



Angular 2

na prática

A faint, semi-transparent illustration of a city skyline with various buildings, a water tower, and two people standing on a ledge, all set against a blue sky with white clouds.

- + Node/npm
- + SystemJS
- + MongoDB
- + TypeScript
- + Visual Studio Code
- + Restfull APIs com express
- + Autenticação com Json Web Token

Angular 2 na prática (PT-BR)

Com Node/npm, Typescript, SystemJS e Visual Studio Code

Daniel Schmitz e Daniel Pedrinha Georgii

Esse livro está à venda em <http://leanpub.com/livro-angular2>

Essa versão foi publicada em 2016-02-29



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2016 Daniel Schmitz e Daniel Pedrinha Georgii

Conteúdo

1.	Introdução	2
1.1	Pré requisitos	2
1.1.1	Node	2
1.1.2	Servidor web	3
1.1.3	Arquivo package.json	3
1.1.4	Erros ao instalar o Angular 2	5
1.1.5	Arquivo package.json	6
1.1.6	Editores de texto e IDEs	8
1.2	Além do Javascript	9
1.3	TypeScript	10
1.4	Código fonte	10
2.	TypeScript	11
2.1	Instalando TypeScript	12
2.2	Uso do Visual Studio Code	13
2.2.1	Detectando alterações	25
2.2.2	Debug no Visual Studio Code	26
2.2.3	Debug no navegador	28
2.3	Tipos	29
2.3.1	Tipos Básicos	29
2.3.2	Arrays	30
2.3.3	Enum	30
2.3.4	Any	31
2.3.5	Void	31
2.4	Classes	31
2.4.1	Construtor	31
2.4.2	Visibilidade de métodos e propriedades	32
2.5	Herança	32

CONTEÚDO

2.6	Accessors (get/set)	34
2.7	Métodos Estáticos	34
2.8	Interfaces	35
2.9	Funções	36
2.9.1	Valor padrão	36
2.9.2	Valor opcional	36
2.10	Parâmetros Rest	36
2.11	Parâmetros no formato JSON	37
2.12	Módulos	38
2.12.1	Exemplo com Systemjs	38
2.12.2	Omitindo arquivos js e map no VSCode	41
2.12.3	Uso do SystemJS	42
2.13	Decorators (ou annotation)	46
2.14	Conclusão	48
3.	Um pouco de prática	49
3.1	Projeto AngularBase	49
3.1.1	Configurando o projeto	49
3.1.2	Erros ao instalar o Angular 2	49
3.1.3	Configurando a compilação do TypeScript	50
3.1.4	Criando o primeiro componente Angular 2	52
3.1.5	Criando o bootstrap	53
3.1.6	Criando o arquivo html	53
3.2	Criando uma pequena playlist	55
3.2.1	Estrutura inicial dos arquivos	56
3.2.2	Criando um arquivo de configuração da aplicação	57
3.2.3	Adicionando bootstrap	59
3.2.4	Criando a classe Video	61
3.2.5	Criando uma lista simples de vídeos	62
3.2.6	Criando sub-componentes	63
3.2.7	Formatando o template	65
3.2.8	Repassando valores entre componentes	67
3.2.9	Selecionando um vídeo	70
3.2.10	Eventos	70
3.2.11	Propagando eventos	72
3.2.12	Exibindo os detalhes do vídeo	74
3.2.13	Editando os dados do video selecionado	78
3.2.14	Editando o título	81

CONTEÚDO

3.2.15	Criando um novo item	84
3.2.16	Algumas considerações	85
3.3	Criando Componentes	86
3.4	Componentes Hierárquicos	93
4.	Um pouco de teoria	96
4.1	Visão Geral	96
4.2	Módulo (module)	97
4.2.1	Library Module	98
4.3	Componente (component)	99
4.4	Template	100
4.4.1	Interpolation (Uso de {{ }})	101
4.4.2	Template Expressions	101
4.5	Property Bind	101
4.5.1	Laços	102
4.5.2	Pipes (Operador)	103
4.6	Metadata (annotation)	103
4.7	Serviço (Service)	104
4.8	Injeção de dependência	104
4.8.1	Uso do @Injectable()	106
5.	Formulários	108
5.1	Criando o projeto inicial	108
5.2	Uso do ngControl	114
5.3	Exibindo uma mensagem de erro	116
5.4	Desabilitando o botão de submit do formulário	117
5.5	Submit do formulário	118
5.6	Controlando a visibilidade do formulário	119
6.	Conexão com o servidor	122
6.1	Criando o projeto	122
6.2	Uso da classe Http	123
6.3	Utilizando services	128
6.4	Organização do projeto	128
6.5	Model user	130
6.6	Service user	131
6.7	Alterando o componente AppComponent	132
6.8	Enviando dados	133

CONTEÚDO

7. Routes	135
7.1 Aplicação AngularRoutes	135
7.2 Dividindo a aplicação em partes	137
7.3 Criando a área onde os componentes serão carregados	138
7.4 Configurando o router	138
7.5 Criando links para as rotas	139
7.6 Repassando parâmetros	140
8. Exemplo Final - Servidor	142
8.1 Criando o servidor RESTful	142
8.2 O banco de dados MongoDB	142
8.3 Criando o projeto	146
8.4 Estrutura do projeto	147
8.5 Configurando os modelos do MondoDB	148
8.6 Configurando o servidor Express	149
8.7 Testando o servidor	158
8.8 Testando a api sem o Angular	159
9. Exemplo Final - Cliente	165
9.1 Arquivos iniciais	165
9.2 Preparando o Template base da aplicação	170
9.3 Implementando o roteamento (Router)	172
9.3.1 Criando componentes	172
9.3.2 Configurando o @RouteConfig	174
9.3.3 Configurando o menu	175
9.3.4 Configurando o router-outlet	176
9.4 Exibindo Posts	176
9.5 Login	180
9.6 Services	183
9.6.1 LoginService	183
9.6.2 UserService	184
9.6.3 HeadersService	185
9.7 Conectando no servidor	186
9.8 Posts	189
9.8.1 PostService	191
9.9 Refatorando a tela inicial	192
9.10 Conclusão	196
10. Utilizando Sublime Text	197

CONTEÚDO

10.1	Instalação	197
10.2	Adicionando suporte a linguagem TypeScript	197
10.3	Automatizando a build TypeScript	199
11.	Publicando a aplicação em um servidor cloud	201
11.1	Criando a conta na Digital Ocean	201
11.2	Criando o droplet (servidor)	202
11.3	Configurando o acesso SSH	204
11.4	Criando o usuário	206
11.5	Instalando o git	208
11.6	Instalando Node	209
11.7	Instalando o nginx	209
11.8	Instalando os módulos do node	211
11.9	Recompilando os arquivos TypeScript	211
11.10	Teste inicial	212
11.11	Integração entre nginx e node	212
11.12	Algumas considerações sobre node+nginx	216
11.13	Domínio	216
11.14	Conclusão	218

Uma nota sobre PIRATARIA

Esta obra não é gratuita e não deve ser publicada em sites de domínio público como o *scrib.* Por favor, contribua para que o autor invista cada vez mais em conteúdo de qualidade **na língua portuguesa**, o que é muito escasso. Publicar um ebook sempre foi um risco quanto a pirataria, pois é muito fácil distribuir o arquivo pdf.

Se você obteve esta obra sem comprá-la no site <https://leanpub.com/livro-angular2>, pedimos que leia o ebook e, se acreditar que o livro mereça, compre-o e ajude o autor a publicar cada vez mais. Obrigado!!

1. Introdução

Esta obra tem como objetivo apresentar o framework Angular, em sua nova versão, na qual foi totalmente reescrita. Quase todos os conceitos da versão 1 ficaram obsoletos e novas técnicas estão sendo utilizadas no Angular 2, com o objetivo de prover um framework mais dinâmico e moderno.

Neste contexto, existem duas situações em que o leitor se encontra. A primeira é definida para aqueles que já conhecem e usam o *Angular*, desde a sua versão 1.0, e que estão buscando uma forma de continuarem o desenvolvimento de seus programas neste framework. A segunda é definida por aqueles que não conhecem o angular, e estão começando agora o seu estudo.

Para aqueles que nunca trabalharam com o framework *Angular*, aprender a versão 2 logo no início será uma tarefa simples, já que conceitos antigos não irão “perturbar” os novos conceitos. Para quem conhece *Angular 1*, é preciso de um pouco de paciência para entender as extensas mudanças no framework. O ideal é que o desenvolvedor pense no Angular 2 como um novo framework, porque quase nenhum dos conceitos da versão anterior são aproveitados.

1.1 Pré requisitos

Antes de abordarmos o Angular 2 é necessário rever algumas tecnologias que são vitais para o desenvolvimento de qualquer biblioteca utilizando HTML/Javascript. Primeiro, usaremos extensivamente o **Node**, que é uma forma de executar Javascript no servidor. O uso do Node será vital para que possamos manipular bibliotecas em nosso projeto, que serão instaladas através do gerenciador de pacotes do Node chamado **npm**.

1.1.1 Node

Se você utiliza Windows, instale o node através do <http://www.nodejs.org>¹. Faça o download e instale o Node e deixe selecionado a opção npm, que é o gerenciador de pacotes do node. Além do node, é útil instalar também o GIT, disponível em <https://git-scm.com/>².

No Linux, podemos usar o gerenciador de pacotes apt - get para instalar tudo que precisamos, bastando apenas executar o seguinte comando:

¹<http://www.nodejs.org>

²<https://git-scm.com/>

```
$ sudo apt-get install git npm
```

Após instalar todos os pacotes necessários, é necessário criar um link simbólico para a palavra node, conforme o comando a seguir:

```
$ sudo ln -s /usr/bin/nodejs /usr/bin/node
```

1.1.2 Servidor web

Para o servidor web, iremos utilizar um pequeno pacote chamado `live-server`. Este servidor, escrito em node, atualiza automaticamente a página web quando há uma modificação no código fonte, o que ajuda no aprendizado. Para instalar o `live-server`, abra um terminal e digite o seguinte comando:

```
$ npm install live-server -g
```

Neste comando, o parâmetro `-g` indica que o pacote será instalado globalmente ao sistema. Desta forma, pode-se utilizá-lo em qualquer parte do sistema de arquivos.



Dicas para Windows

Para abrir um terminal no Windows, vá em *Iniciar, Executar* e digite `cmd`. Também pode-se clicar com o botão direito do mouse no Desktop e escolher a opção “Git Bash Here”, caso tenha instalado o git.



Dicas para Linux

No Linux, para instalar o pacote globalmente, use o comando `sudo`, por exemplo:
`sudo npm install live-server -g`

1.1.3 Arquivo package.json

Em todo projeto criado iremos definir um conjunto de regras de instalação e execução que estarão armazenadas no arquivo `package.json`. Este é um arquivo de configuração padrão dos projetos em Node.

Como teste, crie o diretório “test” e escreva o seguinte comando no terminal:

```
$ npm init
```

Uma série de perguntas serão feitas sobre o projeto. Pode-se, neste momento, deixar o valor padrão e pressionar `enter` até o processo terminar. Após o término, um novo arquivo será criado no diretório, com o seguinte texto.

```
{
  "name": "test",
  "version": "0.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Vamos adicionar a biblioteca `angular2` a este projeto. Se não estivéssemos utilizando `npm`, teríamos que acessar o site [angular.io³](http://angular.io) e realizar o download do arquivo zip da biblioteca. Como estamos utilizando o `npm`, a instalação será feita através do seguinte comando:

```
$ npm i angular2 --S
```

Neste comando, o parâmetro `i` é um acrônimo para `install`. Já o `--S` é um acrônimo para `--save` que tem como objetivo adicionar a biblioteca instalada no arquivo `package.json`. A resposta para este comando é semelhante a imagem a seguir.

³angular.io

```
x - □ daniel@debian: ~/test
daniel@debian:~/test$ npm i angular2 -S
npm WARN package.json test@0.0.1 No repository field.
npm WARN package.json test@0.0.1 No README data
npm WARN engine rxjs@5.0.0-beta.0: wanted: {"npm":">>=2.0.0"} (current: {"node":"0.10.25","npm":"1.4.21"})
es6-promise@3.0.2 node_modules/es6-promise

zone.js@0.5.10 node_modules/zone.js

es6-shim@0.33.13 node_modules/es6-shim

reflect-metadata@0.1.2 node_modules/react-metadata

rxjs@5.0.0-beta.0 node_modules/rxjs

angular2@2.0.0-beta.0 node_modules/angular2
daniel@debian:~/test$ █
```

1.1.4 Erros ao instalar o Angular 2

Na versão 3 do npm, alguns erros podem acontecer na instalação do Angular 2, como no exemplo a seguir:

```
C:\Users\daniel\test (test@1.0.0)
└─ npm i angular2 systemjs --S
  test@1.0.0 C:\Users\daniel\test
    ├─ angular2@2.0.0-beta.7
    │  └── UNMET PEER DEPENDENCY es6-promise@^3.0.2
    ├── UNMET PEER DEPENDENCY es6-shim@^0.33.3
    ├── UNMET PEER DEPENDENCY reflect-metadata@0.1.2
    ├── UNMET PEER DEPENDENCY rxjs@5.0.0-beta.2
    └── systemjs@0.19.22
      ├─ es6-module-loader@0.17.11
      └── when@3.7.7
        └── UNMET PEER DEPENDENCY zone.js@0.5.15

npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of es6-promise@^3.0.2 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of es6-shim@^0.33.3 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of reflect-metadata@0.1.2 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of rxjs@5.0.0-beta.2 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of zone.js@0.5.15 but none was installed.
npm WARN EPACKAGEJSON test@1.0.0 No description
npm WARN EPACKAGEJSON test@1.0.0 No repository field.

C:\Users\daniel\test (test@1.0.0)
└─ |
```

Até que a instalação de dependências pelo npm volte a funcionar corretamente, você terá que resolver este problema instalando manualmente as bibliotecas que resultaram em erro. Copie o nome da biblioteca seguido da sua versão e use o npm novamente para a instalação:

```
$ npm i es6-promise@^3.0.2 --S
$ npm i es6-shim@^0.33.3 --S
$ npm i reflect-metadata@0.1.2 --S
$ npm i rxjs@5.0.0-beta.2 --S
$ npm i zone.js@0.5.15 --S
```

Perceba que as versões de cada biblioteca podem mudar de acordo com a versão beta-x do Angular 2. Então não copie do livro os comandos de instalação, copie dos erros gerados pelo npm no seu sistema.

1.1.5 Arquivo package.json

Reveja o arquivo package.json e perceba que uma nova entrada chamada dependencies foi adicionada, exibido a seguir.

```
{  
  "name": "test",  
  "version": "0.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "angular2": "^2.0.0-beta.0",  
    "es6-promise": "^3.0.2",  
    "es6-shim": "^0.33.13",  
    "reflect-metadata": "^0.1.2",  
    "rxjs": "^5.0.0-beta.0",  
    "zone.js": "^0.5.10"  
  }  
}
```

Além da alteração no arquivo package.json, um novo diretório foi criado chamado node_modules, que contém as bibliotecas do projeto e suas dependências.

O arquivo package.json tem diversas outras funcionalidades. Por exemplo, na entrada scripts temos um item chamado test, que a princípio exibe uma mensagem de erro simples. Poderíamos adicionar uma nova entrada chamada start, e iniciarmos o servidor web a partir dela, conforme o exemplo a seguir:

```
{  
  "name": "test",  
  "version": "0.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "live-server --port=12345"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {}  
}
```

```
"dependencies": {  
    "angular2": "^2.0.0-beta.0",  
    "es6-promise": "^3.0.2",  
    "es6-shim": "^0.33.13",  
    "reflect-metadata": "^0.1.2",  
    "rxjs": "^5.0.0-beta.0",  
    "zone.js": "^0.5.10"  
}  
}
```

Em scripts adicionamos a entrada start com o comando `live-server --port=12345`. No terminal, basta usar o comando `npm start` para automaticamente executar o *live-server* com os parâmetros estabelecidos, conforme a imagem a seguir.



