boa usabilidade e carregamento rápido, garantir o funcionamento nos diferentes navegadores, integrar com os serviços do *back-end*, etc.

1.2. Back-end

O *back-end* é a parte do software que roda no servidor, por isso também é conhecida como *server-side*.

É o *back-end* que fornece e garante todas as regras de negócio, acesso a banco de dados, segurança e escalabilidade.

Embora o *front-end* também possa ter algumas regras e validações, é o *back-end* que deve garantir a integridade dos dados.

Desenvolvedores que preferem se especializar apenas em *back-end*, geralmente, não são muito bons em deixar páginas bonitas e com boa usabilidade, mas são melhores em regras de negócio, banco de dados e todas as coisas que rodam no servidor.

Esses desenvolvedores trabalham com linguagens de programação como, por exemplo, Java, C#, PHP, Python, Ruby ou até mesmo JavaScript (sim, tem jeito de rodar JavaScript no servidor também).

Cada linguagem de programação é um mundo a parte, com comunidades, eventos, frameworks, livros, cursos, etc.

Neste livro, nós vamos falar de Spring, que é um ecossistema de projetos que nos ajuda a desenvolver aplicações de *back-end* em Java.

E aliás, Java é a linguagem de programação mais usada no mundo, segundo o *TIOBE Index*.

1.3. Desenvolvedor Fullstack

Quem trabalha tanto com *front-end* quanto *back-end* é conhecido como “Desenvolvedor Fullstack” ou “Programador Fullstack”.

Esse é um tipo de profissional mais completo, que pode entregar um projeto do início ao fim, sem necessariamente precisar de ajuda de outra pessoa para criar uma parte do sistema.

Naturalmente, é um profissional mais valorizado no mercado, especialmente se for realmente bom em *front-end* e *back-end* de forma semelhante, ou seja, não é um desenvolvedor meia boca em *front* ou *back*.

Para se tornar um Desenvolvedor Fullstack reconhecido e valorizado no mercado, o primeiro passo é conhecer os fundamentos da arquitetura e tecnologias que você vai utilizar.

É um grande erro começar a aprender desenvolvimento de software pelas tecnologias, sem entender antes a base da arquitetura.

Exatamente por esse motivo que milhares de pessoas ficam “patinando” por vários e vários meses e até anos, e nunca aprendem corretamente como criar software profissional.

Imagine um trabalhador da construção civil aprender a assentar janelas e enfileirar tijolos, sem antes entender como tudo isso vai se sustentar? Bom, é quase a mesma coisa com software!

A boa notícia é que agora você tem este livro, que tem como objetivo ser um guia para você aprender os fundamentos de uma arquitetura moderna e como algumas tecnologias que estão no auge e em pleno crescimento se encaixam.

Vamos falar de uma arquitetura e tecnologias usadas por empresas no mundo todo, inclusive no Brasil, como Google, Microsoft, Citibank, NBA, Netflix, PayPal, etc.

Ao finalizar este livro, você será capaz de identificar, entender e até discutir sobre uma arquitetura moderna de desenvolvimento de software, onde o *back-end* é entregue com projetos do ecossistema Spring e o *front-end* com Angular.

Capítulo 2

Arquitetura

A evolução dos projetos de software facilita nossas vidas em uma medida que agora podemos ir além e ainda manter um bom nível de produtividade.

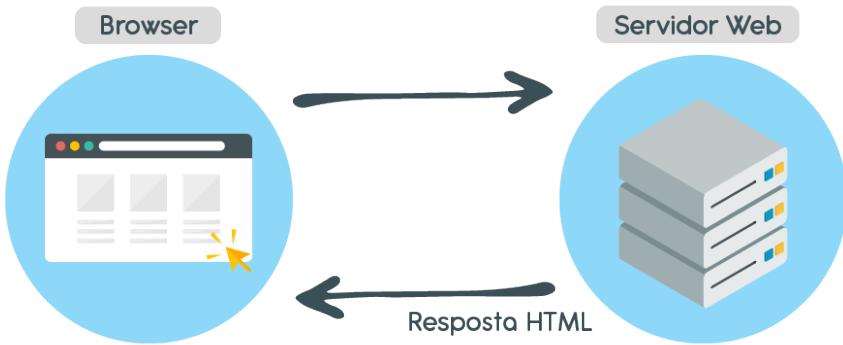
E para explicarmos melhor o que significa esse “ir além”, precisamos passar para você o conceito da **arquitetura tradicional** e da **arquitetura moderna**.

Embora esses termos não sejam tão conhecidos ou existam variações, os dois modelos de arquitetura são amplamente usados, então usar estes termos no livro vai facilitar na nossa explicação sobre cada tipo de arquitetura.

Após entender esses dois conceitos, vamos poder explicar quais as vantagens que você terá ao adotar as tecnologias, técnicas e dicas que passaremos no livro.

2.1. Arquitetura tradicional

Nosso foco neste livro é falar sobre desenvolvimento de aplicações web, e essa é a arquitetura mais tradicional e usada nos últimos anos para desenvolvimento web.



Nesta arquitetura, temos os seguintes elementos:

- Browser
- Servidor web

Basicamente, quando um browser (navegador) faz uma requisição web, ela é recebida pela aplicação servidora, que está em alguma máquina na internet ou intranet.

Lembrando que ainda não estamos falando especificamente de qualquer tecnologia, então essa aplicação poderia ter sido desenvolvida em Java, C#, PHP, Python, etc.

Quando a requisição chega na máquina servidora, a aplicação recebe e faz um primeiro processamento, de modo a entender qual a funcionalidade do sistema deve ser executada.

Identificado isso, a funcionalidade é executada e, se houverem dados a serem persistidos em um banco de dados, esse é o momento que tudo acontece.

No final, ainda na aplicação que está na máquina servidora, será montado dinamicamente um código HTML com os dados mesclados, para a resposta.

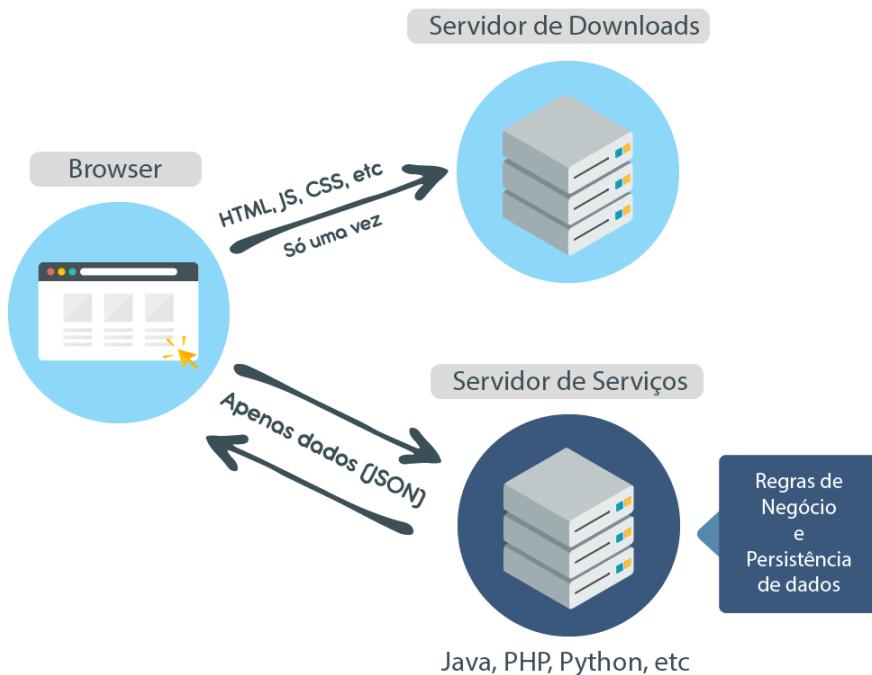
Depois de processada, o código HTML da página é colocado na resposta. O navegador recebe essa resposta e renderiza (exibe) a página com os dados para o usuário.

O cenário descrito até aqui ainda vigora no mundo das aplicações web. Não tem nada de errado com essa arquitetura e ninguém pode dizer que ela vai deixar de existir, porque é a base para o desenvolvimento de projetos web.

Inclusive, é interessante que você aprenda a desenvolver projetos nessa arquitetura também, mas não é o foco deste livro.

2.2. Arquitetura moderna

Agora veja também um diagrama que representa a outra arquitetura, que estamos chamando de “arquitetura moderna”:



Nesta arquitetura, temos os seguintes elementos:

- Browser
- Servidor de downloads
- Servidor de serviços

Servidor de serviços

Para ficar mais fácil o entendimento, vamos começar pelo **Servidor de serviços**.

Este servidor nada mais é que um servidor web, onde o *back-end* da aplicação está hospedado. Ou seja, todas as regras de negócio e código de acesso ao banco de dados rodam neste servidor.

A aplicação que roda no servidor de serviços pode ser desenvolvida em qualquer linguagem para servidores, como por exemplo Java, C#, Python, Ruby, PHP, JavaScript, etc.

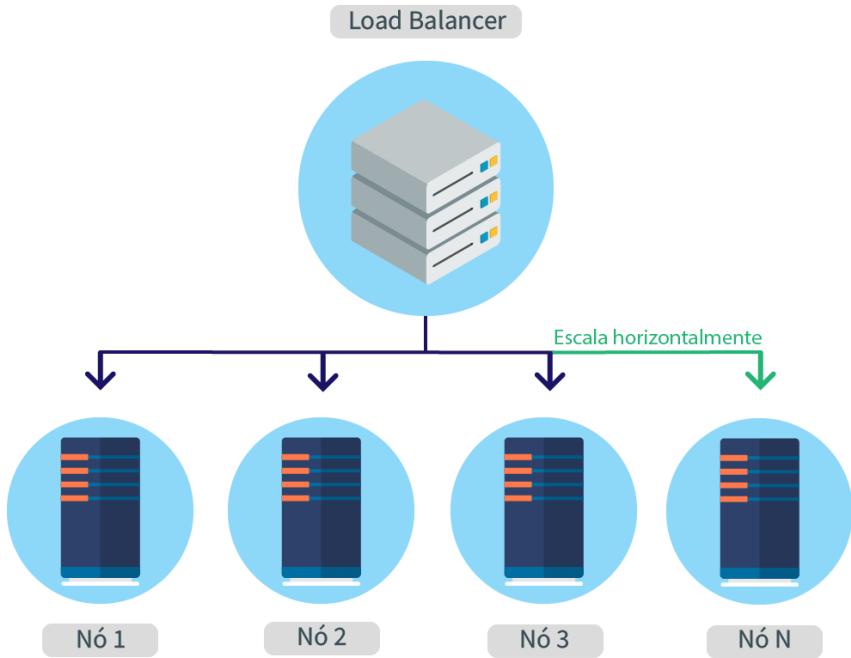
Em se tratando de Java, podemos usar frameworks para nos ajudar, como o Spring, que vamos abordar neste livro.