```
x - daniel@debian:~/test  
daniel@debian:~/test$ npm start  
> test@0.0.1 start /home/daniel/test  
> live-server --port=12345  
  
Serving "/home/daniel/test" at http://127.0.0.1:12345
```

Para saber mais sobre os tipos de scripts disponíveis no arquivo `package.json`, acesse [este link⁴](#).

1.1.6 Editores de texto e IDEs

Existem dezenas de editores de texto e IDEs no mercado. Dentre eles destacamos o [Sublime Text⁵](#), [Atom⁶](#) e [Visual Studio Code⁷](#). Através de testes realizados pelo autor, recomendamos que o Visual Studio Code seja utilizado, pois possui um suporte melhor a algumas funcionalidades que iremos utilizar no desenvolvimento, como uma boa compatibilidade com a linguagem TypeScript e acesso fácil ao Git e ao Debug.

⁴<https://docs.npmjs.com/misc/scripts>

⁵<http://www.sublimetext.com/>

⁶<http://atom.io>

⁷<https://code.visualstudio.com/>

1.2 Além do Javascript

Desde o anúncio do Angular 2 foi estabelecido grandes mudanças em relação ao Angular 1, inclusive em seu próprio nome. A biblioteca antes conhecida como *Angularjs* foi renomeada apenas para *Angular*. Além disso, o Angular que era mantido pelo *Google* recebeu suporte também da Microsoft, principalmente com a compatibilidade ao TypeScript.

O Angular 2 agora tem suporte as linguagens Typescript e Dart, na qual pode-se escolher de acordo com a sua preferência. Claro que a linguagem javascript pode também ser utilizada. Como estaremos desenvolvendo componentes no Angular 2, vamos exibir um pequeno exemplo de como é um componente no angular de acordo com as 3 linguagens disponíveis.

No caso do TypeScript, podemos ter a seguinte estrutura:

```
import {Component} from 'angular2/core'

@Component({
  selector: 'my-app',
  template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent {}
```

Este é um exemplo simples de componente no Angular 2. Mesmo sem entender muito o que está acontecendo, perceba que usamos o comando `import` para importar o Angular, e que depois usamos a notação `@Component` para adicionar características a classe `AppComponent`, criado através do comando `export class`.

Caso deseje escrever este componente utilizando Dart, o código fica:

```
import 'package:angular2/angular2.dart';
import 'package:angular2/bootstrap.dart';
@Component(selector: 'my-app', template: '<h1>My First Angular 2 App</h1>')
class AppComponent {}
main() {
  bootstrap(AppComponent);
}
```

Perceba algumas diferenças na forma de importar o Angular 2, na notação do componente e na criação da classe `AppComponent`. Agora vamos dar uma olhada em como seria a criação deste componente utilizando javascript puro:

```
(function(app) {  
    app.AppComponent = ng.core  
        .Component({  
            selector: 'my-app',  
            template: '<h1>My First Angular 2 App</h1>'  
        })  
        .Class({  
            constructor: function() {}  
        });  
})(window.app || (window.app = {}));
```

Em javascript não temos a criação da classe, nem do import, e a utilização de funções anônimas para a criação do componente.

1.3 TypeScript

Nesta obra estaremos utilizando formalmente o TypeScript, por ser uma linguagem tipada e com diversas funcionalidades. Basicamente, o TypeScript é uma linguagem com suporte ao ES6 (ECMAScript 2015, nova especificação do JavaScript), com suporte a *annotations* e suporte a tipos (definição de tipos em variáveis, parâmetros e retorno de métodos).

Estaremos abordando TypeScript no capítulo 2, para que possamos detalhar todas as suas funcionalidades básicas. É necessário conhecer o básico do TypeScript antes de utilizarmos o Angular 2.

1.4 Código fonte

Todo o código fonte desta obra está no GitHub, no seguinte endereço:

<https://github.com/danielschmitz/angular2-codigos>

2. TypeScript

Antes de iniciarmos o estudo sobre o Angular 2, é preciso abordar a linguagem TypeScript, que será extensivamente utilizada nesta obra. O TypeScript pode ser considerado uma linguagem de programação que contém as seguintes particularidades:

- Compatível com a nova especificação ECMAScript 2015 (ES6)
- Linguagem tipada
- Implementa Annotations
- Implementa Generics

Podemos afirmar que o TypeScript torna a linguagem JavaScript semelhante às linguagens derivadas do c, como php, c#, java etc.

Mesmo que TypeScript seja nova linguagem, ela não será executada no navegador. Após escrever o código TypeScript com alguma funcionalidade, é preciso converter este código para JavaScript puro. No exemplo a seguir, criamos uma classe chamada User, um método construtor e outro método chamado hello. Após criar a classe, instanciamos a variável user repassando os dois parâmetros do construtor e chamamos o método hello(). Veja:

```
class User {  
    fullname : string;  
    constructor(firstname:string,lastname:string) {  
        this.fullname = firstname + " " + lastname;  
    }  
    hello():string{  
        return "Hello, " + this.fullname;  
    }  
}  
var user = new User("Mary", "Jane");  
alert(user.hello());
```

Este código, quando traduzido para o JavaScript, torna-se algo semelhante ao código a seguir:

```
var User = (function () {
    function User(firstname, lastname) {
        this.fullname = firstname + " " + lastname;
    }
    User.prototype.hello = function () {
        return "Hello, " + this.fullname;
    };
    return User;
})();
var user = new User("Mary", "Jane");
alert(user.hello());
```

Escrever código em TypeScript te ajuda a criar programas mais estruturados, e a definição do tipo de dados ajuda a evitar bugs. Editores de texto podem exibir erros de sintaxe, como no exemplo a seguir com o editor *Visual Studio Code*:

```
1 class User {
2     fullname : string;
3     constructor(firstname:string,lastname:string) {
4         this.fullname = firstname + " " + lastname;
5     }
6     hello():string{
7         return "Hello, " + this.fullname;
8     }
9 }
10 var user = new User("Mary", 2);
11 alert(user.hello());
```

A screenshot of the Visual Studio Code interface showing a TypeScript file. At line 10, there is a syntax error: the argument '2' is passed to the 'User' constructor instead of a string. A red arrow points from the bottom right towards this error. A tooltip box appears over the error, containing the text: 'Argument of type 'number' is not assignable to parameter of type 'string''. The code uses standard ES6 syntax with some TypeScript annotations like the 'fullname' type declaration.

2.1 Instalando TypeScript

O código TypeScript precisa ser “traduzido” (chamamos este processo de *transpiler*) para javascript puro. Isso pode ser realizado de diversas formas, sendo a mais simples através da linha de comando. Primeiro, usamos o **npm** para instalar a biblioteca `typescript`, através do seguinte comando:

```
$ npm i typescript -g
```

Após a instalação do TypeScript globalmente, podemos utilizar o comando **tsc** (TypeScript Compiler) para compilar um arquivo com a extensão `ts` em um arquivo `js`. Por exemplo, crie o arquivo `user.ts` e adicione o seguinte código:

```
class User {  
    fullname : string;  
    constructor(firstname:string,lastname:string) {  
        this.fullname = firstname + " " + lastname;  
    }  
    hello():string{  
        return "Hello, " + this.fullname;  
    }  
}  
var user = new User("Mary", "Jane");  
alert(user.hello());
```

Após adicionar o código TypeScript no arquivo com a extensão .ts, execute o seguinte comando:

```
$ tsc user.ts
```

Após executar o comando tsc será gerado o arquivo user.js com o seguinte código javascript:

```
var User = (function () {  
    function User(firstname, lastname) {  
        this.fullname = firstname + " " + lastname;  
    }  
    User.prototype.hello = function () {  
        return "Hello, " + this.fullname;  
    };  
    return User;  
})();  
var user = new User("Mary", "Jane");  
alert(user.hello());
```

2.2 Uso do Visual Studio Code

Pode-se utilizar qualquer editor de textos para trabalhar com Angular 2. Nesta obra, utilizaremos extensivamente o Visual Studio Code pois é um dos editores gratuitos que possui

uma maior compatibilidade com TypeScript. Vamos criar um pequeno projeto chamado `HelloWorldVSCode` demonstrando como o uso do Visual Studio Code pode acelerar nosso desenvolvimento em Angular 2.

Como abordamos no capítulo anterior, o VSCode pode ser instalado [neste link¹](https://code.visualstudio.com/). O VSCode é um editor do tipo “folder based”, ou seja, quando ele abre uma pasta ele tenta ler naquela pasta algum arquivo de projeto (`package.json`, `project.json`, `tsconfig.json`) para adicionar as funcionalidades de acordo com projeto em questão.

Ou seja, antes de abrir o VSCode devemos, no mínimo, criar uma pasta e criar um arquivo de projeto. Isto é feito facilmente com `npm`, de acordo com a imagem a seguir:

¹<https://code.visualstudio.com/>

```
x - daniel@debian: ~/HelloWorldVsCode
daniel@debian:~$ mkdir HelloWorldVsCode
daniel@debian:~$ cd HelloWorldVsCode/
daniel@debian:~/HelloWorldVsCode$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

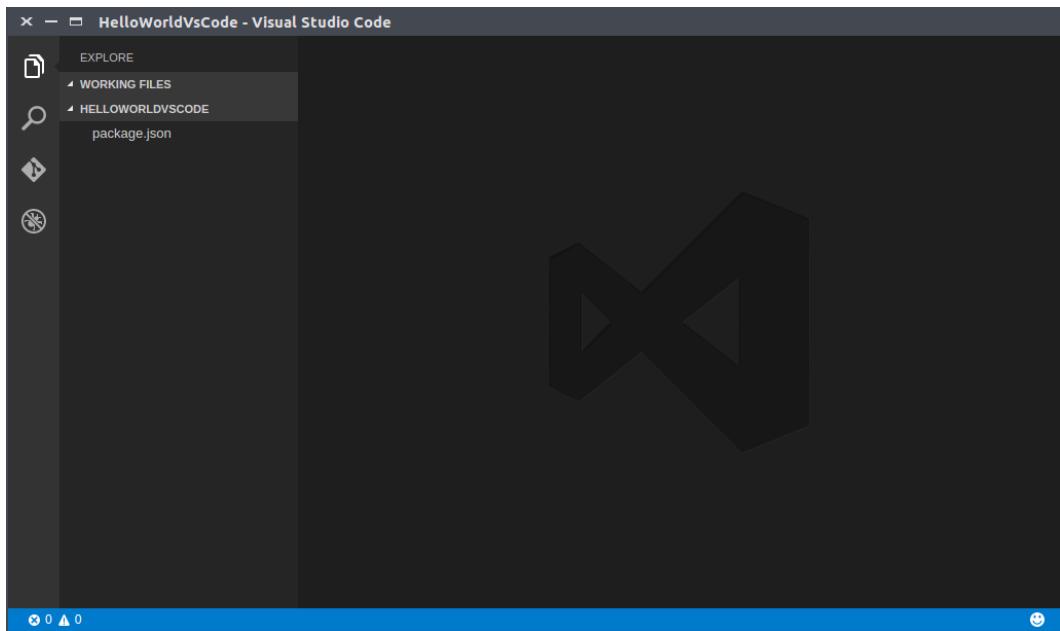
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (HelloWorldVsCode)
version: (0.0.0) 0.0.1
description: A simple hello world project
entry point: (index.js)
test command:
git repository:
keywords:
author: Daniel
license: (ISC)
About to write to /home/daniel>HelloWorldVsCode/package.json:

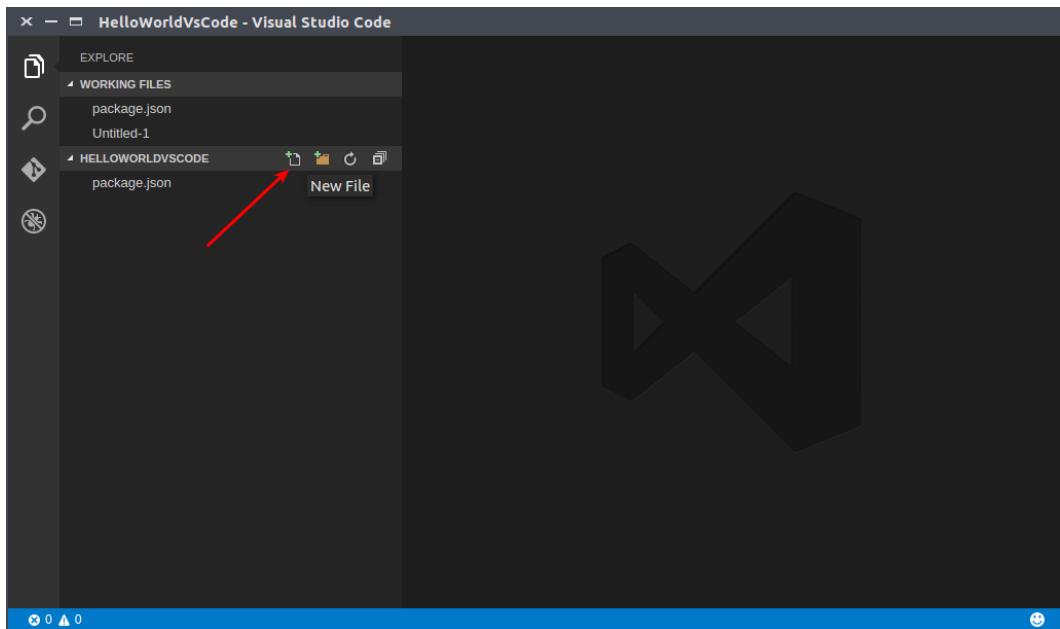
{
  "name": "HelloWorldVsCode",
  "version": "0.0.1",
  "description": "A simple hello world project",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Daniel",
  "license": "ISC"
}