Basicamente, um servidor de serviços expõe uma API (Application Programming Interface) para acesso às regras de negócio e aos dados.

Qualquer cliente que tenha autorização pode usar esses serviços. Pode ser o *front-end* web de uma outra aplicação, um aplicativo de celular, um programa desktop ou até mesmo uma geladeira.

Note que o servidor de serviços é independente do *front-end*. Essa é uma das principais vantagens desta arquitetura.

Outra vantagem desta arquitetura é que os serviços podem ser escalados verticalmente com muito mais facilidade, porque eles não armazenam sessões de usuários, ou seja, se precisarmos de mais poder computacional, é só adicionar novos servidores com a mesma aplicação *back-end* implantada abaixo de um *load balancer* (balanceador de carga).



Servidor de downloads

Pelo menos conceitualmente, o servidor de serviços não hospeda o *front-end* da aplicação, mas apenas o *back-end*. Não existe nenhum código no servidor de serviços que apresenta qualquer página ou tela.

Então, dentro do contexto de uma aplicação onde o acesso é feito por um navegador web, precisamos de um lugar para hospedar o código do *front-end* web.

Neste cenário, geralmente desenvolvemos uma aplicação de página única (*Single-Page Applications*), que vamos discutir com mais detalhes em breve.

Esse tipo de aplicação é desenvolvida com HTML, CSS e JavaScript, e roda 100% no browser do usuário.

Mas precisamos hospedar esses arquivos de *front-end* em algum lugar, e eis que surge o servidor de downloads.

Basicamente, quando o usuário digita no navegador a URL da aplicação web, por exemplo `http://estoque-ui.algaworks.com`, esse domínio direciona para um servidor de downloads.

Como o código de *front-end* é estático, o servidor de downloads é, geralmente, um servidor HTTP simples, como *Apache HTTP Server* ou *Nginx*. Não existe nenhuma necessidade de usar servidores de aplicação mais complexos, como *Tomcat*, *Wilfly*, etc.

Simplesmente, o servidor de downloads deve enviar para o browser os arquivos solicitados, para serem executados na máquina do próprio cliente, dentro do navegador.

Repare na imagem que temos a representação do servidor de downloads feita de forma isolada. A intenção é que você entenda que este servidor é independente, mas tecnicamente, o servidor de downloads pode ficar na mesma máquina física e até mesmo usar o mesmo servidor web do servidor de serviços, embora não seja recomendado.

Browser

Dentro do contexto de uma aplicação onde o acesso é feito por um navegador web, temos, claro, o próprio navegador iniciando o processo.

Antes de fazer uma requisição para o servidor de serviços da aplicação, o *front-end* web da aplicação precisa ser carregado no navegador.

Ao digitar a URL da aplicação *front-end* no navegador, os arquivos HTML, CSS e JavaScript (estáticos) são carregados e executados no próprio navegador, afinal, é para isso que os browsers servem, né?

Lembre que essa requisição dos arquivos estáticos é feita para o servidor de downloads, então temos uma requisição e resposta do browser para o servidor de downloads.

O *front-end* da aplicação é exibido e, geralmente, aguarda alguma interação do usuário dentro do sistema.

Imagine, por exemplo, que o usuário faça uma pesquisa de clientes no *front-end*, usando seu navegador. Neste caso, o código implementado no *front-end* deve disparar uma requisição para o servidor de serviços, ordenando que uma pesquisa de clientes seja feita.

O servidor de serviços executa as regras de negócio, faz a conexão com o banco de dados, se for o caso, executa a pesquisa e devolve o resultado dela na resposta. Essa resposta inclui apenas dados em um formato específico. Não tem código HTML.

Com essa resposta, a aplicação *front-end* pode renderizar uma tabela de dados bonitinha para o usuário, ou fazer o que for mais conveniente, como por exemplo mostrar uma mensagem que não encontrou nenhum registro, se for o caso.

Tanto a criação da requisição quanto o tratamento da resposta e renderização no *front-end* são feitos por código JavaScript, por isso a grande importância desta linguagem.

2.3. Arquitetura tradicional versus moderna

A primeira coisa que precisamos esclarecer é que, ao dizer **arquitetura tradicional**, não estamos nos referindo a uma solução ruim, incorreta ou defasada. Muito pelo contrário, em muitos casos, ela ainda tem um grande valor.

Entendido isso, quais são as vantagens de cada modelo de arquitetura?

Construir uma aplicação com a **arquitetura tradicional** é mais fácil e dependendo do tipo do projeto, pode ser mais produtivo também.

Na **arquitetura moderna**, temos uma maior flexibilidade e, dependendo do framework usado no *front-end*, a produtividade pode ser imensa, principalmente para projetos comerciais e corporativos, onde temos muitas páginas de cadastros, pesquisas, *dashboards* e relatórios.

Tanto a arquitetura moderna quanto a tradicional nos permite criar aplicações profissionais e com muita qualidade, mas a arquitetura moderna nos permite ter

uma separação mais clara entre *back-end* e *front-end*, ou seja, uma parte não fica acoplada à outra.

É importante notar que, na arquitetura tradicional, nós também temos *back-end* e *front-end*, mas geralmente essas partes ficam dentro do mesmo projeto, com uma separação não tão clara e um acoplamento um pouco mais forte.

Voltando a falar sobre o desacoplamento de *back-end* e *front-end* na arquitetura moderna, nós podemos ter um mesmo *back-end* sendo utilizado por dois ou mais *front-ends*, por exemplo, uma interface web em Angular, outra interface web com outro framework JavaScript e uma terceira interface em Android.

Quando as aplicações *mobile* surgiram, muitas vezes, nós construímos um *back-end* específico para atender as necessidades da aplicação. Além de ter o *back-end* que atendia as requisições feitas a partir de um navegador, tínhamos também outro apenas para o aplicativo *mobile*.

Com a solidificação das aplicações para dispositivos móveis, a evolução dos frameworks JavaScript para *front-end* e a onda crescente da “internet das coisas”, não dá para ficar construindo um *back-end* para cada cliente que tiver que ser criado.

Outro ponto importante é o seguinte: da mesma forma que podemos ter vários clientes utilizando um mesmo *back-end*, é possível um único cliente (*front-end*) usando mais de um servidor de serviços (aplicações de *back-end*). Não vamos detalhar muito esse caso, por ser menos comum, mas estamos colocando ele aqui para que você entenda ainda melhor as possibilidades.

2.4. Single-Page Applications

Uma *Single-Page Application* (SPA) é uma aplicação web que tem apenas uma página HTML. Esse tipo de aplicação está altamente relacionada com a arquitetura que nos referimos como “arquitetura moderna”.

Basicamente, uma SPA depende bastante de código JavaScript, que executa toda a “mágica” por trás.

Ser uma aplicação web de apenas uma página HTML não necessariamente significa que ela deve apresentar só uma página/tela para o usuário.

O JavaScript implementado na aplicação é inteligente o suficiente para renderizar novas telas dinamicamente, fazer chamadas AJAX e atualizar a tela a medida que o usuário interage com a aplicação.

O uso de uma SPA gera uma experiência mais fluida para o usuário, no sentido de que o usuário não experimenta muito tempo de carregamento entre as telas do sistema e não tem qualquer necessidade de recarregar a aplicação ou a página para o usuário, na maioria das vezes.

Mas claro, isso requer muito código implementado no lado do cliente, ou seja, em JavaScript.

Felizmente, existem vários frameworks JavaScript para ajudar no desenvolvimento desse tipo de aplicação, como por exemplo o Angular (que vamos abordar neste livro).

Videoaula: Vantagens e desvantagens de Single-Page Applications

Assista um vídeo do nosso canal do YouTube, sobre as vantagens e desvantagens de uma Single-Page Application e quais tipos de aplicações se encaixam melhor nesse modelo.



https://www.youtube.com/watch?v=Y_PedfN_Pcs

Vantagens de SPA

A primeira vantagem desse modelo é que tudo que é necessário para a aplicação *front-end* rodar no navegador é carregado logo no início e somente uma vez, ou seja, o HTML, CSS e JavaScript é carregado no momento em que o usuário acessa a aplicação.

Isso pode ser considerado uma vantagem porque as interações do usuário são mais fluidas, já que a aplicação *front-end* permanece carregada no navegador.

A segunda vantagem é que, em uma SPA, trafegamos somente dados entre o *front-end* e *back-end*. O resultado é menos tráfego de rede e, portanto, o usuário espera menos pela apresentação da informação na tela.

Desvantagens de SPA

Se por um lado a gente tem a vantagem do carregamento dos arquivos HTML, CSS e JavaScript ser feito de uma vez só e no início, por outro lado isso pode ser um problema. Obviamente, carregar uma aplicação inteira de uma vez só consome um pouco mais de tempo na inicialização dela.

E existem alguns tipos de aplicações web que precisam de um primeiro carregamento muito rápido.

Por exemplo, em um e-commerce, que deve apresentar uma listagem de produtos a venda muito rapidamente, pode não ser interessante ter esse tempo maior de inicialização.

As tecnologias que suportam o desenvolvimento de SPAs fornecem mecanismos de implementar carregamento de módulos da aplicação por demanda (*lazy loading*), que já ajuda bastante, mas mesmo assim, ainda existe um carregamento inicial que consome um tempo extra.

Uma segunda desvantagem é a questão do SEO (*Search Engine Optimization*), ou seja, é mais difícil (mas não impossível) fazer com que sua página seja bem ranqueada pelos motores de busca, como o Google.

Para a maioria das *webapps*, isso não chega a ser um problema, porque geralmente SEO não é um requisito do projeto.

Quando usar e não usar SPA

O fato de termos a vantagem de precisar carregar os recursos estáticos somente na primeira vez que o sistema é acessado no navegador e também de trafegar somente dados, faz com que esse cenário seja perfeito para criação de sistemas em geral.

Uma regrinha simples que geralmente funciona é pensar da seguinte forma:

- Se o que queremos é uma *webapp*, então SPA pode ser um bom caminho
- Se queremos um *website*, então é melhor evitar SPA

Alguns exemplos de *webapps*:

- ERP
- Controle de estoque
- Sistema financeiro
- Sistema de cadastros
- Administração de um e-commerce

Alguns exemplos de *websites*:

- E-commerce
- Blog
- Site institucional

Capítulo 3

REST

A ideia que passamos até aqui é que existe uma arquitetura moderna, que traz mais flexibilidade e desacoplamento, e pode ser colocada em prática através de uma aplicação *front-end* separada de uma outra aplicação *back-end*.

Mas qual a melhor forma de fazer o *front-end* “conversar” com o *back-end*?

Aqui entra o REST!

Mas o que é REST?

REST, que vem de *Representational State Transfer* (Transferência de Estado Representacional), é um **modelo arquitetural**, ou seja, ele não é uma tecnologia e, portanto, não é possível baixá-lo em algum site.

Apesar de, aparentemente, ser uma proposta nova, REST surgiu no início dos anos 2000, a partir da tese de PhD de um cientista chamado Roy Fielding.

O intuito geral era a formalização de um conjunto de melhores práticas, denominadas *constraints* (restrições).

Essas *constraints* tinham como objetivo determinar a forma na qual padrões como HTTP e URI deveriam ser modelados, aproveitando, de fato, todas as características oferecidas pelos mesmos.

3.1. Por que usar REST?

Uma das primeiras vantagens que você vai perceber ao adotar REST é que ele deixa nossas **aplicações mais simples**, até pela organização que ele nos impõe. Essa organização está ligada às *constraints* que temos que atender. Falaremos sobre elas mais a frente.

Essa simplicidade acaba nos dando uma segunda vantagem, que é a de que nosso software pode **evoluir de forma incremental**, ou seja, evoluir sem impactos ao que já existe, sem “quebrar” o que já existe.

Destacamos também o fato de que esse modelo nos força a ter uma aplicação *back-end* com ausência de estado.

Como a aplicação *back-end* não tem estado, então **podemos escalar de maneira muito mais fácil**.

Se uma máquina para o servidor de serviços não está sendo suficiente, então podemos adicionar mais uma sem ter de ficar replicando a sessão (ou o estado) de uma máquina para outra. Qualquer máquina poderá atender qualquer requisição.

Além dessas vantagens, ainda existem algumas outras provenientes da **demandas de mercado**.

Empresas pequenas e grandes têm contratado cada vez mais sistemas de diferentes fornecedores. A questão é que chega um ponto em que é necessária a integração dessas informações. E REST ajuda muito nisso.

Se existe um modelo adotado por todos os sistemas, ou seja, se todos têm pontos de entrada e saída usando um mesmo modelo arquitetural, é relativamente mais simples fazer a integração.

Além disso, temos cada vez mais dispositivos (celulares, TVs, geladeiras, câmeras, etc) acessando diversos serviços na nuvem todos os dias, ou seja, estamos o tempo todo adicionando e recuperando novos dados em algum serviço na internet. Um modelo arquitetural simples e eficiente, como REST, ajuda a tornar isso possível.

Videoaula: Por que você deve aprender REST?

Neste vídeo você vai aprender o que é este modelo arquitetural e porquê isso faz tanto sentido nos dias de hoje.



<https://www.youtube.com/watch?v=P6GcoLU0iCo>

3.2. Melhores práticas na adoção de REST

Falamos que o REST teve o intuito de formalizar um conjunto de melhores práticas, que são as seis abaixo:

- Cliente-servidor
- Stateless
- Cache
- Interface uniforme
- Sistema em camadas
- Código sob demanda

Antes de discorrer sobre cada uma, é importante notar que seu sistema não precisa colocar todas as restrições em prática, mas ele precisa estar preparado para cada uma.

Exemplo: talvez sua regra de negócio não permita fazer cache, porque os dados estão em constante mudança, mas tecnicamente e arquiteturalmente, seu

software deve permitir a aplicação do cache sem que sejam necessárias alterações naquilo que já existe.

Cliente-servidor

A restrição **cliente-servidor** é autoexplicativa. Precisamos de um cliente (que em nosso caso é uma aplicação *front-end* rodando no navegador) enviando requisições para um servidor (uma aplicação web como *back-end*).

Stateless

Stateless significa “sem estado”, e essa *constraint* diz que a aplicação não deve possuir estado.

Na prática, isso quer dizer que a requisição feita ao servidor de serviços deve conter todo o necessário para que seja devidamente processada, e também que não podemos manter uma sessão no servidor.

Cache

Sobre a restrição de **cache**, temos que nossa aplicação deve permitir cache em qualquer ponto, sem que haja alterações em nossas regras existentes.

Interface uniforme

Por **interface uniforme** você pode entender como um conjunto de operações bem definidas do seu sistema.

Para ficar mais claro, você pode fazer a analogia com o *CRUD*, ou seja, seria padronizar a forma como as operações “criar”, “pesquisar”, “atualizar” e “deletar” são feitas. Lembrando que foi uma analogia. Interface uniforme não se resume ao *CRUD*.

Sistema em camadas

A quinta restrição é relacionada a ter um **sistema em camadas**. Entre o cliente e o servidor deve ser possível adicionar elementos intermediários, sem que haja impacto tanto no cliente quanto no servidor.

Isso é útil para adicionarmos funções de monitoramento, log, dentre outras coisas, sem ter que conhecer a aplicação por dentro.

Código sob demanda

Por último, a restrição de **código sob demanda** diz que a parte servidora da aplicação (o *back-end*) deve prover ao cliente uma forma de ele se adaptar às novas funcionalidades, sem que isso tenha um impacto no código.

Um exemplo dessa restrição seriam os arquivos JavaScripts que os servidores entregam aos seus clientes (os navegadores, no caso). Se o arquivo JavaScript for alterado, o navegador não precisa ficar sabendo e não sofre com o impacto. É só carregar, rodar e pronto.

3.3. É REST ou RESTful?

Se você chegou a estudar um pouco sobre REST, então já deve ter se perguntando sobre a diferença entre os termos REST e RESTful, não é?

Não se preocupe! Uma confusão muito comum é o uso intercambiável desses termos. Mas de fato, uma API ou aplicação deve receber o nome REST ou RESTful?

De forma bem direta, o correto é você dizer RESTful API (ou API RESTful, para “brasileirar”). Além disso, é importante você entender o porquê dessa nomenclatura e qual o momento certo de usar cada uma.

Quando estamos discutindo sobre o modelo e sobre as características que nós vimos anteriormente, você deve utilizar o termo REST. Já no momento em que

você estiver falando de uma implementação que usa essas mesmas características, você deve usar RESTful.

Apesar de não ser algo tão relevante, achamos interessante dizer isso, para que você sempre utilize as nomenclaturas corretas.

Isso também nos ajuda a deixar claro que REST nada mais é que um conjunto de boas práticas.

3.4. Protocolo HTTP

Como foi dito, o modelo arquitetural REST independe de tecnologia, mas podemos dizer mais: Ele não depende de tecnologia e nem de protocolos, como por exemplo o HTTP.

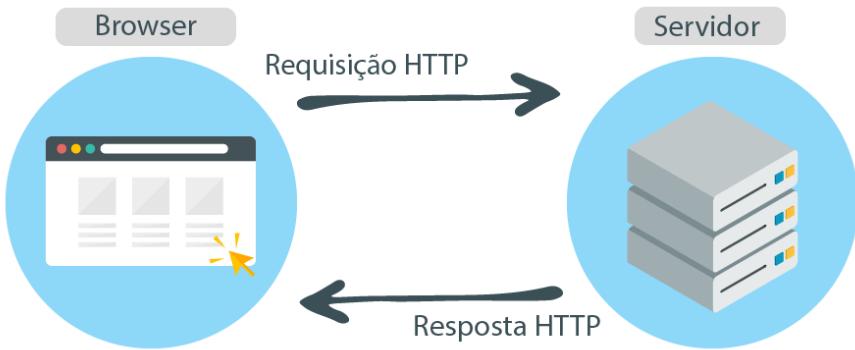
Mas por que falar de HTTP então?

Apesar do modelo ser independente de protocolo, para colocar REST em prática, é preciso de um. E o mais comum é o protocolo HTTP.

Por isso vamos descrever as características do HTTP, para que você entenda como atender as restrições utilizando esse protocolo.

O objetivo não é listar todas as funcionalidades possíveis, mas navegaremos pelas características principais que você deve conhecer para implementar uma API RESTful.

A primeira coisa sobre ele é o fato de ser um protocolo **requisição-resposta**.



Esse primeiro ponto nos ajuda a atender a restrição **cliente-servidor**.

Agora, é importante irmos mais a fundo e entender a composição de uma requisição e a de uma resposta.

Primeiro vamos entender os elementos de uma **requisição**, veja:

[MÉTODO] [URI] HTTP/[VERSÃO]
 [CABEÇALHOS]

[CORPO/PAYLOAD]

Para formar uma requisição, nós precisamos de:

- Um método
- Da URI
- Versão do protocolo HTTP
- Cabeçalhos
- Corpo

Sobre o método e a URI, nós comentaremos em um próximo tópico.

Quanto a **versão** do HTTP, apesar de já estarmos na versão 2.0, o mais comum ainda é a versão 1.1.

Usar a versão 2.0 vai depender do servidor web utilizado e também do cliente (geralmente o navegador). A boa notícia é que a gente não precisa se preocupar com isso agora, pois é algo que os navegadores tratam automaticamente.

Os **cabeçalhos** são informações sobre a requisição. Temos vários possíveis e, inclusive, podemos criar os nossos. Alguns exemplos de cabeçalhos comuns em uma requisição são:

- *User-Agent*: informações sobre quem está fazendo a requisição
- *Accept*: qual o tipo de conteúdo é esperado/aceito na resposta
- *Content-Type*: qual o tipo do conteúdo está indo no corpo da requisição

Vale notar que, durante o desenvolvimento, os frameworks abstraem um pouco o protocolo HTTP, então nosso contato com o protocolo não é tão baixo nível assim, como estamos explicando aqui.