Is this ok? (yes)
daniel@debian:~/HelloWorldVsCode$ █
```

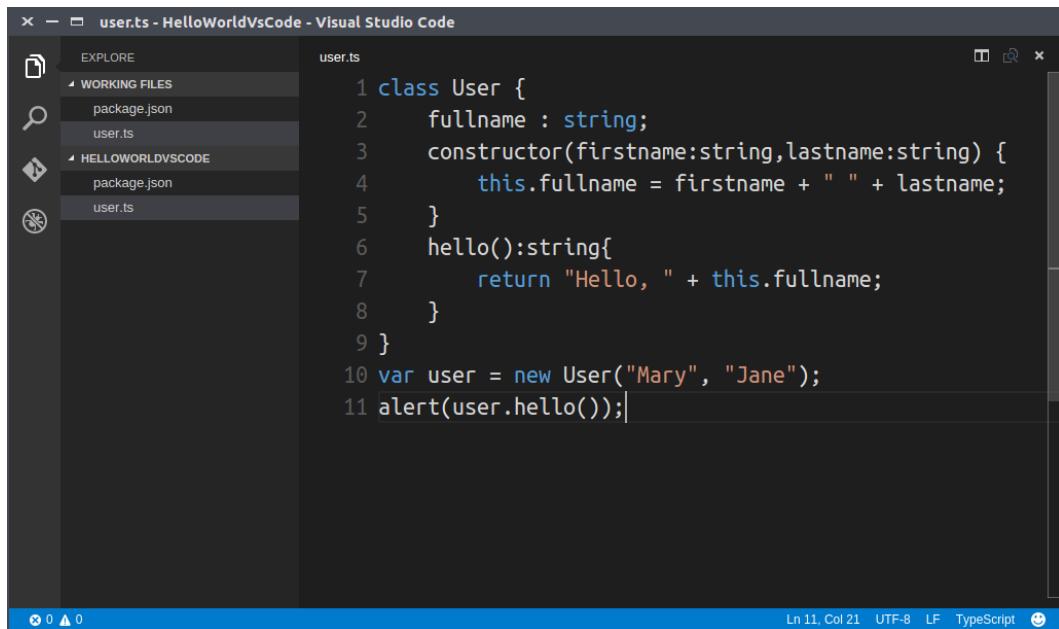
Após a criação do diretório, abra o Visual Studio Code, acesse `File > Open Folder`, e escolha o diretório recém criado. O VSCode fica semelhante a imagem a seguir:



Vamos novamente criar o arquivo `user.ts` dentro do diretório `HelloWorldVSCode`. Pelo VSCode, basta acessar `File > New File`, ou clicar no ícone em destaque da imagem a seguir:



Crie o arquivo `user.ts` com o mesmo código TypeScript anterior, conforme a imagem a seguir:

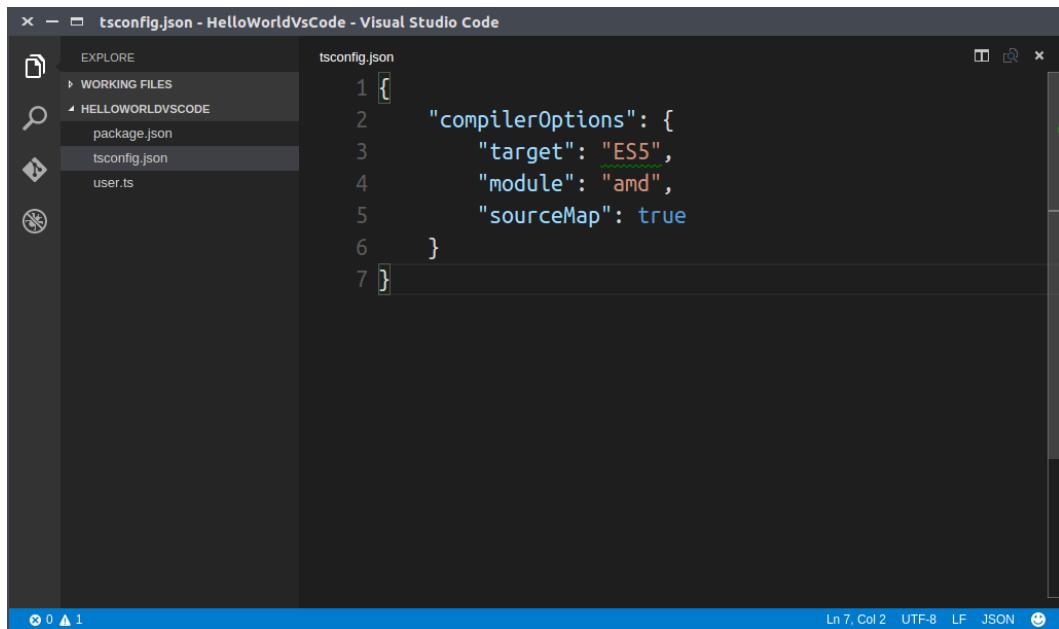


```
user.ts
1 class User {
2     fullname : string;
3     constructor(firstname:string,lastname:string) {
4         this.fullname = firstname + " " + lastname;
5     }
6     hello():string{
7         return "Hello, " + this.fullname;
8     }
9 }
10 var user = new User("Mary", "Jane");
11 alert(user.hello());
```

Precisamos agora usar o conversor do TypeScript para compilar o arquivo .ts em .js. Ao invés de usar a linha de comando, podemos configurar o próprio VSCode para isso. Primeiro, crie o arquivo tsconfig.json com o seguinte texto:

/HelloWorldVSCode/tsconfig.json

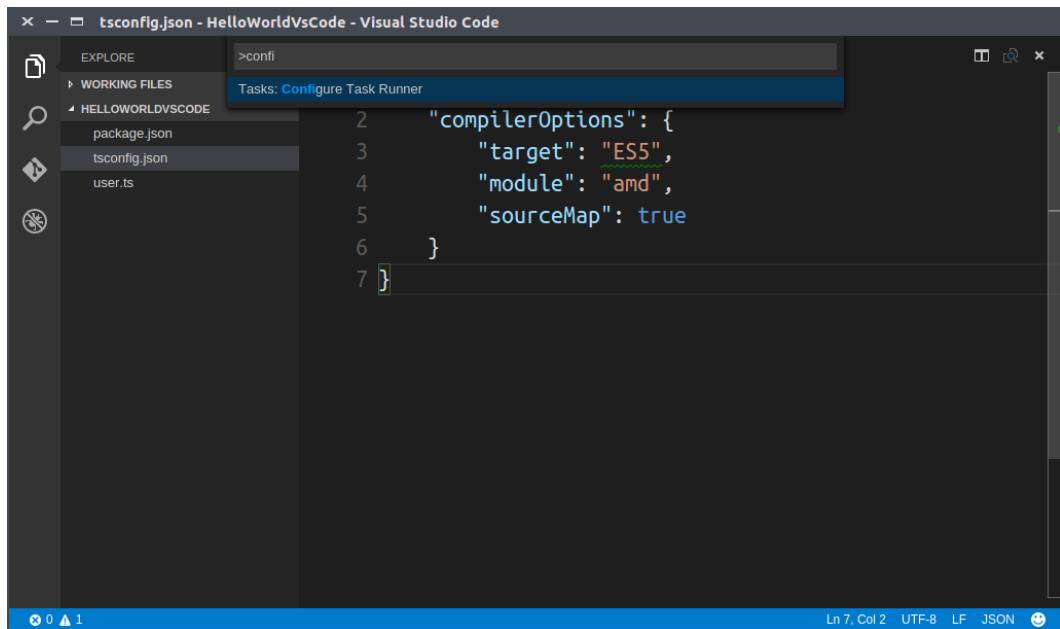
```
{
  "compilerOptions": {
    "target": "ES5",
    "module": "amd",
    "sourceMap": true
  }
}
```



```
tsconfig.json - HelloWorldVsCode - Visual Studio Code
EXPLORE          tsconfig.json
WORKING FILES
  HELLOWORLDVSCODE
    package.json
    tsconfig.json
    user.ts
1 [
2   "compilerOptions": {
3     "target": "ES5",
4     "module": "amd",
5     "sourceMap": true
6   }
7 ]
```

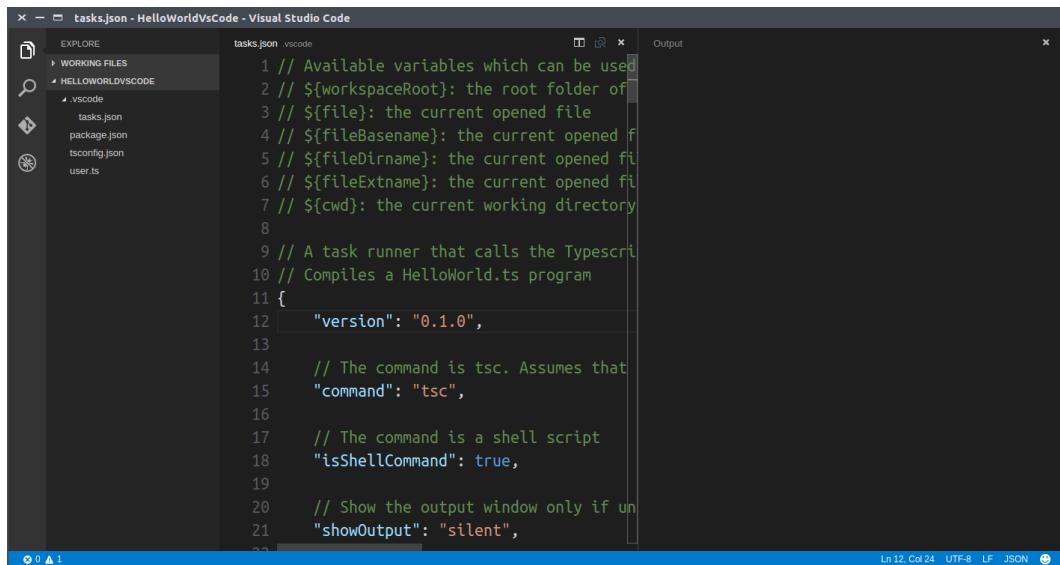
O arquivo `tsconfig.json` será identificado pelo VSCode e suas opções de compilação serão utilizadas. Estas opções (`target`, `module`, `sourceMap`) serão úteis em um projeto mais complexo, neste projeto elas não se aplicam. Por exemplo, o `module` indica a forma como iremos carregar módulos quando estivermos trabalhando com vários arquivos separados. O `sourceMap` indica se o arquivo “map” será criado o que ajudará no debug do TypeScript. Já `target` configura a versão ECMAScript utilizada.

Com o arquivo `tsconfig` criado, vamos criar uma tarefa (task) para compilar o arquivo `ts`. Para isso, aperte F1 e digite “configure”. Surge a opção `Configure Task Runner` semelhante a imagem a seguir:



```
2     "compilerOptions": {
3         "target": "ES5",
4         "module": "amd",
5         "sourceMap": true
6     }
7 }
```

Quando selecionamos Configure Task Runner, uma pasta chamada .vscode é criada, juntamente com o arquivo tasks.json, conforme a imagem a seguir:



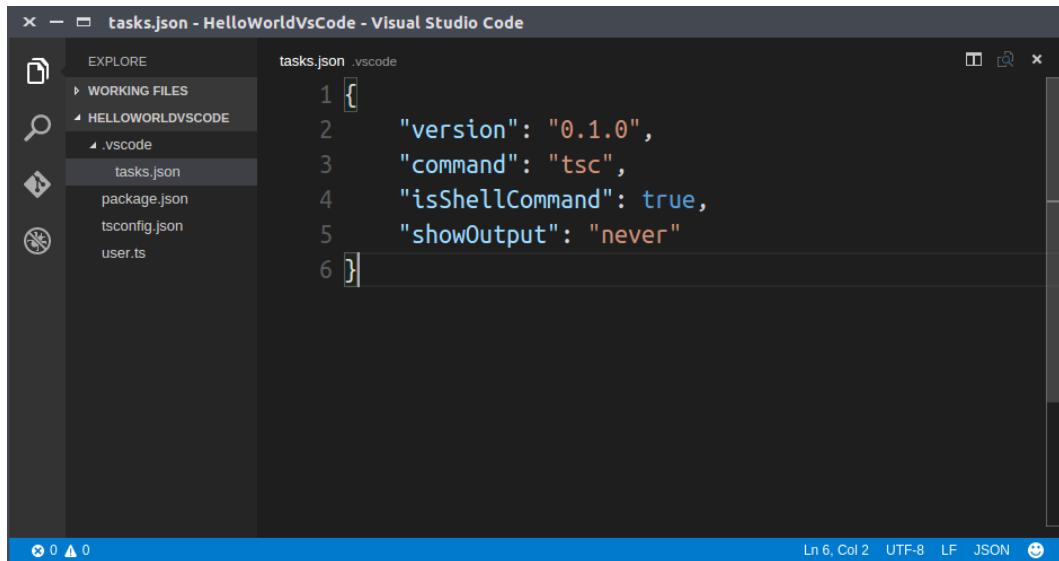
```
1 // Available variables which can be used
2 // ${workspaceRoot}: the root folder of
3 // ${file}: the current opened file
4 // ${fileBasename}: the current opened f
5 // ${fileDirname}: the current opened fil
6 // ${fileExtname}: the current opened fil
7 // ${cwd}: the current working directory
8
9 // A task runner that calls the Typescri
10 // Compiles a HelloWorld.ts program
11 {
12     "version": "0.1.0",
13
14     // The command is tsc. Assumes that
15     "command": "tsc",
16
17     // The command is a shell script
18     "isShellCommand": true,
19
20     // Show the output window only if un
21     "showOutput": "silent",
22 }
```

O arquivo tasks.json possui diversas configurações para diversos tipos de tarefa, mas vamos

adicionar somente o básico para compilar o TypeScript. Apague todo o conteúdo do arquivo e adicione a seguinte configuração:

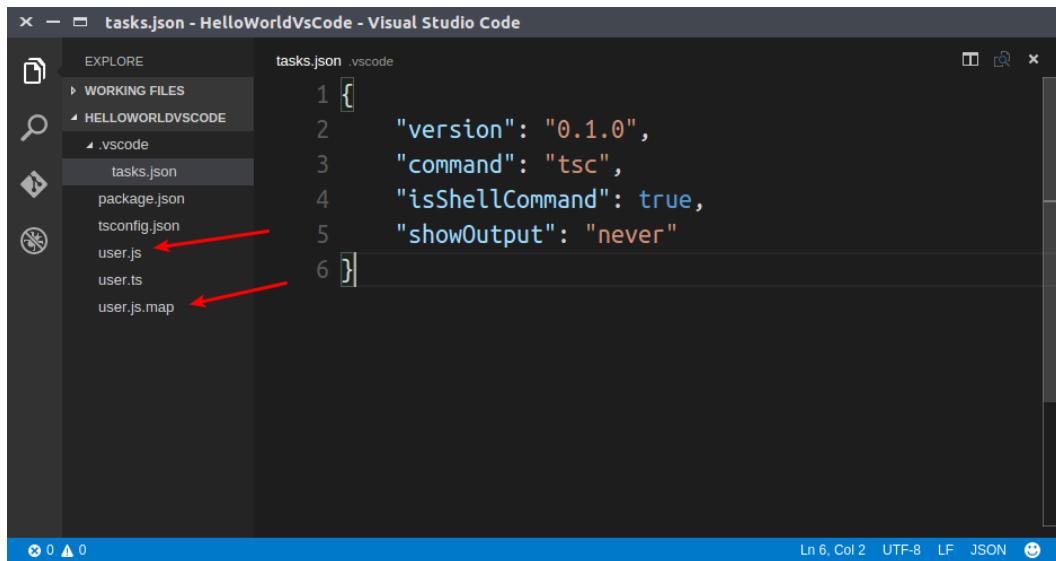
/HelloWorldVSCode/.vscode/tasks.json

```
{  
  "version": "0.1.0",  
  "command": "tsc",  
  "isShellCommand": true,  
  "showOutput": "never"  
}
```



Neste arquivo, criamos a tarefa cujo o comando é tsc (command). Para executar esta tarefa no VSCode, pressione CTRL+SHIFT+B, ou então aperte F1 e digite “run task build”. O comando tsc será executado, o arquivo tsconfig.json será lido e os arquivos js e map serão criados.

Após alguns segundos os arquivos user.js e user.js.map serão criados, semelhante a imagem a seguir:



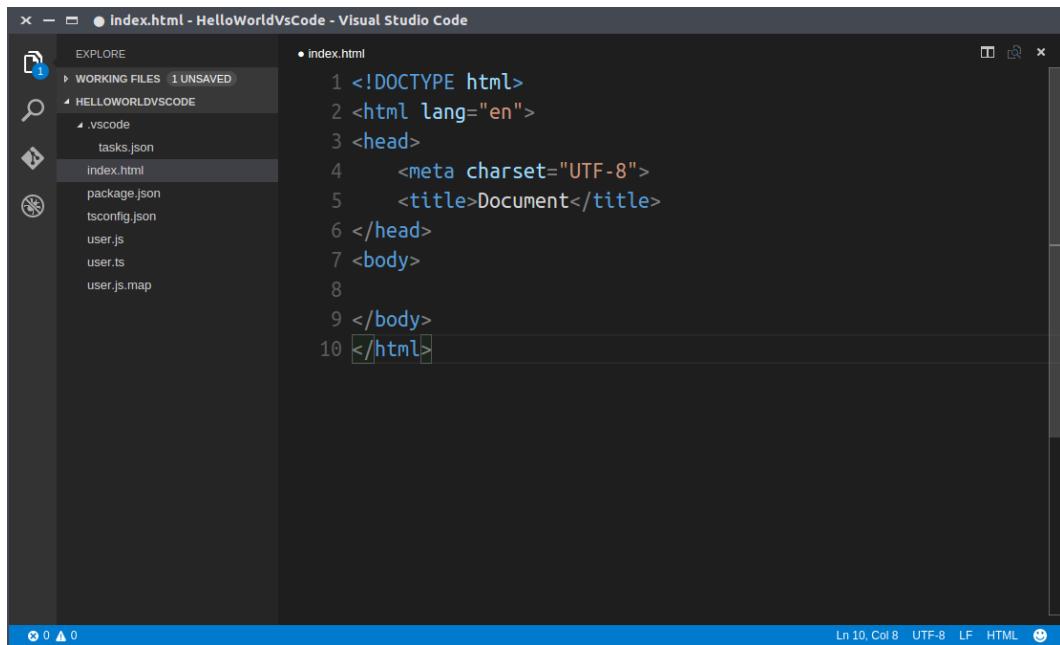
```
tasks.json - HelloWorldVsCode - Visual Studio Code
EXPLORE          tasks.json .vscode
WORKING FILES
  ▾ HELLOWORLDVSCODE
    ▾ vscode
      tasks.json
      package.json
      tsconfig.json
      user.js
      user.ts
      user.js.map

1 {
2   "version": "0.1.0",
3   "command": "tsc",
4   "isShellCommand": true,
5   "showOutput": "never"
6 }
```



Se os arquivos não forem criados, altere o atributo `showOutput` para `always` e verifique o erro em questão. Verifique também se o comando `tsc` está funcionando na linha de comando. Este comando é adicionado ao sistema quando executamos `npm i typescript -g`.

Para testar o arquivo `user.js` no navegador, crie o arquivo `index.html` no diretório `HelloWorldVSCode`. Com o cursor do mouse no início do arquivo `index.html` digite `html:5` e pressione `tab`. O código será convertido em um documento `html` completo, conforme a imagem a seguir:



The screenshot shows the Visual Studio Code interface with the title bar "index.html - HelloWorldVsCode - Visual Studio Code". The left sidebar is titled "EXPLORE" and lists "WORKING FILES 1 UNSAVED" under "HELLOWORLDVSCODE", including ".vscode", "tasks.json", "index.html", "package.json", "tsconfig.json", "user.js", "user.ts", and "user.js.map". The main editor area displays the following HTML code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Document</title>
6   </head>
7   <body>
8
9 </body>
10 </html>
```

The status bar at the bottom right shows "Ln 10, Col 8" and "HTML".