Com relação ao **corpo** de uma requisição, ele não é obrigatório e depende muito do método HTTP utilizado. Basicamente, é no corpo da requisição que enviamos os dados do cliente para o servidor.

Agora veja o exemplo de uma requisição configurada:

```
POST /produtos HTTP/1.1
Accept: application/json

{
  "nome": "Notebook i7",
  "preco": 2100.00
}
```

Quando uma requisição é feita para o servidor, o cliente espera por uma resposta. Uma resposta também tem alguns elementos importantes. Veja:

HTTP/[VERSÃO] [STATUS]
[CABEÇALHOS]

[CORPO]

Os elementos de uma resposta são:

- Versão do protocolo HTTP
- Status
- Cabeçalhos
- Corpo

Sobre a versão, é a mesma coisa que já comentamos um pouco antes, ao falar da requisição.

O cabeçalho e o corpo também seguem a mesma ideia já explicada. Daremos então uma atenção para o status.

O **status**, como o próprio nome indica, serve para descrever qual foi o resultado do processamento da requisição.

O servidor web deve retornar um status adequado para cada situação. Veja abaixo um resumo das categorias de códigos de retorno (status) suportados pelo HTTP.

- 1xx - Informações
- 2xx - Sucesso
- 3xx - Redirecionamento
- 4xx - Erro no cliente
- 5xx - Erro no servidor

Para cada uma das categorias acima, existem códigos específicos que podem ser aplicados nas diversas situações que podemos ter ao tentar processar uma requisição.

Repare abaixo como seria uma resposta adequada para uma solicitação de inclusão de produto, considerando que ela foi atendida com sucesso pelo *backend*.

```
HTTP/1.1 201 Created
Location: /produtos/331
```

```
{
  "codigo": 331,
```

```
"nome": "Notebook i7",
"preco": 2100.00
}
```

Note que o status 201 é retornado na resposta, que representa que o recurso foi criado com sucesso.

Os status mais comuns são:

- **200**: Padrão para requisições executadas com sucesso
- **201**: A requisição foi atendida, criando um novo recurso com sucesso
- **204**: O servidor processou a requisição com sucesso e não está retornando nenhum conteúdo
- **301**: Esta requisição e todas as futuras devem ser redirecionadas para uma URI fornecida no cabeçalho da resposta
- **403**: A requisição é válida, mas o servidor se recusou a executá-la, geralmente por não estar autorizado
- **404**: O recurso solicitado não existe
- **500**: Um erro inesperado aconteceu no servidor

3.5. URI e recursos

Nas aplicações que criamos, nós precisamos sempre fazer representações de coisas abstratas ou concretas do mundo real.

Em um sistema de estoque, por exemplo, você teria que representar o produto, o galpão, a compra, a venda, etc.

Geralmente chamamos essas coisas de entidades, mais especificamente, de entidades de banco de dados. No REST chamamos essas mesmas coisas de **recursos**.

Um recurso no REST é representado por sua URI que, por sua vez, é especificada pela RFC 3986 detalhadamente.

Basicamente, temos a modelagem de um recurso sob um substantivo, ao invés de verbos, como é usualmente utilizado em APIs. Veja alguns exemplos:

Lista de clientes

<http://estoque.algaworks.com/clientes>

Cliente de código 23

<http://estoque.algaworks.com/clientes/23>

Notificações do cliente de código 23

<http://estoque.algaworks.com/clientes/23/notificacoes>

Lista de produtos

<http://estoque.algaworks.com/produtos>

Produto de código 98

<http://estoque.algaworks.com/produtos/98>

Uma URI deve ser visualizada como um recurso, e não como uma ação a ser executada sob um servidor.

Alguns autores e literaturas defendem a utilização de verbos em casos específicos, porém isso não é totalmente aceitável e você deve usar com cautela.

Além de ter uma URI que o identifica, um recurso tem também uma **representação**. Veja abaixo uma possível representação para o recurso cliente:

```
{  
    "id": 23,  
    "nome": "João da Silva",  
    "nascimento": "23/06/1912",  
    "profissao": "Físico",  
    "endereco": {  
        "cidade": "Uberlândia",  
        "pais": "Brasil"  
    }  
}
```

Essa poderia ser a representação retornada no corpo de uma requisição para a URI: <http://estoque.algaworks.com/clientes/23>

Outra coisa a se notar é que podemos ter vários tipos de representação para um recurso. Por exemplo, pode ser em formato XML, JSON, HTML, etc.

No exemplo acima, o recurso está representado com JSON (*JavaScript Object Notation*), que é o mais utilizado atualmente em APIs RESTful e, geralmente, tomado como padrão pelos frameworks para desenvolvimento *front-end* e *backend*.

Temos que esclarecer também que um recurso não necessariamente precisa estar ligado a alguma parte concreta de um sistema, como por exemplo uma tabela de banco de dados.

Videoaula: Boas práticas para uma API RESTful

Como você anda mapeando sua a API RESTful? Nesse vídeo você vai conferir uma série de boas práticas para a hora de criar seus Web Services RESTful.



https://www.youtube.com/watch?v=P-juXKmJy_g

3.6. Métodos

A RFC 7231 especifica um conjunto de 8 métodos, os quais podemos utilizar para a criação de uma API RESTful. Esse conjunto de métodos possui a semântica de operações possíveis de serem efetuadas sob um determinado recurso.

Dentre esses 8 métodos, abaixo segue um detalhamento dos 4 mais conhecidos:

- GET

- POST
- PUT
- DELETE

O **método GET** é utilizado quando existe a necessidade de se obter um recurso. Ele é considerado idempotente, ou seja, independente da quantidade de vezes que é executado sob um recurso, o resultado sempre será o mesmo. Exemplo:

GET /clientes/23 **HTTP/1.1**

Essa chamada irá retornar uma representação para o recurso /clientes/23. Repare que não precisamos ter um corpo quando utilizamos o método GET.

Para a inclusão de um recurso no sistema a partir de uma representação, usamos o **método POST**. Exemplo:

POST /clientes **HTTP/1.1**

```
{  
  "nome": "João da Silva"  
}
```

O **método PUT** é utilizado como forma de atualizar um determinado recurso. Em alguns cenários muito específicos, ele também pode ser utilizado como forma de criação, por exemplo quando o cliente tem a liberdade de decidir a URI a ser utilizada.

PUT /clientes/23 **HTTP/1.1**

```
{  
  "nome": "João da Silva Saulo"  
}
```

Quanto ao **método DELETE**, ele tem como finalidade a remoção de um determinado recurso. Exemplo:

DELETE /clientes/23 **HTTP/1.1**

Apesar de aparentemente os métodos disponibilizarem uma interface CRUD (*Create, Read, Update e Delete*) para manipulação de recursos, alguns autores não concordam muito com esse ponto de vista e defendem a ideia que esses métodos

podem possuir também uma outra representação semântica um pouco mais abstrata.

Capítulo 4

Ecossistema Spring

O Spring não é um framework apenas, mas um conjunto de projetos que resolvem vários problemas do dia a dia de um programador, ajudando a criar aplicações Java com simplicidade e flexibilidade.

Inclusive, é bem comum fazer referência para esse conjunto de projetos como **ecossistema Spring**.

Existem muitos projetos interessantes no ecossistema Spring, como o Spring MVC, para atender a requisições HTTP, o Spring Security, para prover segurança, e diversos outros projetos que vão de persistência de dados até *cloud computing*.

O Spring surgiu como uma alternativa ao Java EE, e seus criadores sempre se preocuparam para que ele fosse o mais simples e leve possível.

Desde a sua primeira liberação, em Outubro de 2002, o Spring tem evoluído muito, com diversos projetos maduros, seguros e robustos para utilizarmos em produção.

Os projetos Spring são *open source*, e não só por isso, mas também pela qualidade, são amplamente utilizados no mercado de desenvolvimento de software.

Ao longo deste capítulo, vamos conhecer os projetos essenciais para desenvolvermos o *back-end* de uma aplicação web, ou seja, uma API RESTful que pode ser consumida por diferentes clientes.

4.1. Spring Framework

É fácil confundir todo o ecossistema Spring com apenas o Spring Framework, mas o Spring Framework é apenas um, dentre o conjunto de projetos que o Spring possui.

Ele é o projeto do Spring que serve de base para todos os outros. Talvez por isso haja essa pequena confusão.

O Spring Framework foi pensado para que nossas aplicações pudessem ter foco maior nas regras de negócios e menos na infraestrutura.

Dentre suas principais funcionalidades, podemos destacar:

- Spring MVC
- Suporte para JDBC e JPA
- Injeção de dependências (*Dependency injection*)

O **Spring MVC** é um framework para criação de aplicações web, incluindo APIs RESTful. Ele ajuda no desenvolvimento de aplicações web robustas, flexíveis e com uma clara separação de responsabilidades no tratamento da requisição. Através dele são disponibilizados vários recursos que são de grande ajuda na tratativa de uma requisição HTTP.

Videoaula: Conheça o Spring MVC

Quer entender melhor o que o Spring MVC pode fazer por você? Nesse vídeo você vai descobrir isso e muito mais!



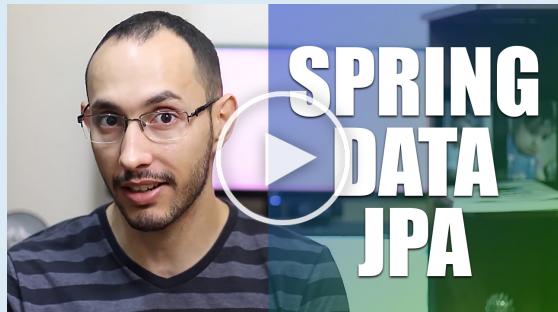
<https://www.youtube.com/watch?v=8duNNsRhD9k>

Com o mesmo pensamento do Spring MVC, temos o **suporte para JDBC e JPA**. Isso porque persistência também é um recurso muito utilizado nas aplicações. É difícil conceber uma aplicação hoje que não armazene dados em algum banco de dados.

Abrindo um parênteses, esse suporte para JPA é hoje o que podemos considerar de suporte básico, porque para mais produtividade no uso do JPA, podemos usar o Spring Data JPA, que é um projeto a parte e também está dentro do ecossistema.

Videoaula: Produtividade com Spring Data JPA

Quer aprender a criar sua camada de persistência com muito mais produtividade? Então veja esse vídeo!



<https://www.youtube.com/watch?v=wSiVxOsJ7es>

Por último, assim como o Spring Framework é a base do ecossistema, a **injeção de dependências** é a base do Spring Framework. Imagino que os projetos Spring seriam quase que um caos, caso esse conceito não tivesse sido implementado.

4.2. Spring Security

Como o próprio nome diz, esse projeto trata da segurança que, por sua vez, se dá ao nível de aplicação. Ele entrega um suporte excelente para autenticação e autorização.

Spring Security torna bem simples a parte de autenticação. Com algumas poucas configurações, já podemos ter uma autenticação via banco de dados, LDAP, etc. Sem falar nas várias integrações que ele suporta e na possibilidade de criar as suas próprias.

Quanto a autorização, ele é bem flexível também. Através das permissões que atribuímos aos usuários autenticados, podemos proteger as requisições web, a simples invocação de um método e até a instância de um objeto.

Além disso, ele nos deixa protegidos de diversos ataques conhecidos, como o *Session Fixation* e o *Cross Site Request Forgery*.

Videoaula: Proteja suas aplicações Java com Spring Security

Você quer deixar suas aplicações web Java seguras, de forma profissional? Então conheça mais sobre o Spring Security neste vídeo.



<https://www.youtube.com/watch?v=ptcjehUbz8>

4.3. Spring Boot

Enquanto os componentes do Spring são simples de escrever, as configurações podem ser extremamente complexas e cheia de códigos XML.

Por isso mesmo, a partir da versão 3.0, a configuração pode ser feita de forma programática, através de código Java.

Essa configuração programática trouxe alguns benefícios, como o de evitar erros de digitação, já que a classe de configuração precisa ser compilada, mas nós ainda precisamos escrever muito código explicitamente.

Com toda essa configuração excessiva, o desenvolvedor perde o foco na coisa mais importante: desenvolver as regras de negócio da aplicação.

O Spring Boot veio para retirar essa pedra de nossos sapatos. Ele trouxe agilidade e nos possibilita focar nas funcionalidades da nossa aplicação com o mínimo de configuração, porque ele utiliza o conceito de *Convention over Configuration* (*CoC*).

Basicamente, o Spring Boot já configura quase tudo pra gente, adotando uma visão opinativa sobre criação de aplicações Spring prontas para produção.

É claro que é possível customizar as configurações, mas o Spring Boot segue o padrão que a maioria das aplicações precisa, então muitas vezes não é necessário fazer coisa alguma.

Vale destacar que, toda essa “mágica” que o Spring Boot traz para o desenvolvimento Spring não é realizada com geração de código. Não, o Spring Boot não gera código! Ele simplesmente analisa o projeto e, automaticamente, configura pra gente.

Se você já trabalha com o Maven, sabe que precisamos adicionar várias dependências no arquivo `pom.xml`, que pode ficar bastante extenso, com centenas de linhas de configuração. Com o Spring Boot podemos reduzir muito isso, com os *starters* que ele fornece.

Os *starters* são dependências que agrupam outras dependências, assim, se você precisa trabalhar com JPA e Hibernate, por exemplo, basta adicionar uma única entrada no `pom.xml`, e todas as dependências necessárias serão adicionadas ao *classpath*.

Videoaula: Spring Boot vs Spring MVC

Você está se perguntando qual é melhor? Então é **obrigatório** que você assista esse vídeo.



<https://www.youtube.com/watch?v=fwyYZ9tVJsY>

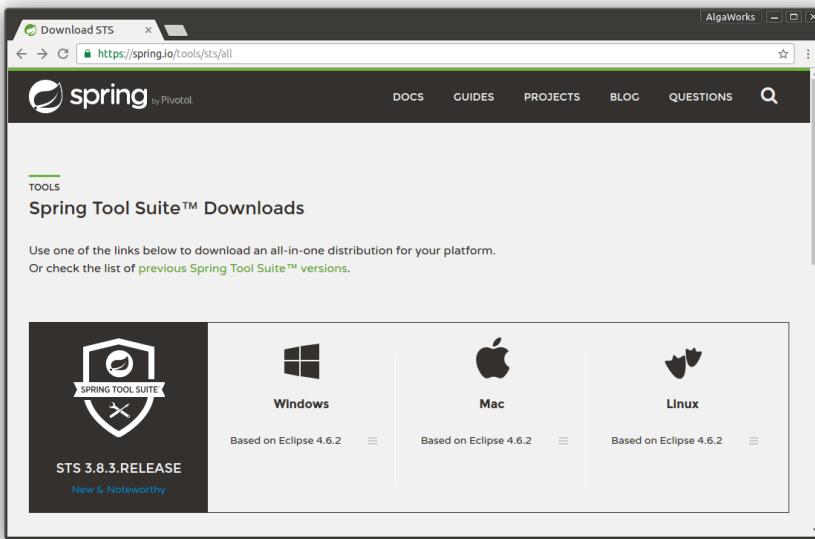
4.4. Spring Tool Suite - STS

O Spring Tool Suite, ou STS, é a IDE que está crescendo muito entre os desenvolvedores Spring.

O STS é um Eclipse com plugins e funcionalidades extras para quem cria projetos usando o ecossistema Spring, em especial o Spring Boot.

Ele tem um assistente que nos ajuda a rapidamente criar um projeto Maven com todas as dependências necessárias para tirarmos proveito do Spring Boot.

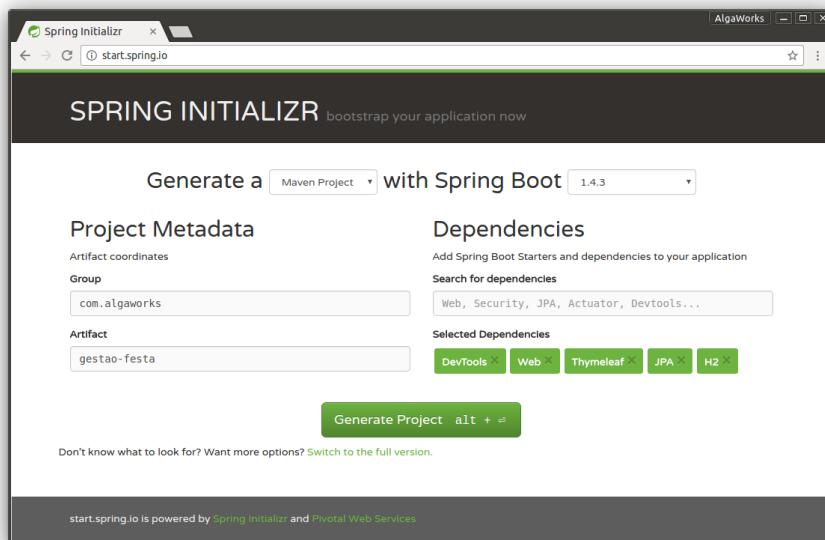
Existem versões para Windows, Mac e Linux em <https://spring.io/tools/sts/all>.



A instalação é como a do Eclipse, basta baixar e descompactar.

Embora o STS seja uma ferramenta excelente, você não precisa ficar preso a ela. Você pode desenvolver projetos Spring com Eclipse, NetBeans, IntelliJ IDEA, etc.

Se você já tem uma certa experiência e gosta de outra IDE, existe uma alternativa para ajudar na criação dos seus projetos que usam o Spring Boot também. Basta acessar o Spring Initializr online em <http://start.spring.io>.



Nesse site você consegue quase a mesma facilidade que temos para criar projetos usando o STS. Nele, você informa os dados do projeto, frameworks e bibliotecas que deseja ter como dependência, e então um projeto Maven será gerado para ser importado na sua IDE preferida.

Capítulo 5

Angular

Angular é uma plataforma e framework para construção da interface de aplicações usando HTML, CSS e, principalmente, JavaScript, criada pelos desenvolvedores da Google.

Ele possui alguns elementos básicos que tornam essa construção interessante.

Dentre os principais, podemos destacar os componentes, templates, diretivas, roteamento, módulos, serviços, injeção de dependências e ferramentas de infraestrutura que automatizam tarefas, como a de executar os testes unitários de uma aplicação.

Angular nos ajuda a criar *Single-Page Applications* com uma qualidade e produtividade surpreendente!

Alguns outros pontos dessa plataforma que merecem destaque são o fato de que ela é *open source*, possui uma grande comunidade, existem várias empresas utilizando e tem muito material de estudo para quem deseja se aperfeiçoar.

5.1. Angular vs AngularJS

É importante deixar claro que, neste livro, nós estamos falando de Angular, ou Angular 2+. Não estamos falando de AngularJS 1.x.

Caso você seja novo no mundo Angular, deve estar se perguntando: Mas qual é a diferença? É só a versão?

Na verdade, são tecnologias completamente diferentes!

AngularJS é um framework JavaScript para desenvolvimento web. Foi o framework queridinho no mercado por alguns anos.

Mas para acompanhar a evolução da tecnologia, os desenvolvedores perceberam que seria melhor criar um novo framework do zero, usando toda a experiência que tiveram com o AngularJS e necessidades dos desenvolvedores.

E então, surgiu o Angular 2, uma verdadeira plataforma para desenvolvimento de aplicações não só web, mas também mobile, com mudanças significativas na sua estrutura.

Obviamente, uma aplicação desenvolvida com AngularJS não é compatível com Angular.

E claro, essa grande mudança deixou alguns desenvolvedores preocupados. Ninguém gosta de saber que a tecnologia que está usando vai ser substituída por uma nova, né?

Embora o projeto do AngularJS continue sendo mantido, essa mudança fez com que ele perdesse força no mercado, dando lugar para o novo Angular.

E não seria nenhuma novidade pra gente se passasse por sua cabeça: “E se eu começar a usar o novo Angular e acontecer isso outra vez, de recriarem o framework?”.

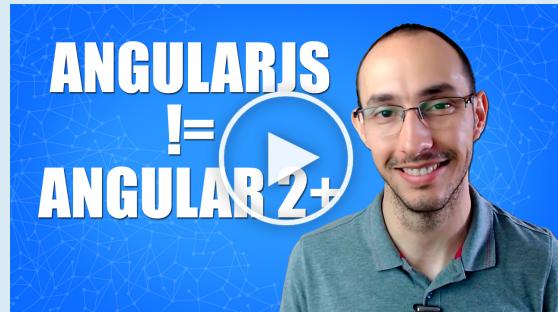
Embora essa possibilidade exista, na nossa opinião ela é muito pequena. E como desenvolvedores de software, nós estamos sempre suscetíveis a isso com **qualquer tecnologia**. E por um lado, achamos isso muito bom!