Esta geração de código é realizada pela integração do Visual Studio Code com o emmet, que pode ser consulta [neste link²](#).

Agora coloque o cursor antes do `</body>` e digite `script:src`:

²<http://docs.emmet.io/cheat-sheet/>

The screenshot shows the Visual Studio Code interface. The title bar says "index.html - HelloWorldVsCode - Visual Studio Code". The left sidebar is the "EXPLORE" view, showing the project structure:

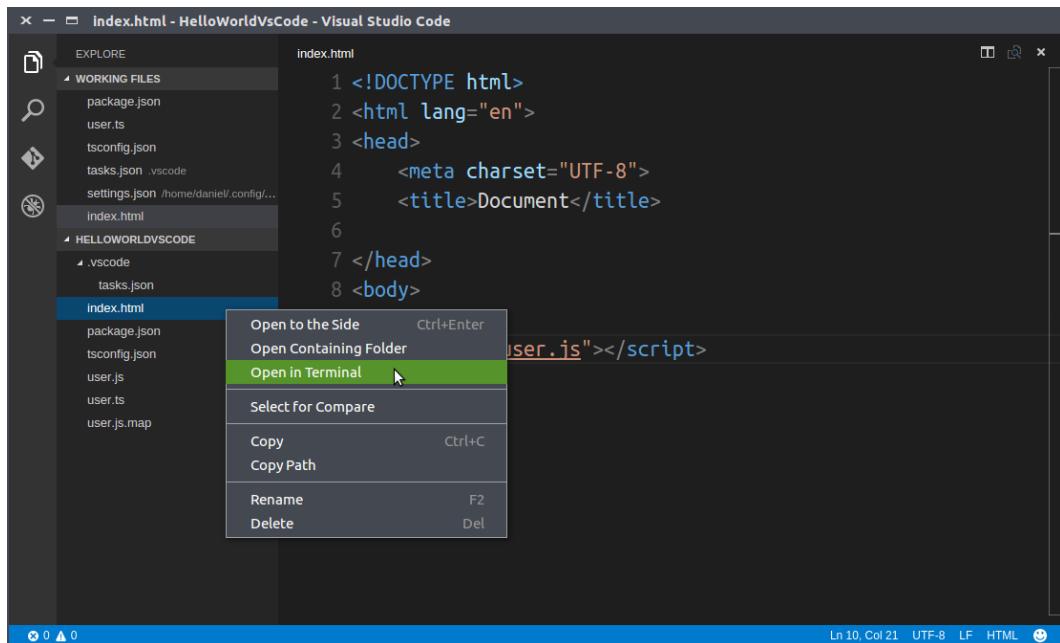
- WORKING FILES (1 UNSAVED)
 - package.json
 - user.ts
 - tsconfig.json
 - tasks.json .vscode
 - settings.json /home/daniel/config/...
 - index.html
- HELLWORLDSVS CODE
 - .vscode
 - tasks.json
 - index.html
 - package.json
 - tsconfig.json
 - user.js
 - user.ts
 - user.js.map

The main code editor window displays the following HTML code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Document</title>
6
7 </head>
8 <body>
9
10 <script>
11   <!-->
12 </script>
```

The status bar at the bottom right shows "Ln 10, Col 11" and "HTML".

Pressione tab e veja que o código gerado foi `<script src=""></script>`, onde devemos adicionar o arquivo `user.js` no atributo `src`. Após adicioná-lo, podemos finalmente testar a aplicação no navegador. Selecione o arquivo `index.html` na lista de arquivos e pressione o botão direito do mouse, escolha o item “Open in terminal”:



Com o terminal aberto no diretório HelloWorldVSCode, digite live-server e aguarde a página abrir. Quando a página abrir, uma mensagem de alert será exibida. Não é preciso fechar o navegador ou reiniciar o live-server para alterar o código TypeScript, bastando apenas recompilar a aplicação. Por exemplo, no arquivo user.ts, onde está `alert(user.hello())`; altere para `document.body.innerHTML = user.hello();` e depois aperte `ctrl+shift+b` para recompilar o arquivo user.ts e alterar a mensagem na tela.

Após realizar estas configurações iniciais, o uso do TypeScript torna-se mais dinâmico, pois não será preciso usar a linha de comando para compilação a cada alteração dos arquivos.

2.2.1 Detectando alterações

Pode-se configurar o VSCode para compilar automaticamente o TypeScript após uma mudança no código. Desta forma, após salvar o arquivo .ts a alteração será percebida e o arquivo javascript será gerado novamente.

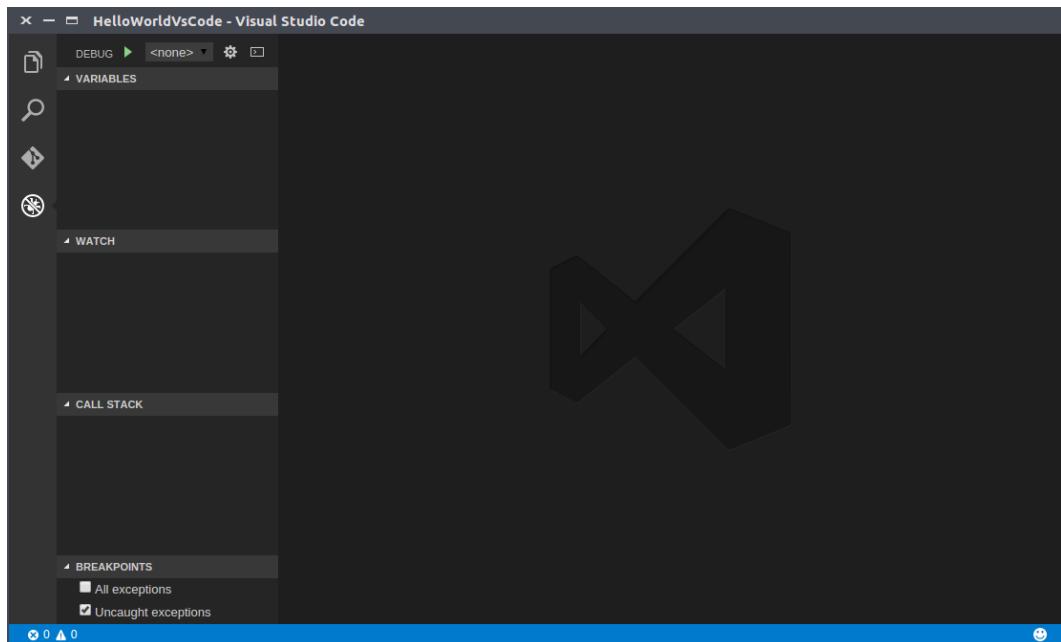
Para isso, altere o arquivo `tsconfig.json` adicionando o item `watch:true` e ativando a saída (`output`) para verificar se o arquivo foi compilado, conforme a configuração a seguir:

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "amd",  
    "sourceMap": true,  
    "watch": true  
  }  
}
```

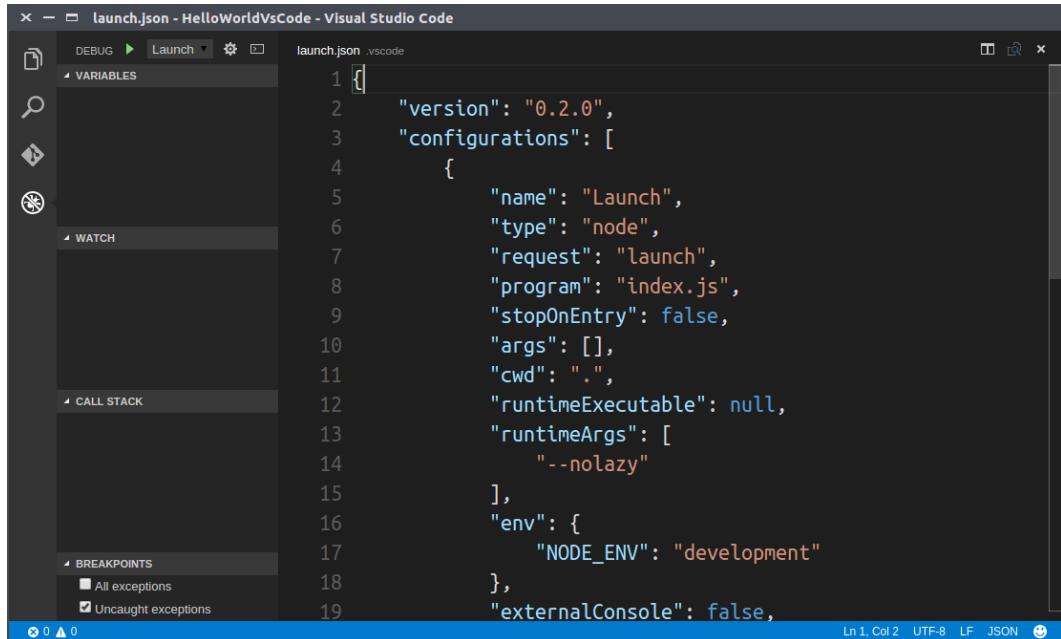
Após a primeira compilação (ctrl+shift+b), a mensagem “Watching for file changes” surgirá na janela de output. Altere o arquivo user.ts e, ao salvar, uma nova compilação será realizada sem a necessidade de pressionar ctrl+shift+b. Para terminar o processo, aperte F1, digite task e selecione Terminate running task.

2.2.2 Debug no Visual Studio Code

Para depurar código no VSCode, clique no ícone de depuração do projeto ou pressione ctrl+shift+d. Surge a tela a seguir, a princípio sem o debug configurado:



Clique na seta verde para iniciar a primeira configuração do depurador. O arquivo `.vscode/launch.json` será criado, de acordo com a imagem a seguir:



The screenshot shows the Visual Studio Code interface with the "DEBUG" tab selected. On the left, there are several panels: "VARIABLES", "WATCH", "CALL STACK", and "BREAKPOINTS". The "BREAKPOINTS" panel has checkboxes for "All exceptions" and "Uncaught exceptions", both of which are checked. The main editor area displays the `launch.json` file content:

```
1 [{}]
2   "version": "0.2.0",
3   "configurations": [
4     {
5       "name": "Launch",
6       "type": "node",
7       "request": "launch",
8       "program": "index.js",
9       "stopOnEntry": false,
10      "args": [],
11      "cwd": ".",
12      "runtimeExecutable": null,
13      "runtimeArgs": [
14        "--nolazy"
15      ],
16      "env": {
17        "NODE_ENV": "development"
18      },
19      "externalConsole": false,
```

At the bottom right of the editor, status information is displayed: Ln 1, Col 2, UTF-8, LF, JSON.

Altere o parâmetro `program` para `user.js` e o parâmetro `sourceMaps` para `true`. Lembre que, sempre que for depurar código TypeScript, deve-se ter o `sourceMap` referente.

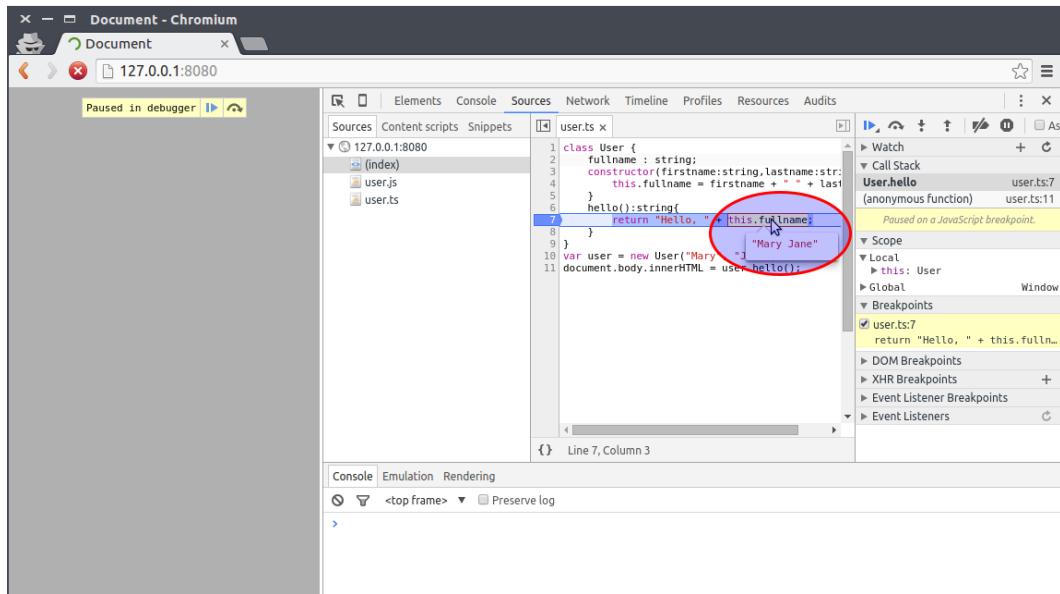
Acesse o arquivo `user.ts` e adicione o breakpoint no método construtor, na linha 4, conforme a imagem a seguir:

```
1 class User {  
2     fullname : string;  
3     constructor(firstname:string,lastname:string) {  
4         this.fullname = firstname + " " + lastname;  
5     }  
6     hello():string{  
7         return "Hello, " + this.fullname;  
8     }  
9 }  
10 var user = new User("Mary", "Jane");  
11 document.body.innerHTML = user.hello();
```

2.2.3 Debug no navegador

É sempre importante ter a depuração (debug) de código ativada no navegador. É fundamental gerar os arquivos “.map” através do “–sourceMap” da configuração do .vscode/tasks.json, ou executar o comando tsc utilizando o parâmetro --sourceMap. Neste projeto, o arquivo user.ts tem o seu arquivo compilado user.js e o seu map user.js.map, permitindo assim que o debug seja usado.

No Google Chrome, pressione F12 e navegue até a aba sources. Encontre o arquivo user.ts e adicione um breakpoint no método hello(). Recarregue a página para verificar que o fluxo de código está parado na linha em que o breakpoint foi adicionado, conforme a imagem a seguir:



2.3 Tipos

Agora que temos um ambiente de trabalho capaz de compilar os arquivos TypeScript em JavaScript podemos iniciar o nosso estudo nesta linguagem. É fundamental aprendê-la para que possamos ter êxito no desenvolvimento Angular 2. Felizmente TypeScript é semelhante às linguagens derivadas da linguagem c, como java ou c#. Diferentemente do JavaScript, no TypeScript podemos criar classes, interfaces, definir tipos de variáveis e tipos de retorno de métodos.