Se eles precisaram recriar o framework para entregar pra gente uma nova ferramenta muito melhor, porque isso não seria bom?

Uma das únicas certezas que temos nessa área de desenvolvimento de software é que, o que usamos hoje não será mais usado daqui alguns anos, pelo menos não da mesma forma. Por isso mesmo precisamos estar em constante aprimoramento, lendo livros e fazendo cursos, como você está fazendo agora.

Videoaula: Não confunda AngularJS com Angular!

Você ainda acha que AngularJS é a mesma coisa que Angular? Então você precisa ver esse vídeo agora!



<https://www.youtube.com/watch?v=HOgTIsfp2kI>

5.2. O versionamento do Angular

A partir do Angular 2, a tecnologia começou a mudar suas versões de uma forma diferente. Foi lançado o Angular 4, Angular 5, Angular 6 e dependendo da época que você está lendo esse livro, talvez Angular 7 e assim por diante.

Quem não entende o que está acontecendo, pode achar que os desenvolvedores da plataforma ficaram loucos e estão recriando a tecnologia como se não houvesse amanhã. Mas não é nada disso!

A partir do Angular 2, a tecnologia começou a usar *Semantic Versioning (SEMVER)*.

Basicamente, o *SEMVER* divide o número da versão em 3 partes, sendo *Major.Minor.Patch*. Por exemplo, 5.2.10, 6.0.1, 6.0.2, etc.

Quando um bug é corrigido e liberado, o número do *patch* é incrementado. Se uma nova funcionalidade é adicionada, o número do *Minor* é incrementado, e se

uma mudança pode potencialmente quebrar código de quem usa a plataforma, o número do *Major* é incrementado.

Em outras palavras, isso significa que todas as mudanças que correm o risco de quebrar código de quem usa o framework, ou seja, que potencialmente pode precisar que o desenvolvedor faça algum ajuste para continuar com o projeto funcionando, devem ser lançadas com uma mudança do número *Major*.

Essas *breaking changes* não significam uma reescrita do framework. Não significa que o mundo acabou e que você vai precisar reaprender tudo novamente. São apenas mudanças que são inevitáveis e que podem quebrar código, como acontece em 100% das tecnologias que evoluem e não estão paradas no tempo.

E isso nem quer dizer que vai quebrar código com toda certeza. Quer dizer apenas que, potencialmente, pode ter sim quebra de código, e por isso necessita de uma atenção especial por parte do desenvolvedor. É até muito comum atualizarmos a versão do Angular e tudo continuar funcionando normalmente.

Então, você pode ficar totalmente tranquilo em relação a isso. Angular é uma tecnologia que evolui muito rápido e você, como um bom profissional de tecnologia, deve achar isso excelente!

E para encerrar essa questão sobre as versões, olha só uma curiosidade: você não vai encontrar a versão 3 do Angular. Isso porque, simplesmente, ela não existiu. Por uma série de conflitos em suas bibliotecas, resolveu-se pular essa versão e ir direto do Angular 2 para o Angular 4.

5.3. Como está o mercado?

O mercado absorveu o Angular 2+ muito rápido. Pequenas e grandes empresas no Brasil já adotaram a tecnologia, mas infelizmente a quantidade de profissionais qualificados ainda é muito pequena e insuficiente para a grande demanda que existe.

A forma mais fácil de comprovar isso é fazendo uma rápida pesquisa em sites de empregos. Você vai notar que existem muitas vagas com salários de R\$3.000,00 a R\$10.000,00 (e até mais), procurando desesperadamente por profissionais.

The screenshot shows three job listings for Java and Angular developers:

- Consultor Java Angular** (Sexta, 11/05)
De R\$ 9.001,00 a R\$ 10.000,00
1 vaga: São Paulo - SP (1)
Desenvolvimento de sistemas usando Java, PHP e C, Linguagens de Programação: Java, PHP, C/C++. Banco de Dados: Oracle, PostgreSQL, MySQL, SQL Server [continuar lendo...](#)
[enviar currículo](#) 7 dias grátis
- Arquiteto Java - Angular / REST** (Quinta, 05/04)
De R\$ 7.001,00 a R\$ 8.000,00
1 vaga: Curitiba - PR (1)
Responsabilidade: liderar tecnicamente a equipe, analisar documento de requisitos, criar arquitetura integrada dos sistemas e soluções. Desenvolver [continuar lendo...](#)
[enviar currículo](#) 7 dias grátis
- Arquiteto Java - Angular / REST** (Quinta, 05/04)
De R\$ 7.001,00 a R\$ 8.000,00
1 vaga: Curitiba - PR (1)
Responsabilidade: liderar tecnicamente a equipe, analisar documento de requisitos, criar arquitetura integrada dos sistemas e soluções. Desenvolver [continuar lendo...](#)
[enviar currículo](#) 7 dias grátis

Muitas dessas vagas exigem também conhecimento de *back-end*, como REST, Java e Spring, ou seja, são vagas para desenvolvedores *fullstack*.

5.4. A linguagem TypeScript

Para criarmos aplicações com Angular, podemos usar as linguagens TypeScript, JavaScript e Dart.

O Angular foi desenvolvido em TypeScript, e essa linguagem é a mais usada para aplicações Angular e a que mais encontramos documentações.

Nos nossos cursos de Angular da AlgaWorks, também usamos TypeScript. Por essas e outras razões, nós recomendamos que você use essa linguagem também.

O TypeScript é uma linguagem de programação criada pela Microsoft. Ela é um *superset* do JavaScript, ou seja, faz tudo o que o JavaScript faz e ainda mais algumas coisas.

Uma das coisas legais ao escolher TypeScript é a possibilidade de utilizar os recursos mais recentes do JavaScript, que ainda não funcionam nos navegadores.

Os navegadores (browsers) não conseguem executar código TypeScript, por isso, o que acontece é que os desenvolvedores escrevem códigos em TypeScript e esses códigos são transpilados para JavaScript.

Transpilação é parecido com a compilação, mas ao invés de gerar um código de mais baixo nível, é gerado o código em outra linguagem, ou na mesma linguagem, só que com o objetivo de deixar compatível com o ambiente que vai ser executado.

No caso de aplicações Angular desenvolvidas com TypeScript, a transpilação é o mesmo que gerar código JavaScript a partir de código TypeScript.

A maioria das pessoas que começam com Angular não conhecem TypeScript ainda. A boa notícia é que o TypeScript é uma linguagem com uma documentação excelente e bem fácil de aprender. Melhor ainda para quem vem do Java, pois as duas linguagens são bem parecidas. Se você já é um programador Java, vai se sentir em casa.

Videoaula: O que é TypeScript?

Vamos dar uma olhada em como funciona o TypeScript? Então assista esse vídeo e aprenda mais sobre a linguagem, conceito de transpilação, como instalar o TypeScript e executar o seu primeiro script.

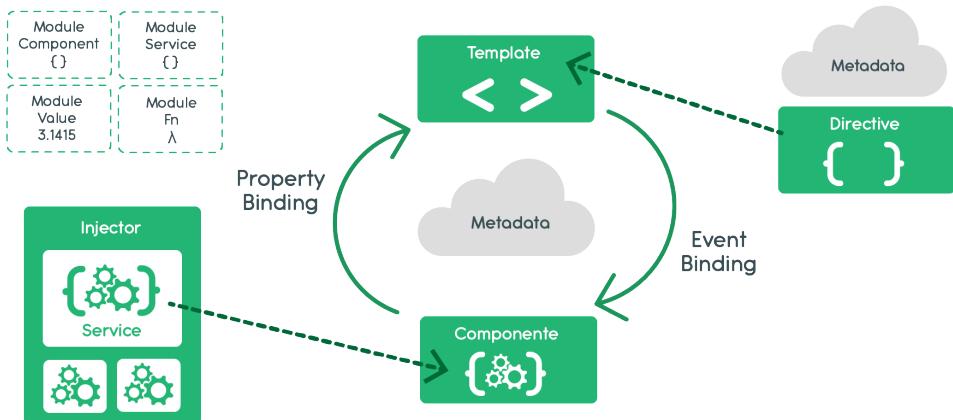


https://www.youtube.com/watch?v=HTS-Vi_pAXk

5.5. Elementos de uma aplicação Angular

Os elementos básicos (*building blocks*) de uma aplicação Angular são:

- Módulos
- Componentes
- Templates
- Metadata
- Data binding
- Diretivas
- Serviços
- Injeção de dependências



Não vamos entrar em detalhes sobre todos os elementos neste livro. Estudaremos apenas os principais, para você entender o poder do Angular.

Componentes

Uma aplicação Angular é baseada em **componentes**. Com eles, nós podemos encapsular comportamentos e regras da interface, o que torna a criação de aplicações algo mais simples. Inclusive, um componente pode encapsular outros componentes.

Isso é um ponto muito positivo, porque o componente pode ser reaproveitado em vários lugares da aplicação.

Um componente é composto de 3 itens:

- Template HTML
- CSS
- Uma classe que gerencia as propriedades e comportamentos

Poderíamos, por exemplo, criar uma barra de ferramentas (com botões “Salvar”, “Limpar”, “Excluir”, etc) e reutilizar ela em várias telas de cadastro.

Seria possível também criar um componente de listagem de dados, com paginação para exibir resultados de uma pesquisa qualquer, e reaproveitá-lo em todas as telas de pesquisa de um sistema.

Quem sabe, criar também uma entrada de texto que já venha com a possibilidade de incluir uma máscara (de telefone ou CPF, por exemplo).

Com esses exemplos, acho que agora você já entendeu que pode criar qualquer coisa que precisar reutilizar, né?

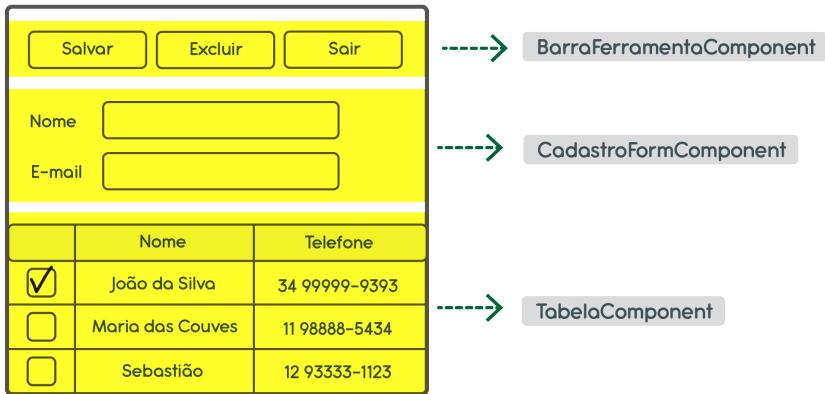
É importante notar que mesmo uma tela completa ou parte de uma tela do sistema que não seja possível ou necessário reaproveitar também será um componente.

Isso mesmo! Ao invés de criar apenas uma página, como talvez você já esteja acostumado, você vai criar um componente que representa aquela página.

Você pode criar um componente chamado `CadastroUsuarioComponent` e já colocar tudo o que precisa:

<input type="button" value="Salvar"/>	<input type="button" value="Excluir"/>	<input type="button" value="Sair"/>
Nome	<input type="text"/>	
E-mail	<input type="text"/>	
	Nome	Telefone
<input checked="" type="checkbox"/>	João da Silva	34 99999-9393
<input type="checkbox"/>	Maria das Couves	11 98888-5434
<input type="checkbox"/>	Sebastião	12 93333-1123

Você ainda poderia incluir um ou mais componentes dentro do componente que representa a sua tela, como uma barra de ferramentas, um componente de formulário e uma tabela para exibir a listagem dos registros.



Essa escolha sobre o que vai virar um componente ou não será sua e dependerá da necessidade da sua aplicação.

Caso esteja na dúvida sobre criar certos componentes, então não crie. Espere e, se identificar futuramente que será vantajosa a separação de alguma parte da tela em um componente isolado e reutilizável, então faça isso.

Com o tempo você vai se tornando mais ágil e conseguirá antecipar trechos reaproveitáveis na sua aplicação e que merecerão se tornar componentes.

Videoaula: Componentes avançados com Angular, passo a passo

Aprenda a criar componentes avançados com Angular, usando os decoradores `@Input` e `@Output`, para fazer binding de propriedades e eventos.

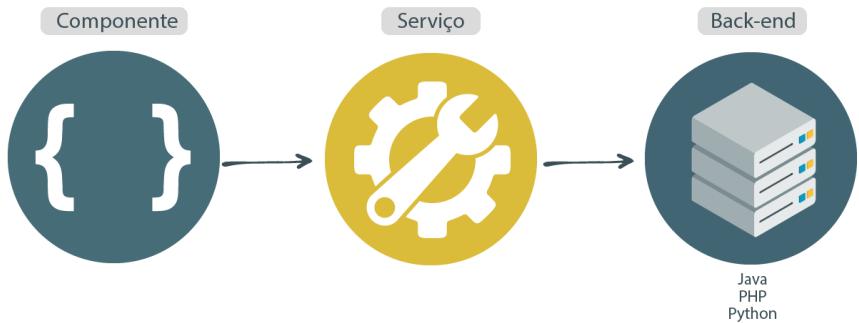


<https://www.youtube.com/watch?v=4x6ybpHQ5U0>

Serviços

Os **serviços** são criados para concentrar todas as regras de negócio da aplicação. Cada serviço pode ser usado por mais de um componente.

Eles não são restritos a isso, mas o que normalmente acontece é que usamos serviços para encapsular o código responsável por regras com um propósito específico, como por exemplo para recuperar ou submeter dados em uma API RESTful (*back-end*).



Com isso, podemos deixar os componentes focados em gerenciar partes da tela que o usuário visualizará e os serviços focados em código com um outro propósito específico.

Videoaula: Criando um projeto Angular e chamando API RESTful

Você quer começar a criar projetos com Angular, que fazem integração com serviços REST? Nessa aula você vai aprender isso, passo a passo.

<https://www.youtube.com/watch?v=UNkqooT1x8E>

Injeção de dependências

Injeção de dependências é um padrão de projeto que pode ser aplicado independentemente da linguagem. Esse padrão permite deixar as classes de

componentes mais limpas e eficientes, delegando tarefas complexas e regras de negócios para os serviços.

Indo mais a fundo, injeção de dependências é o processo de prover as instâncias necessárias que uma determinada classe precisa para ser instanciada e utilizada. A grande vantagem é que temos um menor acoplamento entre nossas classes.

O Angular implementa um mecanismo de injeção de dependências e executa ele na inicialização da aplicação.

Módulos

Outro elemento importante são os **módulos**. Eles ajudam a tornar nossa aplicação mais organizada.

É possível criar módulos que agrupem componentes, serviços e outros elementos por alguma semelhança.

5.6. Bibliotecas de componentes

Apesar de Angular ser um framework relativamente novo, já temos várias bibliotecas de componentes excelentes e gratuitas a nossa disposição.

A própria filosofia do framework, de ser baseada em componentes, cria um ambiente propício a isso.

Atualmente, as principais bibliotecas de componentes para Angular são:

- PrimeNG
- Angular Material
- ng-bootstrap

E nós sabemos que nesse momento você está perguntando: Qual é a melhor biblioteca?

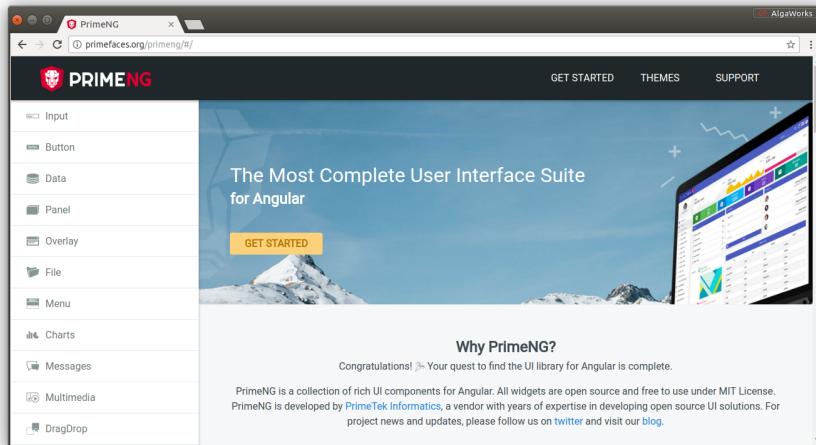
A melhor biblioteca é a que resolve o seu problema. Não existe uma resposta 100% correta para isso.

O recomendado é que você entre nos sites de cada biblioteca, conheça os componentes disponíveis, a facilidade de uso, comunidade, documentação, etc, e analise se o que existe disponível atende os seus requisitos.

Isso será importante para você tomar uma decisão na hora de escolher qual delas irá usar em seus projetos profissionais.

PrimeNG

O **PrimeNG** é a biblioteca com o maior número de componentes e a que evolui com maior velocidade. Ela tem uma qualidade incrível e é excelente para desenvolver projetos comerciais e corporativos.



<http://primefaces.org/primeng>

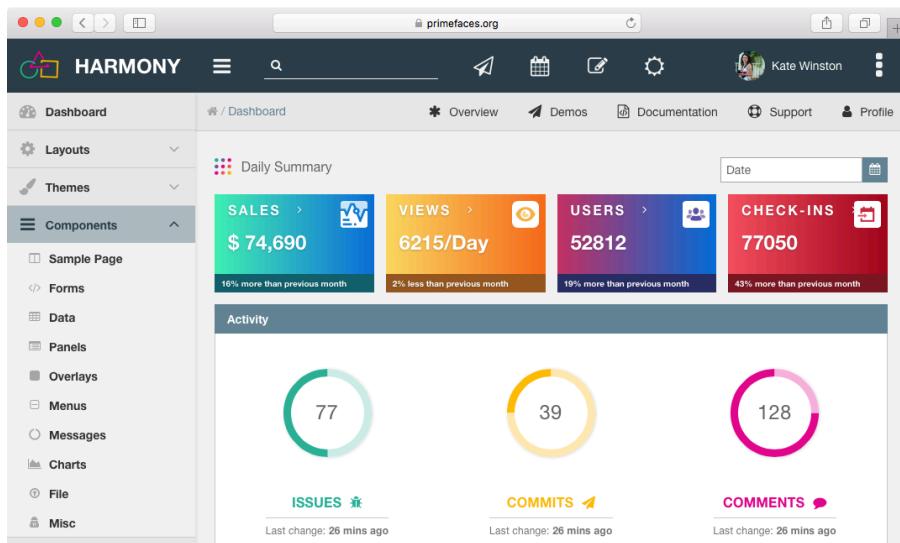
A biblioteca é desenvolvida e mantida pela PrimeTek, uma empresa com sede na Turquia.

A PrimeTek tem bastante experiência com desenvolvimento de bibliotecas de componentes. É a mesma empresa por trás do PrimeFaces (JSF), PrimeReact e PrimeUI.

As bibliotecas de componentes desenvolvidas pela PrimeTek são gratuitas e *open source*, mas você tem opção de adquirir serviços de suporte profissionais, caso ache interessante.

Além das bibliotecas, a PrimeTek também fornece temas gratuitos e comerciais (pagos), além de layouts comerciais prontos.

Ou seja, você pode mudar a “pele” da sua aplicação com uma infinidade de opções, e ainda pode adquirir layouts completamente prontos, com exemplos de *dashboards*, páginas de cadastro, página de login, etc.



É muito legal ter essas opções, para caso você não seja muito bom em deixar o *front-end* bonito e com uma boa experiência para o usuário, mas é algo totalmente opcional.

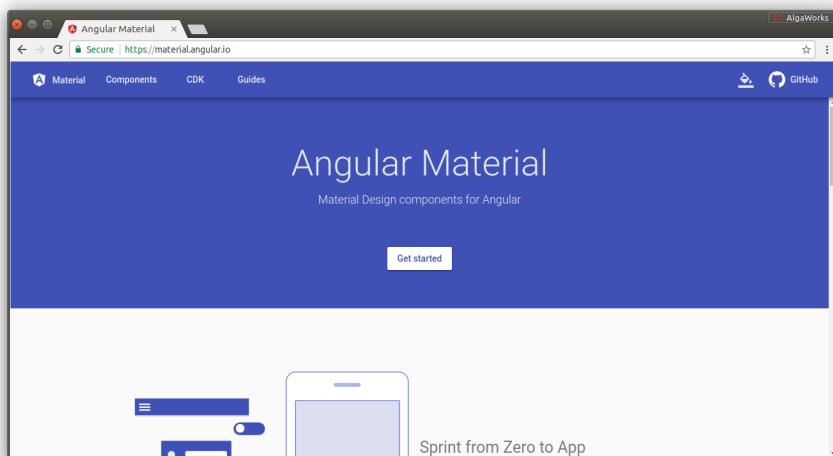
Você precisa acessar o site e conhecer tudo, porque somos um pouco suspeitos para falar, já que a AlgaWorks é parceira oficial da PrimeTek.