2.3.1 Tipos Básicos

Como já comentamos, com TypeScript é possível adicionar tipos às variáveis. Para criar uma variável, usamos a seguinte sintaxe:

```
var NOME_DA_VARIAVEL : TIPO = VALOR
```

Os tipos básicos existentes são:

- boolean: Pode assumir os valores true ou false
- number: Assume qualquer número, como inteiro ou ponto flutuante.
- string: Tipo texto, pode ser atribuído com aspas simples ou duplas.

2.3.2 Arrays

Já os Arrays no TS podem ser criados através de duas formas. A primeira delas, usa-se [] na definição do tipo da variável, veja:

```
var list:number[] = [1,2,3];
```

A segunda é mais conhecida como “generics” e usa <> para definir o tipo, veja:

```
var list:Array<number> = [1,2,3];
```

Pode-se criar arrays de tipos complexos, como por exemplo a seguir, onde a classe Point é uma classe TypeScript criada previamente.

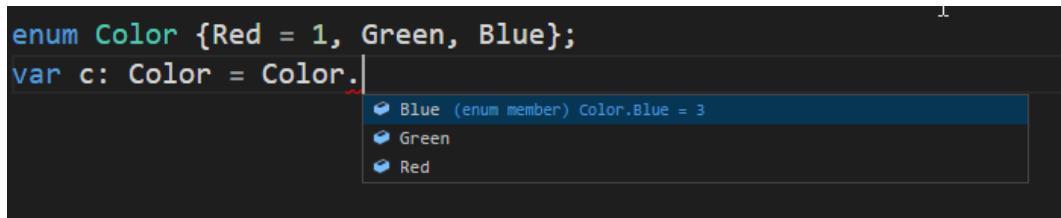
```
var arr:Array<Point> = [
    new Point(10,10),
    new Point(20,30),
    new Point(30,10)
]
```

2.3.3 Enum

Também podemos criar Enums, que são enumerações que podem definir um status ou um conjunto de valores, como no exemplo a seguir:

```
enum Color {Red, Green, Blue};
var c: Color = Color.Green;
```

Enums são facilmente manipulados através do Visual Studio Code, como na imagem a seguir, com a complementação de código (ctrl+espaço):



2.3.4 Any

O tipo `any` assume qualquer tipo, sendo `string`, `number`, `boolean` etc.

2.3.5 Void

O `void` é usado para determinar que um método não retorna nenhum valor, conforme o exemplo a seguir.

```
function warnUser(): void {
    alert("This is my warning message");
}
```

2.4 Classes

O conceito de classes no TypeScript é o mesmo de uma classe em qualquer linguagem orientada a objetos. As classes no TypeScript seguem o padrão ECMAScript 6. A classe possui uma sintaxe muito familiar com c#, veja:

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
var greeter = new Greeter("world");
```

2.4.1 Construtor

O construtor é definido pela palavra `constructor`. Métodos não necessitam da palavra `function`, bastando apenas usar `nomedometodo()`.

2.4.2 Visibilidade de métodos e propriedades

Perceba que, no exemplo apresentado, não definimos visibilidade das propriedades da classe, nem o tipo de retorno do método greet. É claro que podemos definir estes parâmetros, conforme o próximo exemplo.

```
class Greeter {  
    private greeting: string;  
    constructor(message: string) {  
        this.greeting = message;  
    }  
    public greet(): string {  
        return "Hello, " + this.greeting;  
    }  
}  
  
var greeter:Greeter = new Greeter("world");
```

Métodos e propriedades de uma classe podem assumir a visibilidade: private, public e protected.

- private: São visíveis apenas na própria classe.
- public: São visíveis em todas as classes.
- protected: São visíveis na classe e sub classes.

2.5 Herança

A herança entre uma classe e outra é definida pela palavra extends. Pode-se sobrepor métodos e usar a palavra super para chamar o método da classe pai, conforme o exemplo a seguir.

```
class Animal {
    name:string;
    constructor(theName: string) { this.name = theName; }
    move(meters: number = 0) {
        alert(this.name + " moved " + meters + "m.");
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 5) {
        alert("Slithering...");
        super.move(meters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 45) {
        alert("Galloping...");
        super.move(meters);
    }
}

var sam = new Snake("Sammy the Python");
var tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);
```

Neste exemplo, usamos o `super` da classe `Snake` para chamar o método construtor da classe pai `Animal`. Se isso não for claro para você, estude orientação a objetos para que possa compreender melhor, pois estas características são da Orientação em Objetos como um todo, e não do TypeScript.

2.6 Accessors (get/set)

Os accessors visam proteger as propriedades de uma classe. Os accessors do TypeScript são feitos pelas palavras `get` e `set`. Veja o exemplo a seguir:

```
class Pessoa {
    private _password: string;

    get password(): string {
        return this._password;
    }

    set password(p : string) {
        if (p != "123456") {
            this._password = p;
        }
        else {
            alert("Ei, senha não pode ser 123456");
        }
    }
}

var p = new Pessoa();
p.password = "123456"; //exibe o erro
```

2.7 Métodos Estáticos

É possível criar métodos estáticos definindo a palavra `static` antes do método. Existem dezenas de aplicações para métodos estáticos, sendo uma delas não precisar instanciar uma classe, como no exemplo a seguir.

```
class SystemAlert{  
  
    static alert(message:string):void{  
        alert(message);  
    }  
  
    static warn (message:string):void{  
        alert("Atenção: " + message);  
    }  
  
    static error(message:string):void{  
        alert("Erro: " + message);  
    }  
  
}  
  
SystemAlert.alert("Oi");  
SystemAlert.error("Não foi possível conectar na base de dados");
```

2.8 Interfaces

Uma interface define um contrato para a classe. A interface é criada da seguinte forma:

```
interface Point{  
    x: number;  
    y: number;  
    z: number;  
}
```

Para implementar a interface, usamos `implements`:

```
class point3d implements Ponto{  
    //...  
}
```

2.9 Funções

Vamos exemplificar algumas particularidades de uma função em TypeScript. A função pode ser criada fora de uma classe ou dentro, sendo as observações que faremos a seguir podem ser aplicadas em ambas.



Em uma classe não precisamos usar a palavra `function` para definir uma função, mas fora da classe precisamos.

2.9.1 Valor padrão

Pode-se definir um valor padrão para um parâmetro de uma função da seguinte forma:

```
function buildName(firstName: string, lastName : string = "Smith") {  
}  
// or  
class Foo{  
    buildName(firstName: string, lastName : string = "Smith") {  
    }  
}
```

2.9.2 Valor opcional

Use o caractere `?` para definir um parâmetro opcional.

```
class Foo{  
    buildName(firstName: string, lastName? : string) {  
        if (lastName){  
            // blablabla  
        }  
    }  
}
```

2.10 Parâmetros Rest

Pode-se repassar um array de valores diretamente para um parâmetro. É válido lembrar que este modo só pode ser usado no último parâmetro da sua função. Exemplo:

```
class Foo{  
    static alertName(firstName: string, ...restOfName: string[]) {  
        alert(firstName + " " + restOfName.join(" "));  
    }  
}  
Foo.alertName("Fulano", "de", "Tal");
```

2.11 Parâmetros no formato JSON

Uma das maiores facilidades do Javascript é repassar parâmetros no formato JSON. Com TypeScript é possível utilizar este mesmo comportamento, conforme o exemplo a seguir.

```
class Point{  
  
    private _x : number = 0;  
    private _y : number = 0;  
    private _z : number = 0;  
  
    constructor( p: {x:number;y:number;z?:number;} ){  
        this._x = p.x;  
        this._y = p.y;  
        if (p.z)  
            this._z = p.z;  
    }  
  
    is3d():boolean{  
        return this._z!=0;  
    }  
}  
  
var p1 = new Point({x:10,y:20});  
  
alert(p1.is3d()); //false
```

2.12 Módulos

Na aprendizagem do Angular 2 pode-se escrever código em um único arquivo, criando várias classes em um mesmo bloco de código, mas no desenvolvimento real cada classe da aplicação deve ser um arquivo TypeScript diferente.

A forma como carregamos os módulos, classes e arquivos no javascript depende de vários fatores, inclusive da escolha de uma biblioteca para tal. Nesta obra, optamos pelo **systemjs**, pois é a forma utilizada pelo Angular 2.

2.12.1 Exemplo com Systemjs

Vamos criar um novo diretório chamado `testModules`, e usar o Visual Studio Code para abrir este diretório através do menu `File > Open Folder`. Vamos criar inicialmente dois arquivos TypeScript chamados de `index.ts`, que será o arquivo principal da aplicação, e o arquivo `module.ts` que será um módulo genérico.

`module.ts`

```
export class foo{
    getHelloWorldFromModule():string{
        return "Hello World from modules";
    }
}
```

A classe `foo` contém um modificador chamado `export`, inserido antes do `class`. Este modificador diz ao `module.ts` que a classe poderá ser exportada para outras classes da aplicação. Com o módulo pronto, criamos o arquivo `index.ts`, com o seguinte código:

`index.js`

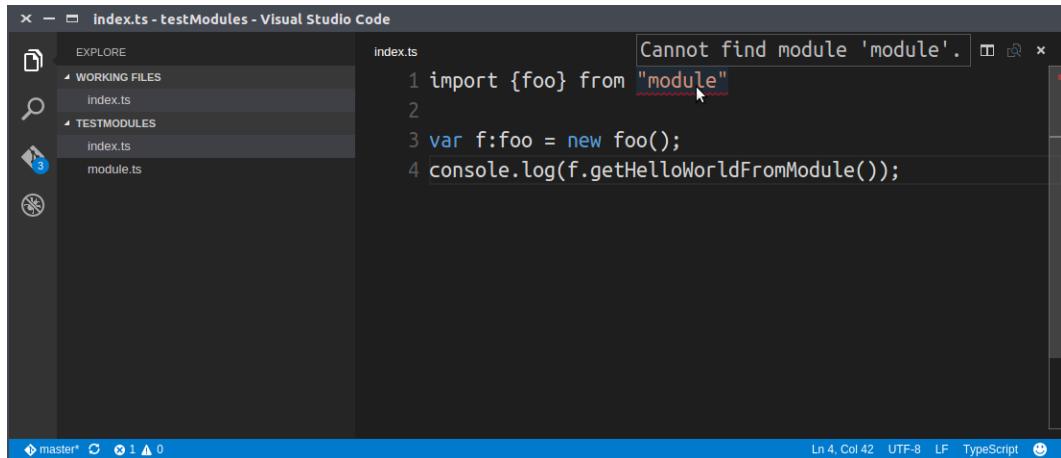
```
import { foo } from "module"

var f:foo = new foo();
console.log(f.getHelloWorldFromModule());
```

Na primeira linha do arquivo `index.ts` usamos a diretiva `import` para importar a classe `foo` do módulo `module`. Nesta sintaxe, o que é inserido entre chaves são as classes a serem

importadas, e o após a palavra `from` inserimos o nome do arquivo do módulo. Como o arquivo é `module.js`, usamos `module`.

Neste primeiro momento ainda não temos as configurações necessárias para que o Visual Studio Code comprehenda o código TypeScript apresentado e possivelmente sua tela será semelhante à imagem a seguir:



Perceba o erro em “module”, pois o editor Visual Studio Code não foi capaz de encontrar o módulo. Na verdade o VSCode ainda nem conseguiu determinar a forma de carregamento de módulos utilizada. Conforme foi visto anteriormente, é necessário criar o arquivo `tsconfig.json` com estas configurações, de acordo com o código a seguir:

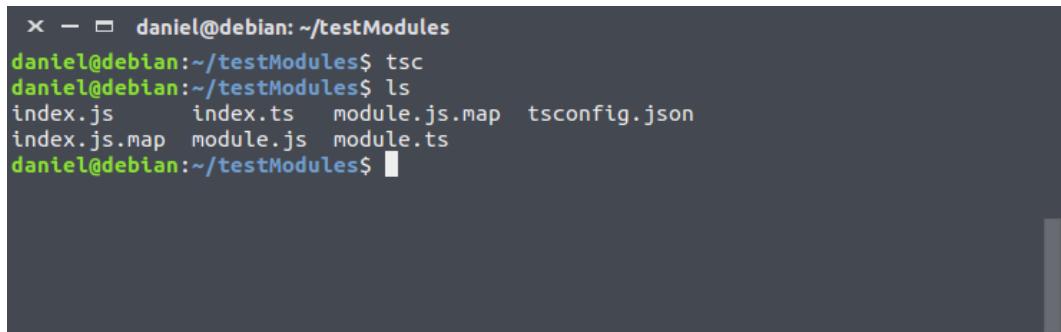
tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "sourceMap": true
  },
  "files": [
    "index.ts"
  ]
}
```

O arquivo de configuração do TypeScript indica através da propriedade `module` qual será

a estratégia de carregamento de arquivos. Neste caso usaremos `system`. `SourceMap` também está habilitado para a geração dos arquivos `.map` com o objetivo de auxiliar o Debug. Também incluímos a propriedade `files` que configura quais os arquivos serão compilados para JavaScript. Se omitirmos a propriedade `files`, todos os arquivos do diretório `testModules` serão compilados. Como fornecemos apenas o arquivo `index.ts`, ele será compilado primeiro, e outros serão compilados se, e somente se, forem importados.

Relembrando a importância do arquivo `tsconfig.json`, ele auxilia o comando `tsc` que foi instalado no sistema através do `npm`. Isso significa que se executarmos `tsc` na linha de comando, dentro da pasta `testModules`, tudo será compilado, conforme a imagem a seguir:



```
x - □ daniel@debian: ~/testModules
daniel@debian:~/testModules$ tsc
daniel@debian:~/testModules$ ls
index.js      index.ts     module.js.map  tsconfig.json
index.js.map  module.js   module.ts
daniel@debian:~/testModules$ █
```

Mas não é o nosso objetivo abrir um terminal e executar o comando `tsc` a todo o momento que queremos compilar o projeto. Para isso iremos configurar uma tarefa (task) no Visual Studio Code. No VSCode, aperte F1 e digite Configre Task Runner. O arquivo `.vscode/tasks.json` será criado com diversas opções. Reduza o arquivo para a seguinte configuração json:

`.vscode/tasks.json`

```
{  
    "version": "0.1.0",  
    "command": "tsc",  
    "isShellCommand": true,  
    "showOutput": "silent",  
    "problemMatcher": "$tsc"  
}
```

Esta configuração irá executar o comando `tsc` com as opções do `config.json`.

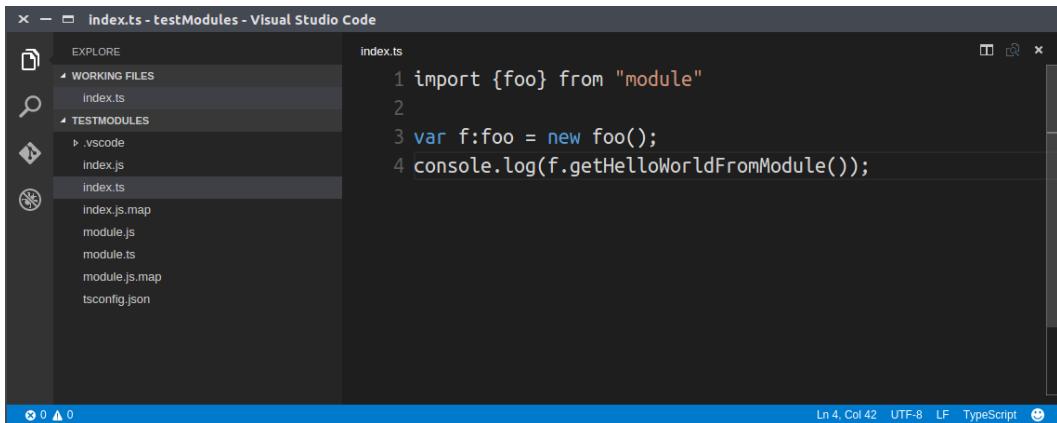


Não inclua no arquivo `tasks.json` a propriedade `args`, pois se isso for feito, o arquivo `tsconfig.json` será completamente ignorado.