Eu (Thiago Faria) tomando "umas" com o Cagatay e equipe

Angular Material

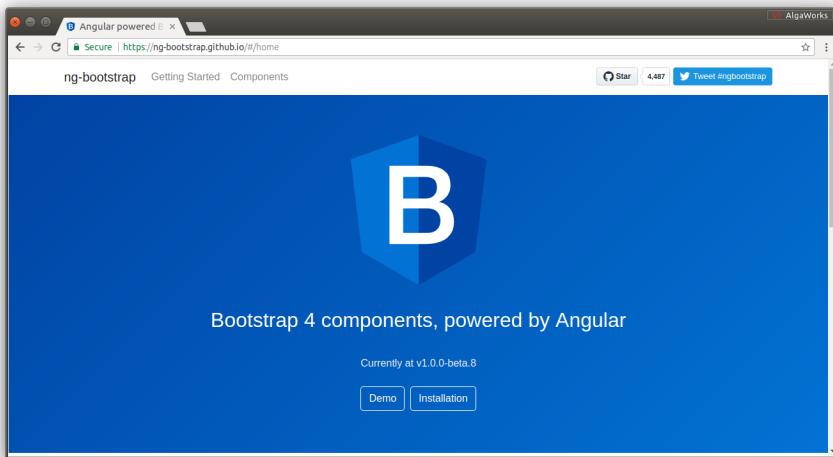
O **Angular Material** é uma biblioteca baseada no Material Design, criada pelo mesmo time de desenvolvedores do Angular. Já tem vários componentes bacanas e vem evoluindo muito.



<http://material.angular.io>

ng-bootstrap

O **ng-bootstrap** é uma biblioteca desenvolvida baseada no Bootstrap 4, que é um framework CSS bem conhecido para criação de interfaces web.



[http://ng-bootstrap.github.io/](https://ng-bootstrap.github.io/)

5.7. Ambiente de desenvolvimento

Para trabalhar com Angular, naturalmente, precisamos de um ambiente de desenvolvimento.

Um ambiente de desenvolvimento muito usado pela maioria dos desenvolvedores Angular é composto pelo NodeJS, Angular CLI e Visual Studio Code (também conhecido como VSCode).

O **NodeJS** é uma plataforma baseada na engine JavaScript do navegador Google Chrome. NodeJS não é utilizado diretamente pelas aplicações Angular, mas o ferramental para desenvolver aplicações Angular precisa dele para funcionar, ou seja, apenas em tempo de desenvolvimento.

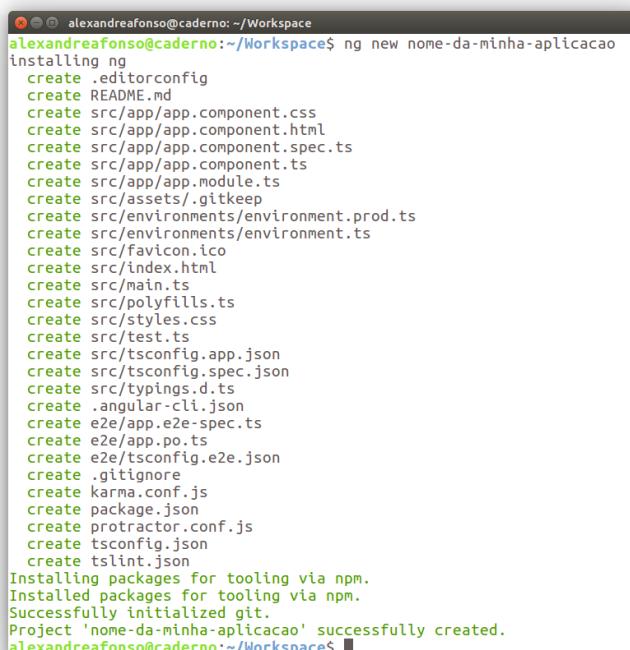
O **Angular CLI** é uma interface de linha de comando para ajudar na construção de aplicações com Angular. Você não é obrigado a usar essa ferramenta, mas ela facilita bastante a nossa vida.

Através dessa ferramenta, nós conseguimos tanto criar a estrutura para um novo projeto, quanto também gerar novos elementos em nossa aplicação, como por exemplo os componentes.

O Angular CLI fornece pra gente o comando `ng`, que usamos para trabalhar com nossa aplicação. Por exemplo, para criar um novo projeto, é só usar o comando `ng new`:

```
$ ng new nome-da-minha-aplicacao
```

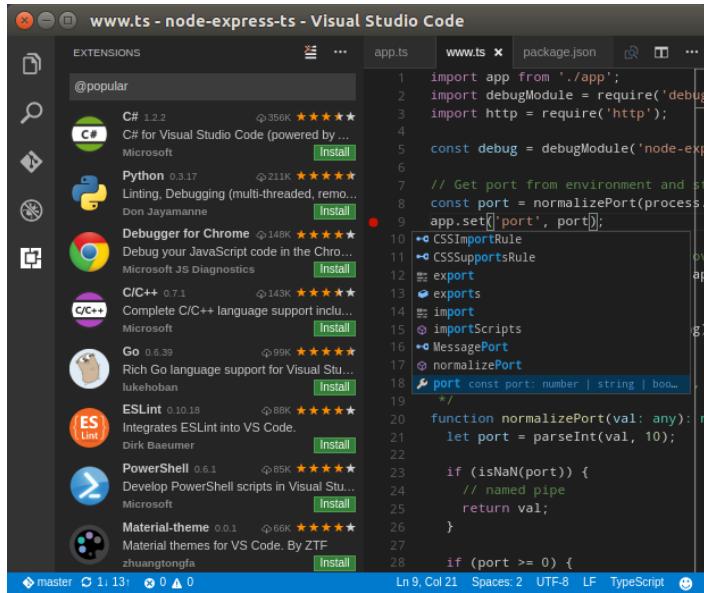
Esse comando cria toda a estrutura do projeto:



```
alexandreadefonso@caderno: ~/Workspace
alexandreadefonso@caderno:~/Workspace$ ng new nome-da-minha-aplicacao
installing ng
  create .editorconfig
  create README.md
  create src/app/app.component.css
  create src/app/app.component.html
  create src/app/app.component.spec.ts
  create src/app/app.component.ts
  create src/app/app.module.ts
  create src/assets/.gitkeep
  create src/environments/environment.prod.ts
  create src/environments/environment.ts
  create src/favicon.ico
  create src/index.html
  create src/main.ts
  create src/polyfills.ts
  create src/styles.css
  create src/test.ts
  create src/tsconfig.app.json
  create src/tsconfig.spec.json
  create src/typings.d.ts
  create .angular-cli.json
  create e2e/app.e2e-spec.ts
  create e2e/app.po.ts
  create e2e/tsconfig.e2e.json
  create .gitignore
  create karma.conf.js
  create package.json
  create protractor.conf.js
  create tsconfig.json
  create tslint.json
Installing packages for tooling via npm.
Installed packages for tooling via npm.
Successfully initialized git.
Project 'nome-da-minha-aplicacao' successfully created.
alexandreadefonso@caderno:~/Workspace$
```

O **VSCODE** é uma IDE criada pela Microsoft e muito popular para desenvolver projetos com Angular.

Ela é gratuita, muito leve e inclui várias funcionalidades necessárias para nos deixar mais produtivos durante o desenvolvimento.



The screenshot shows the Visual Studio Code interface. On the left, the Extensions sidebar is open, displaying a list of popular extensions with their names, versions, descriptions, and ratings. Some extensions have an 'Install' button. The main code editor window shows a file named 'app.ts' with the following TypeScript code:

```
import app from './app';
import debugModule = require('debug');
import http = require('http');

const debug = debugModule('node-express-ts');

// Get port from environment and store in Port
const port = normalizePort(process.env.PORT || '3001');
app.set('port', port);

// CSSImportRule
// CSSSupportsRule
// export
// exports
// import
// importScripts
// MessagePort
// normalizePort
// port const port: number | string | boolean;
normalizePort(val: any): number {
  let port = parseInt(val, 10);

  if (isNaN(port)) {
    // named pipe
    return val;
  }

  if (port >= 0) {
    return port;
  }

  throw new Error(`Port ${val} is not a valid port.`);
}
```

At the bottom of the interface, there are status indicators for master, 113 files, 0 errors, and 0 warnings.

Videoaula: Como instalar o Angular CLI?

Aprenda a instalar o Angular CLI nesta videoaula, passo a passo.



<https://www.youtube.com/watch?v=2vA4zfyLqh4>

Capítulo 6

Conclusão

Parabéns por ter chegado ao final da leitura! Estamos felizes por você ter cumprido mais essa etapa na sua carreira.

Nosso objetivo foi ajudar você a entender os fundamentos dessa arquitetura, que é amplamente utilizada nos dias de hoje, para que você possa guiar seu caminho e se tornar um **Desenvolvedor Fullstack Angular e Spring** de sucesso.

A ideia foi realmente abrir um pouco das cortinas e te mostrar o que acontece de fato por trás dos palcos deste vasto mundo do Angular e das APIs RESTful com Spring.

Se você gostou desse livro, por favor, ajude a manter esse trabalho. Recomende para seus amigos de trabalho, colegas da faculdade e compartilhe nas redes sociais.

6.1. Próximos passos

Agora que você já conhece os fundamentos de uma arquitetura moderna com Angular, REST e Spring, eu recomendo que você aprenda, na prática, como implementar um *back-end* com REST e Spring.

Uma excelente direção para esse próximo passo seria você participar do nosso curso online com videoaulas sobre esse assunto (REST e Spring).

Vamos continuar juntos? Acesse <http://alga.works/livro-fsas-cta>.

FULLSTACK ANGULAR & SPRING

Guia para se tornar um desenvolvedor moderno

Alexandre Afonso

Thiago Faria