Com a task configurada, basta pressionar `ctrl+shift+b` para executar a tarefa. Os arquivos `.js` e `.js.map` serão criados, e o erro inicial no module não acontecerá novamente.

2.12.2 Omitindo arquivos js e map no VSCode

Após a compilação inicial, a interface do Visual Studio Code deve ser semelhante à imagem a seguir:



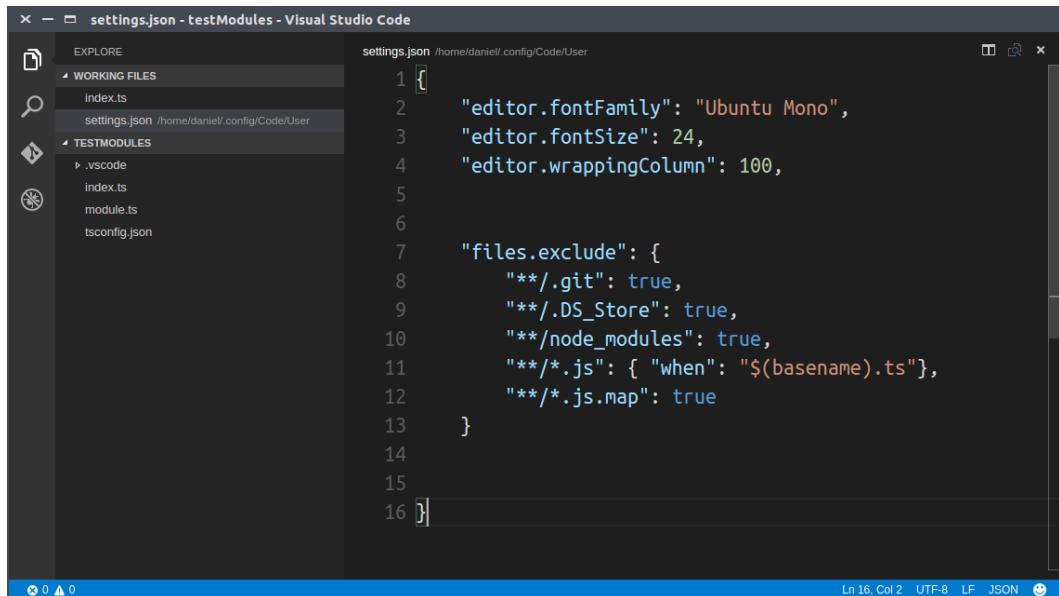
Perceba que a relação de arquivos está um pouco confusa pois existem três arquivos semelhantes, como por exemplo: `index.js`, `index.ts` e `index.js.map`. Os arquivos `.js` e `.map` são os arquivos compilados do TypeScript, e não há a necessidade que eles apareçam na lista de arquivos do editor.

Para resolver este problema, acesse `File > Preferences > User Settings`, onde o arquivo global de preferências do usuário será aberto. Para que possamos omitir os arquivos `.js` e `.map` do editor, devemos configurar a propriedade `files.exclude`, conforme o código a seguir:

```
"files.exclude": {
    "**/.git": true,
    "**/.DS_Store": true,
    "**/node_modules": true,
    "**/*.js": { "when": "$(basename).ts" },
    "**/*.js.map": true
}
```

Neste exemplo, as pastas .git, .DS_Store, node_modules serão omitidas. Na penúltima configuração: "**/*.js": { "when": "\$(basename).ts" } estamos omitindo todos os arquivos com a extensão .js que possuem o seu correspondente .ts e na última configuração estamos omitindo todos os arquivos com a extensão .js.map.

Após inserir estas informações, o VSCode fica semelhante à figura a seguir:



2.12.3 Uso do SystemJS

Agora que já criamos e compilamos os arquivos TypeScript é preciso testá-los no navegador. Para isso deve-se criar o arquivo index.html e inserir a biblioteca SystemJS, que deve ser obtida através do npm.

Incialmente vamos inicializar o *npm* no diretório através do comando `npm init`, conforme a imagem a seguir:

```
x - □ daniel@debian: ~/testModules
daniel@debian:~/testModules$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (testModules)
version: (0.0.0)
description: A test module app
entry point: (index.js)
test command: tsc
git repository:
keywords: test module systemjs
author: daniel
license: (ISC) █
```

Com o arquivo `package.json` criado, é possível adicionar a biblioteca `systemjs`, através do seguinte comando:

```
$ npm i systemjs -S
```

Como esperado o diretório `node_modules` será criado e o `systemjs` será adicionado à ela, conforme a figura a seguir:

```
x - □ daniel@debian: ~/testModules
daniel@debian:~/testModules$ ls
index.js      index.ts    module.js.map  package.json
index.js.map  module.js   module.ts     tsconfig.json
daniel@debian:~/testModules$ npm i systemjs -S
npm WARN package.json testModules@0.0.0 No repository field.
npm WARN package.json testModules@0.0.0 No README data
systemjs@0.19.11 node_modules/systemjs
└── es6-module-loader@0.17.10
  └── when@3.7.7
daniel@debian:~/testModules$ ls
index.js      index.ts    module.js.map  node_modules  tsconfig.json
index.js.map  module.js   module.ts     package.json
daniel@debian:~/testModules$ ls node_modules/
systemjs
daniel@debian:~/testModules$
```

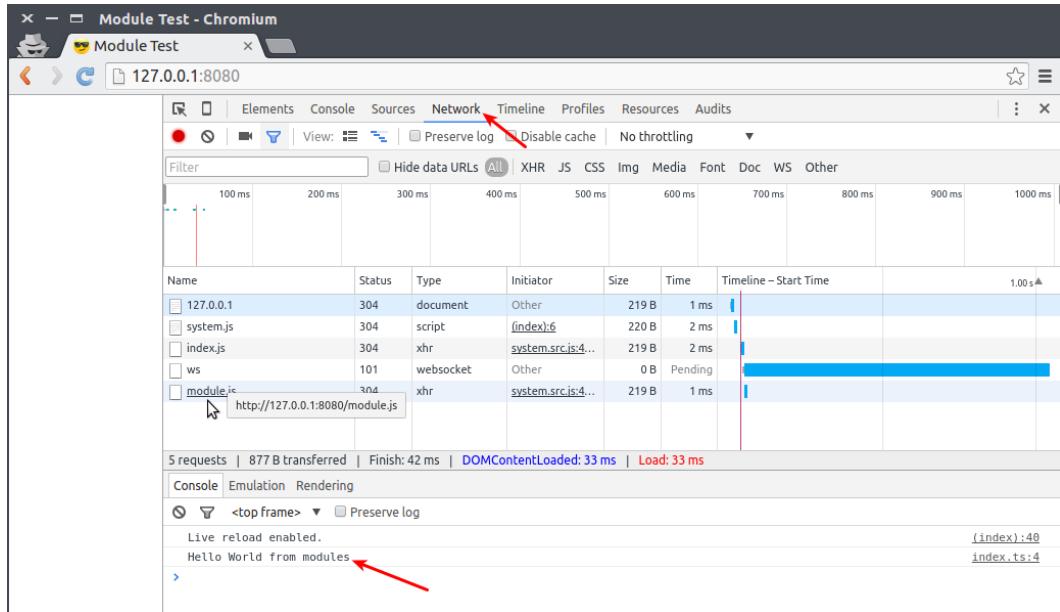
Com a biblioteca systemjs pronta, podemos criar o arquivo `index.html`, com o seguinte conteúdo:

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Module Test</title>
  <script src="node_modules/systemjs/dist/system.js"></script>
  <script>
    System.defaultJSExtensions = true;
    System.import('index');
  </script>
</head>
<body>
</body>
</html>
```

O arquivo `index.html` contém a inclusão da biblioteca `system.js`, instalada previamente pelo `npm`. Após incluir `system.js` no documento `html`, adicionamos `System.defaultJSExtensions = true` para habilitar a extensão automática dos arquivos, e `System.import('index')` para importar o arquivo `index.js`.

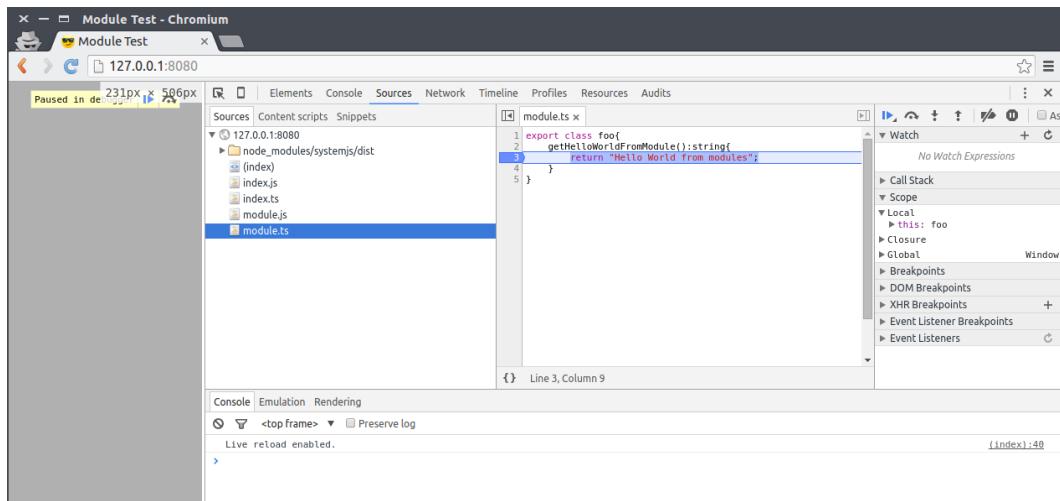
O arquivo `index.js`, por sua vez, importa o arquivo `modules.js`. Este procedimento pode ser analisado no Google Chrome (aperte F12 após carregar a página), na aba Network, conforme a figura a seguir:



The screenshot shows the Network tab in the Google Chrome DevTools. A red arrow points to the 'Timeline' tab at the top. Below it, a table lists network requests. The 'Timeline - Start Time' column shows the sequence of requests. The 'module.js' request, which originated from `http://127.0.0.1:8080/module.js`, is highlighted with a mouse cursor. Another red arrow points to the 'Console' tab at the bottom, where the output `Hello World from modules` is visible.

Name	Status	Type	Initiator	Size	Time	Timeline - Start Time	
127.0.0.1	304	document	Other	219 B	1 ms		
system.js	304	script	(index):6	220 B	2 ms		
index.js	304	xhr	system.src.js:4...	219 B	2 ms		
ws	101	websocket	Other	0 B	Pending		
module.js	304	xhr	system.src.js:4...	219 B	1 ms		

Para habilitar o Debug no Google Chrome, navegue até a aba Sources, selecione o arquivo TypeScript que deseja debugar, e adicione um breakpoint, conforme a figura a seguir:



Para acessar o arquivo index.html no navegador, use o live-server que foi instalado globalmente. Basta acessar o diretório testModules e digitar live-server, conforme a figura a seguir:

```
x - daniel@debian:~/testModules
daniel@debian:~$ cd testModules/
daniel@debian:~/testModules$ live-server
Serving "/home/daniel/testModules" at http://127.0.0.1:8080
```

2.13 Decorators (ou annotation)

Decorators são amplamente usados em linguagens de programação. Sua principal funcionalidade é configurar uma classe/variável/método através de propriedades. Em algumas linguagens um *decorator* é chamado de *annotation*.

Por exemplo, a biblioteca Hibernate do Java usa decorators para configurar uma classe que representa uma tabela, como no exemplo a seguir:

```
import javax.persistence.*;  
  
@Table(name = "EMPLOYEE")  
public class Employee {  
    @Id @GeneratedValue  
    @Column(name = "id")  
    private int id;  
}
```

No exemplo acima, temos o decorator `@Table` com a propriedade `name=="EMPLOYEE"`. O campo `id` também tem o decorator `@id` e `@Column`, configurando o campo da tabela. O mesmo pode ser aplicado no C#, como no exemplo a seguir:

```
public class Product  
{  
    [Display(Name="Product Number")]  
    [Range(0, 5000)]  
    public int ProductID { get; set; }  
}
```

E finalmente o mesmo pode ser aplicado no TypeScript, e é amplamente utilizado no Angular 2. Um exemplo de decorator no Angular 2 é apresentado a seguir:

```
import {Component} from 'angular2/core'  
  
@Component({  
    selector: 'my-app',  
    template: '<h1>My First Angular 2 App</h1>'  
})  
export class AppComponent { }
```

O uso do `@Component` antes da classe `AppComponent` é um decorator do Angular 2. Neste exemplo, o uso do `@Component` transforma a classe `AppComponent` em um componente do Angular, e com duas propriedades definidas: `selector` e `template`.

Não é o foco desta obra aprender a criar um decorator, mas sim usá-los. Uma característica importante que deve-se saber sobre os decorators em TypeScript é que o arquivo `tsconfig.json` deve incluir duas novas informações: `emitDecoratorMetadata` e `experimentalDecorators`, como será visto no próximo capítulo.

2.14 Conclusão

Neste capítulo tivemos uma ampla abordagem ao TypeScript, que é a linguagem que pode ser usada no Angular 2. O TypeScript está ganhando mercado por trazer uma alternativa mais sólida ao desenvolvimento JavaScript, prova disso é o próprio framework Angular 2. Também vimos o Visual Studio Code que é um editor de textos muito poderoso e com ferramentas na medida certa para o desenvolvimento JavaScript.

3. Um pouco de prática

Neste capítulo iremos introduzir alguns conceitos sobre o Angular 2. A primeira tarefa será criar um projeto básico, no qual poderemos Copiar/Colar para criar novos projetos.

3.1 Projeto AngularBase

Inicialmente criaremos um projeto básico para que possamos, a qualquer momento, copiar e colar para um novo projeto.

3.1.1 Configurando o projeto

Crie a pasta AngularBase e, pela linha de comando, configure o arquivo package.json através do npm:

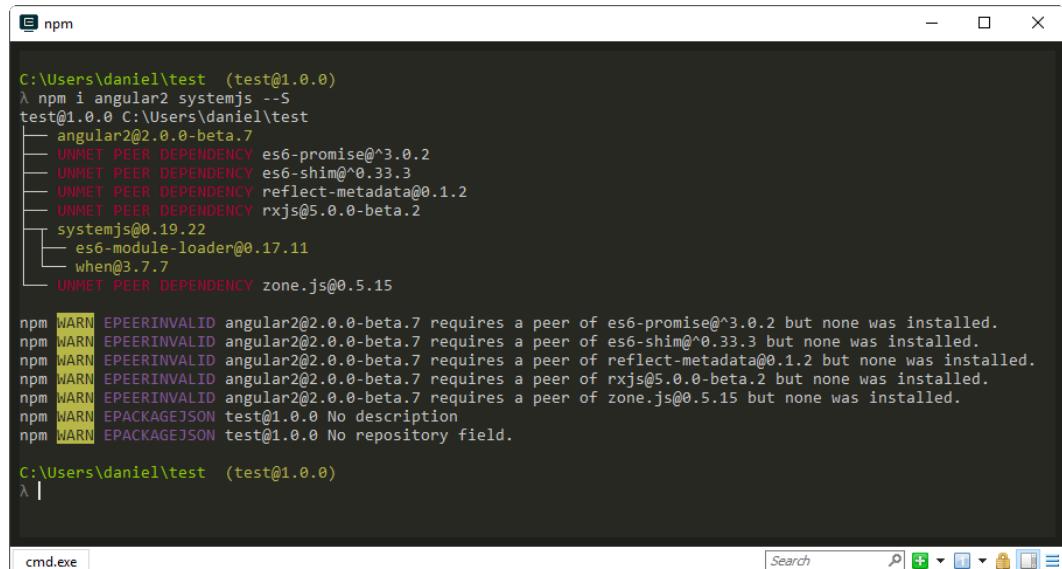
```
$ npm init  
...(configure as opções)...
```

Adicione as bibliotecas angular2 e systemjs ao projeto:

```
$ npm i angular2 systemjs --S
```

3.1.2 Erros ao instalar o Angular 2

Na versão 3 do npm, alguns erros podem acontecer na instalação do Angular 2, como no exemplo a seguir:



```
C:\Users\daniel\test (test@1.0.0)
λ npm i angular2 systemjs --S
test@1.0.0 C:\Users\daniel\test
+-- angular2@2.0.0-beta.7
|   +-- UNMET PEER DEPENDENCY es6-promise@^3.0.2
|   +-- UNMET PEER DEPENDENCY es6-shim@^0.33.3
|   +-- UNMET PEER DEPENDENCY reflect-metadata@0.1.2
|   +-- UNMET PEER DEPENDENCY rxjs@5.0.0-beta.2
|   +-- systemjs@0.19.22
|       +-- es6-module-loader@0.17.11
|           +-- when@3.7.7
|               +-- UNMET PEER DEPENDENCY zone.js@0.5.15
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of es6-promise@^3.0.2 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of es6-shim@^0.33.3 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of reflect-metadata@0.1.2 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of rxjs@5.0.0-beta.2 but none was installed.
npm WARN EPEERINVALID angular2@2.0.0-beta.7 requires a peer of zone.js@0.5.15 but none was installed.
npm WARN EPACKAGEJSON test@1.0.0 No description
npm WARN EPACKAGEJSON test@1.0.0 No repository field.

C:\Users\daniel\test (test@1.0.0)
λ |
```

Até que a instalação de dependências pelo npm volte a funcionar corretamente, você terá que resolver este problema instalando manualmente as bibliotecas que resultaram em erro. Copie o nome da biblioteca seguido da sua versão e use o npm novamente para a instalação:

```
$ npm i es6-promise@^3.0.2 --S
$ npm i es6-shim@^0.33.3 --S
$ npm i reflect-metadata@0.1.2 --S
$ npm i rxjs@5.0.0-beta.2 --S
$ npm i zone.js@0.5.15 --S
```

Perceba que as versões de cada biblioteca podem mudar de acordo com a versão beta-x do Angular 2. Então não copie do livro os comandos de instalação, copie dos erros gerados pelo npm no seu sistema.

3.1.3 Configurando a compilação do TypeScript

Abra o diretório AngularBase no Visual Studio Code e crie o arquivo `tsconfig.json`. Como visto no capítulo anterior, o arquivo `tsconfig.json` conterá informações de compilação do TypeScript para o JavaScript. O conteúdo deste arquivo é:

tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "system",  
    "sourceMap": true,  
    "moduleResolution": "node",  
    "emitDecoratorMetadata": true,  
    "experimentalDecorators": true,  
    "removeComments": false,  
    "noImplicitAny": false  
  },  
  "exclude": [  
    "node_modules"  
  ]  
}
```

Esta configuração apresenta alguns detalhes extras em relação às configurações anteriores. Em `moduleResolution` é definido a estratégia de carregamento dos módulos, neste caso `node`. Desta forma, podemos carregar módulos do diretório `node_modules`. As configurações `emitDecoratorMetadata` e `experimentalDecorators` habilitam o uso de *decorators*. Como o Angular 2 usa-os extensivamente, é necessário ativá-las. Em `noImplicitAny` iremos desabilitar a mensagem de erro para algum método ou propriedade sem um tipo definido. Já `exclude` irá excluir a compilação de qualquer arquivo TypeScript que estiver no diretório `node_modules`.

Para configurar a tarefa (*task*) que irá compilar o TypeScript, pressione `ctrl+shift+b` (Se estiver utilizando o *Visual Studio Code*) e caso ainda não exista, crie-a clicando em *Configure Task Runner*. O arquivo `.vscode/tasks.json` conterá a seguinte configuração:

.vscode/tasks.json

```
{  
    "version": "0.1.0",  
    "command": "tsc",  
    "isShellCommand": true,  
    "showOutput": "silent",  
    "problemMatcher": "$tsc"  
}
```



Se estiver utilizando outro editor de textos, ao invés de configurar uma task, abra o terminal, navegue até o diretório e digite: tsc

3.1.4 Criando o primeiro componente Angular 2

Até o momento nenhum arquivo TypeScript foi criado, então ainda não há o que compilar. O Angular 2 é focado na criação de componentes, então ao invés de adicionarmos código html ou javascript para construir a aplicação, vamos focar na criação e manipulação de componentes.

Como convenção, vamos criar o diretório app, que é onde os componentes ficarão. Cada componente terá a extensão .component.ts, e o primeiro componente a ser criado é o que define a aplicação. Neste caso, criaremos o arquivo app/app.component.ts com o seguinte código:

app/app.component.ts

```
import {Component} from 'angular2/core'  
  
@Component({  
    selector: 'my-app',  
    template: '<h1>My First Angular 2 App</h1>'  
})  
export class AppComponent { }
```

Na primeira linha, usamos o comando `import` para importar a classe `Component`, diretamente do Angular 2. Perceba que `angular2/core` está definido em `node_modules/angular2/core.js`. Após o `import`, é possível usar o `decorator @Component` e repassar dois parâmetros: `selector` e `template`. O parâmetro `selector` define qual a tag será usada para a instanciação do componente na página `html`. Com o valor `my-app`, subtende-se que ao usarmos `<my-app></my-app>` o componente será carregado. Já `template` define o código `html` que será usado como template no componente.

Após o `decorator @Component` ser criado podemos criar a classe `AppComponent`, que usa o `export` para dizer que a classe é pública ao módulo.

3.1.5 Criando o bootstrap

O `bootstrap` é responsável em inicializar o Angular 2, fornecendo diversas informações (serão revisadas nos próximos capítulos). Crie o arquivo `app/boot.ts` com o seguinte código:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'

bootstrap(AppComponent);
```

Na primeira linha do arquivo `app/boot.ts` importamos o método `bootstrap` de `angular2/platform/browser`. Na segunda linha, importamos o componente `AppComponent`. Perceba que o caminho usado no `from` é `./app.component`, onde o `./` indica a raiz do arquivo, neste caso `app`. Após os `imports`, chamamos o método `bootstrap(AppComponent);` que irá inicializar a aplicação Angular.

3.1.6 Criando o arquivo html

Para finalizar o projeto base, precisamos criar o `html` que conterá a aplicação. Crie o arquivo `index.html` na pasta `AngularBase` com o seguinte código:

index.html

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
      System.import('app/boot')
        .then(null, console.error.bind(console));
    </script>
  </head>
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```



Atenção Em alguns códigos do arquivo PDF você verá o fim de uma linha com uma contra barra “\\”, conforme o detalhe da imagem a seguir:
index.html

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></scri\
pt>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <script>
      System.config({
        packages: {

```

Isso indica que a linha não acabou, e continua na linha seguinte. Se você for copiar e colar o código, retire a contra barra do código também. Neste exemplo, ao invés de </script>\pt>, o correto seria </script>

Este arquivo html pode ser separado em três blocos. O primeiro contém o carregamento das bibliotecas padrão do Angular 2, juntamente com o SystemJS que foi abordado no capítulo anterior. Depois usamos a configuração da biblioteca SystemJS para definir o formato da aplicação, repassando a propriedade format como register. Neste caso o register é o mesmo que system.register. Também usamos a propriedade defaultExtension com o valor js, ou seja, todo o arquivo requisitado para o SystemJS terá a extensão adicionada. Na tag <body> temos a criação da tag <my-app></my-app> onde a aplicação será renderizada.

O comando System.import irá importar app/boot que é o arquivo de bootstrap que criamos. A partir do app/boot outras classes serão criadas.

Para testar o projeto, recompile-o com **ctrl+shift+b** no Visual Studio Code (ou digite tsc no terminal, no diretório AngularBase) e use o live-server para iniciar o servidor web.

Com o projeto finalizado, pode-se usá-lo para copiar e colar em um novo projeto (Copiar a pasta e colar em uma nova pasta com outro nome).

3.2 Criando uma pequena playlist

O foco desta obra é apresentar o Angular 2 através de exemplos, na prática. Ao invés de exibir a teoria sobre os mais diferentes aspectos do Angular 2, tais como templates, components, diretivas, metadata, databind, injeção de dependência, entre outros, vamos criar uma aplicação funcional e explicar a teoria somente no próximo capítulo.

Nossa próxima aplicação é uma *playlist*, pode ser de vídeos contendo o título, a url e uma descrição. A princípio a *playlist* mostra somente o título, e quando o usuário clicar no item da lista, poderá acessar a Url e a descrição. Também pode-se editar e criar novos vídeos, mas ao atualizar a página estas modificações serão perdidas, pois em um primeiro momento não estaremos comunicando com o servidor ou com o banco de dados.

Resultado final da playlist:

My Playlist

ID	Title
1	Building apps with Firebase and Angular 2 - Sara Robinson
2	Better concepts, less code in Angular 2 - Victor Savkin and Tobias Bosch
3	aaaaaaaaaa
4	12121212A new video

Building apps with Fir... ⌚ ↗

18n support???

Building apps with Firebase and Angular 2 - Sara Robinson

<http://www.youtube.com/embed/RD0xYicNcaY>

Firebase is a powerful platform for building mobile and web applications. Use Firebase to store and sync data instantly, authenticate users, and easily deploy your web app. In this talk, you'll learn how you can use Firebase to add a backend to your Angular app in minutes. Sara will demonstrate how to get started with Firebase and Angular 2. At the end she'll risk it all by live coding and deploying an app with Firebase and Angular!

Ok

New

3.2.1 Estrutura inicial dos arquivos

Copie o diretório `AngularBase` e cole como `AngularPlaylist`. Verifique se a pasta `node_modules` contém as bibliotecas do `angular2` e `systemjs`. Em caso negativo, execute o comando `npm install` para atualizar estas bibliotecas.

Abra o diretório `AngularPlaylist` no Visual Studio Code. A primeira alteração que devemos fazer é adicionar a propriedade `watch: true` no arquivo `tsconfig.json`, desta forma, toda vez

que um arquivo for alterado o TypeScript irá recompilar o projeto. Pressione **ctrl+shift+b** para recompilar o projeto (e deixar o *watcher* recompilar as próximas alterações), e execute o *live-server* no terminal para que a página `index.html` seja aberta. A página será aberta com o texto `My First Angular 2 App`.

Para testar se o TypeScript está funcionando juntamente com o *live-server*, abra o arquivo `app/app.component.ts` e altere o *template* do componente para:

`app/app.component.ts`

```
import {Component} from 'angular2/core'

@Component({
  selector: 'my-app',
  template: '<h1>My Playlist</h1>'
})
export class AppComponent { }
```

O *live-server* irá recarregar a página com a nova informação do *template*.

3.2.2 Criando um arquivo de configuração da aplicação

Nesta primeira etapa exibiremos como repassar variáveis estáticas entre as classes da aplicação. Chamamos de **Service** uma classe comum com o propósito de servir outra classe. Neste caso, criaremos o arquivo `app/config.service.ts` com o seguinte código:

`app/config.service.ts`

```
export class Config{
  /**
   * Título da página da aplicação
   */
  static TITLE_PAGE : string = "My Playlist";
}
```

A classe `Config` possui inicialmente a variável `TITLE_PAGE` que é estática e possui o valor `My Playlist`. Para podermos usar esta variável no componente `App`, precisamos primeiro alterar o arquivo `app.component.ts` para:

app/app.component.ts

```
import {Component} from 'angular2/core'

@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>'
})
export class AppComponent {
  title = "My Playlist";
}
```

Ou seja, estamos no *template* usando um recurso chamado **Databind**, e através de {{ }} podemos referenciar uma variável da classe. O valor da variável `title` da classe `AppComponent` será atribuída ao {{title}} do *template*. Existem muitos tipos de **databind** que veremos ao longo desta obra.

Agora precisamos ligar a variável `title` da classe `AppComponent` a variável `TITLE_PAGE` da classe `Config`. Para isso, basta importar a classe e usá-la. Como a variável é estática, o título da página é acessado diretamente pela classe, sem a necessidade de instanciar a classe `Config`.

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Config} from './config.service'

@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>'
})
export class AppComponent {
  title = Config.TITLE_PAGE;
}
```

Veja que o editor de textos (Visual Studio Code) pode lhe ajudar com a complementação de código:

3.2.3 Adicionando bootstrap

Para que a aplicação comece a ganhar uma boa forma visual, precisamos estilizar a aplicação com CSS, e uma das formas de fazer isso é utilizando o bootstrap. No terminal, adicione a biblioteca bootstrap através do seguinte comando:

```
$ npm i bootstrap -S
```

O diretório `node_modules/bootstrap` será adicionado, e podemos incluir a folha de estilos no arquivo `index.html`:

`index.html`

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
```

```
<script src="node_modules/rxjs/bundles/Rx.js"></script>
<script src="node_modules/angular2/bundles/angular2.dev.js"></script>
<link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.m\
in.css">
<script>
System.config({
  packages: {
    app: {
      format: 'register',
      defaultExtension: 'js'
    }
  }
});
System.import('app/boot')
  .then(null, console.error.bind(console));
</script>

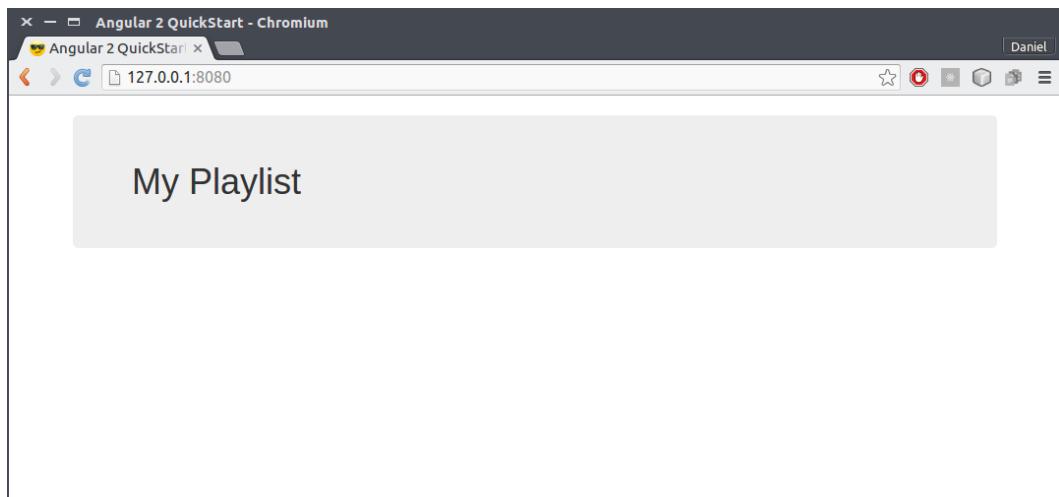
</head>
<body>
  <div class="container">
    <my-app>Loading...</my-app>
  </div>
</body>
</html>
```

Na alteração do arquivo `index.html`, adicionamos a folha de estilos `bootstrap.min.css`, além de criar uma `div` com a classe `container`, adicionando uma margem na página da aplicação. Outra modificação foi no arquivo `app.component.ts`, onde adicionamos uma classe `css` para o título:

app/app.component.ts

```
import {Component} from 'angular2/core'  
import {Config} from './config.service'  
  
@Component({  
    selector: 'my-app',  
    template: '<h1 class="jumbotron">{{title}}</h1>'  
})  
export class AppComponent {  
    title = Config.TITLE_PAGE;  
}
```

Ao adicionarmos `class="jumbotron"` no `<h1>` do componente, temos a aplicação sendo exibida da seguinte forma:



3.2.4 Criando a classe Video

Como a Playlist é sobre vídeos, é natural pensarmos em criar uma classe que representa um vídeo, contendo título, url e descrição. Esta classe se chamará `video.ts`, com o seguinte código:

app/video.ts

```
export class Video{
    id:number;
    title:string;
    url:string;
    desc:string;
    constructor(id:number,title:string,url:string,desc?:string){
        this.id=id;
        this.title=title;
        this.url=url;
        this.desc=desc,
    }
}
```

3.2.5 Criando uma lista simples de vídeos

Com a classe `Video` criada, podemos criar uma lista contendo os vídeos. Esta lista é composta de um *array* com os dados, de uma classe e um *template*. Primeiro, criamos o *array* com os dados, que inicialmente será fixo (não há comunicação com o servidor):

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'

@Component({
    selector: 'my-app',
    template: '<h1 class="jumbotron">{{title}}</h1>'
})
export class AppComponent {
    title = Config.TITLE_PAGE;
    videos : Array<Video>;

    constructor(){
        this.videos = [
            new Video(1,"Test","www.test.com","Test Description"),
            new Video(2,"Test 2","www.test2.com")
        ]
    }
}
```

```
    ]  
}  
}
```

Perceba que voltamos ao `app.component`, importamos a classe `Video` e criamos a variável `videos`. No construtor da classe `app.component` preenchemos o `array` com 2 vídeos. É válido lembrar que para acessar o servidor web e obter dados a estratégia será outra, mas vamos simplificar esta parte inicialmente, para que possamos compreender o funcionamento dos componentes no Angular 2.

3.2.6 Criando sub-componentes

Para que possamos exibir esta lista de vídeos, podemos inserir `` e `` no `template` do `AppComponent` (onde o `<h1>` foi criado), mas esta não é uma boa prática, já que devemos sempre quebrar a aplicação em componentes.



Quebrar uma aplicação em diversos componentes é uma boa prática no Angular 2, respeitando inclusive o princípio da responsabilidade única. Estes princípios se aplicam a qualquer linguagem de programação, [acesse este link¹](#) para maiores detalhes.

Para uma lista de vídeos, podemos criar o componente `VideoListComponent`. Crie o arquivo `app/videolist.component.ts` com o seguinte código inicial:

```
import {Component} from 'angular2/core'  
@Component({  
  selector: 'video-list',  
  templateUrl: 'app/videolist.component.html'  
})  
export class VideoListComponent {  
}
```

Nesta classe usamos a propriedade `templateUrl` ao invés de `template`, apenas para demonstrar que podemos ter um arquivo html de template separado do componente, o que é uma

¹<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

boa prática de programação para o Angular 2. O caminho para o *template* deve ser completo a partir da base da aplicação, por isso o `app/` foi inserido. Nada impede de criar um diretório chamado “templates” na raiz da aplicação, ou dentro do diretório `app`.



Uma outra boa prática de programação é separar cada componente em diretórios, principalmente em projetos com muitos componentes. Como exemplo o componente `VideoList` estaria no diretório `app/components/video`, onde este diretório teria os componentes relativos a informações sobre os vídeos (Lista de vídeos, tela de edição e visualização). O template para `VideoList` estaria em `app/components/video/templates`.

Crie o template do componente `VideoList`, que a princípio possui somente uma mensagem simples:

`app/videolist.component.html`

```
<b> Video List Component </b>
```

Após criar o componente e o template, podemos inserí-lo na aplicação principal `AppComponent`:

`app/app.component.ts`

```
import {Component} from 'angular2/core'  
import {Config} from './config.service'  
import {Video} from './video'  
@Component({  
  selector: 'my-app',  
  template: '<h1 class="jumbotron">{{title}}</h1><video-list></video-list>',  
})  
export class AppComponent {  
  ...  
}
```

Inserimos `<video-list>` no final do template. Se for realizado um teste agora, o componente não será exibido. Isso acontece porque somente o `<video-list>` não faz com que o Angular carregue o componente `VideoListComponent`. É preciso informar ao Angular para carregar o componente, e isso é feito através da propriedade `directives` que deve ser inserida no `@Component`, veja:

app/app.component.ts

```
1 import {Component} from 'angular2/core'
2 import {Config} from './config.service'
3 import {Video} from './video'
4 import {VideoListComponent} from './videolist.component'
5
6 @Component({
7   selector: 'my-app',
8   template: '<h1 class="jumbotron">{{title}}</h1><video-list></video-list>',
9   directives: [VideoListComponent]
10 })
11 export class AppComponent {
12   ...
```

A propriedade `directives` foi adicionada na linha 9, informando a classe `VideoListComponent` com o relativo `import` na linha 4.

3.2.7 Formatando o template

O template do `AppComponent` está descrito em uma linha única, o que pode gerar confusão já que o código html pode ser extenso. Existem duas alternativas para resolver este problema. A primeira é usar o template com várias linhas, usando o caractere acento grave ` ao invés do apóstrofo ' como delimitador, como no exemplo a seguir:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'

@Component({
  selector: 'my-app',
  template: `
    <h1 class="jumbotron">
      {{title}}
    </h1>
  `
```

```
<video-list></video-list>
``,
directives: [VideoListComponent]
})
export class AppComponent {
...

```

Pode-se também utilizar a propriedade `templateUrl` ao invés do `template`, repassando assim a url do arquivo html do template, conforme o exemplo a seguir:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'

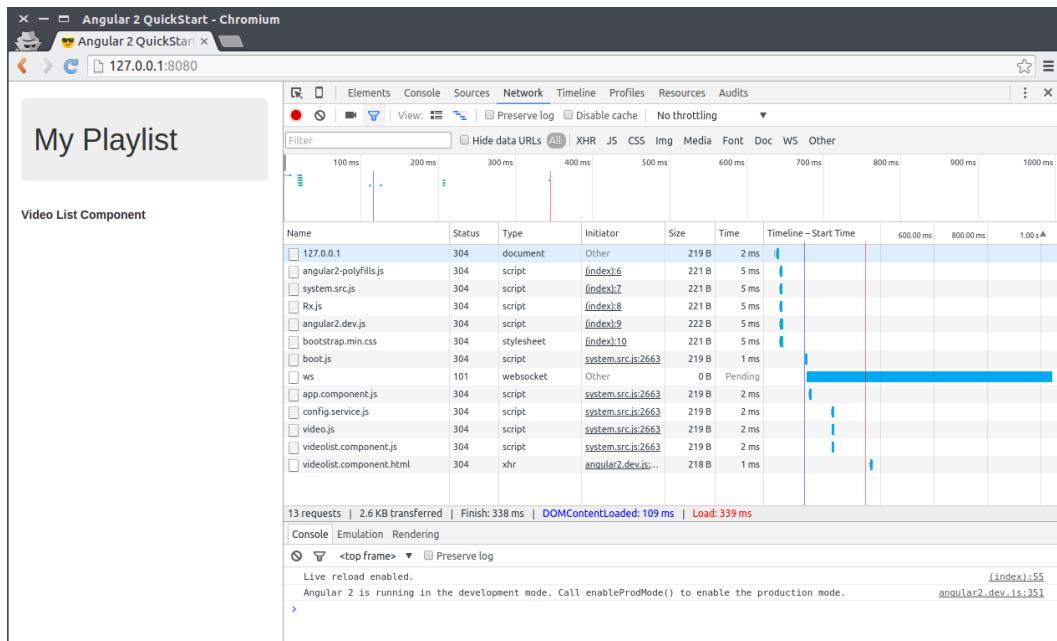
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [VideoListComponent]
})
export class AppComponent {
...

```

app/app.component.html

```
<h1 class="jumbotron">
  {{title}}
</h1>
<video-list></video-list>
```

Após criar o template do `AppComponent`, podemos testar a aplicação, que deve ter a interface semelhante a figura a seguir:



3.2.8 Repassando valores entre componentes

Criamos a variável `videos` no `AppComponent`, com dois itens. Para repassar este array ao componente `VideoList`, usamos o conceito de *binding* do Angular 2. Para repassar um valor de um componente para outro, deve-se criar uma propriedade no seletor do componente utilizando colchetes, como no exemplo a seguir:

`app/app.component.html`

```
<h1 class="jumbotron">
  {{title}}
</h1>
<video-list [videos]="videos"></video-list>
```

Ao adicionarmos `[videos]="videos"`, dizemos que a variável `[videos]` do `VideoListComponent` é igual ao valor da variável `videos` do `AppComponent` (que é o array preenchido no seu construtor). Para criar a variável `videos` no componente `VideoListComponent`, usamos a propriedade `inputs` na configuração do `@Component`, veja:

app/videolist.component.ts

```
import {Component} from 'angular2/core'  
@Component({  
  selector: 'video-list',  
  templateUrl: 'app/videolist.component.html',  
  inputs: ['videos']  
})  
export class VideoListComponent {  
}
```

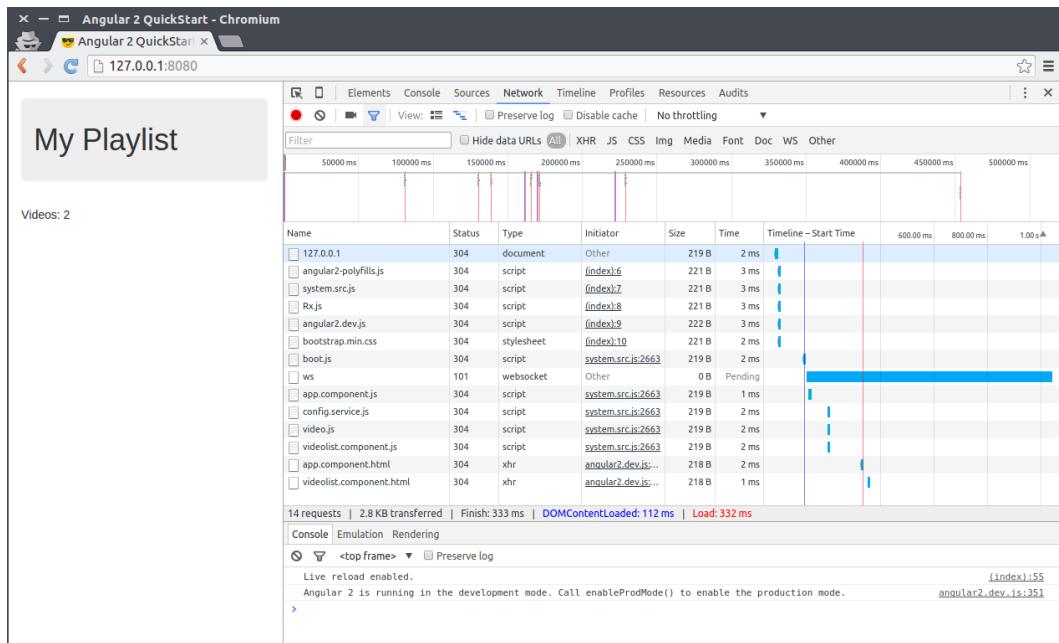
Outra alternativa é criar a variável videos na classe e usar o metadata @input, por exemplo:

```
import {Component} from 'angular2/core'  
@Component({  
  selector: 'video-list',  
  templateUrl: 'app/videolist.component.html'  
})  
export class VideoListComponent {  
  @input() videos;  
}
```

Como o componente VideoListComponent tem a propriedade videos, pode-se alterar o template para usar esta variável. A princípio, podemos alterar o template para:

Videos: {{videos.length}}

Usamos o databind {{videos.length}} para retornar a quantidade de itens da variável videos. Como no construtor de AppComponent foram criados 2 vídeos, o valor será 2, conforme a imagem a seguir:



Ao invés de exibirmos a quantidade de itens do array, vamos alterar o template para desenhar uma tabela com a ajuda do bootstrap:

```
<table class="table">
  <thead>
    <tr>
      <th>ID</th>
      <th>Title</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="#v of videos">
      <td>{{v.id}}</td>
      <td>{{v.title}}</td>
    </tr>
  </tbody>
</table>
```

Para exibir a lista de videos, usamos a diretiva *ngFor no qual o asterisco indica que a tag li faz parte de um elemento mestre/detalhe, ou seja, ela vai se repetir de acordo com a

quantidade de itens. O loop é feito de acordo com `#v of videos`, o que significa que cada item do array `videos` vai ser armazenado na variável `v`. A tralha usada antes da variável `v` indica que ela poderá ser usada como uma variável dentro do loop. Neste caso, usamos `{v.id}` e `{v.title}`.

O resultado com o novo template utilizando o `*ngFor` é apresentado a seguir:

The screenshot shows a browser window with the title "Angular 2 QuickStart - Chromium". The address bar indicates the site is running at "127.0.0.1:8080". The main content area has a header "My Playlist". Below it is a table with two rows of data:

ID	Title
1	Test
2	Test 2

3.2.9 Selecionando um vídeo

Vamos adicionar a funcionalidade de selecionar o vídeo da lista de vídeos. Quando seleciona uma linha da tabela, mostramos o componente `VideoComponent`, que possui uma chamada para o vídeo e três campos de texto contendo o título, url e descrição do mesmo.

3.2.10 Eventos

Para adicionar um evento ao `` que foi gerado, usamos a diretiva `(click)`, da seguinte forma:

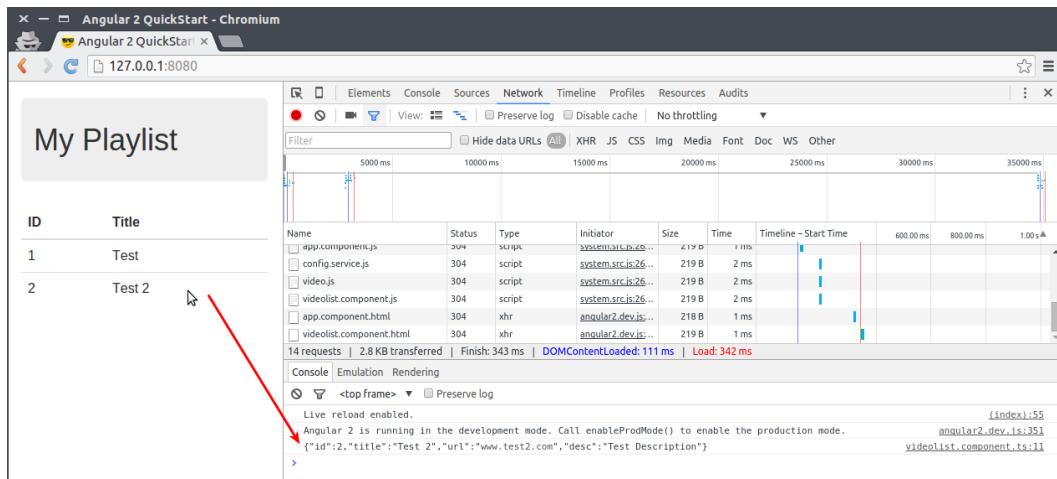
```
<table class="table table-hover">
  <thead>
    <tr>
      <th>ID</th>
      <th>Title</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="#v of videos" (click)="onSelect(v)">
      <td>{{v.id}}</td>
      <td>{{v.title}}</td>
    </tr>
  </tbody>
</table>
```

O método onSelect será executado no VideoListComponent, no qual a princípio podemos apenas emitir um alert para teste:

```
import {Component} from 'angular2/core'
import {Video} from './video'

@Component({
  selector: 'video-list',
  templateUrl: 'app/videolist.component.html',
  inputs: ['videos']
})
export class VideoListComponent {
  onSelect(vid: Video) {
    console.log(JSON.stringify(vid));
  }
}
```

Na classe VideoListComponent criamos o método onSelect com a propriedade vid, que é do tipo Video (incluso o import também). Por enquanto, o método apenas usa o alert para exibir os dados do vídeo selecionado, semelhante a figura a seguir:



3.2.11 Propagando eventos

Quando o usuário clica em um vídeo, deve-se informar ao AppComponent que este evento ocorreu. O componente VideoListComponent não pode chamar diretamente um método do AppComponent, isso é uma má prática de programação. O componente VideoListComponent deve disparar um evento que vai propagar até o componente AppComponent.

Primeiro vamos criar o evento no componente VideoListComponent:

```
import {Component, EventEmitter} from 'angular2/core'
import {Video} from './video'

@Component({
  selector: 'video-list',
  templateUrl: 'app/videolist.component.html',
  inputs: ['videos'],
  outputs: ['selectVideo']
})
export class VideoListComponent {
  selectVideo = new EventEmitter();

  onSelect(vid: Video) {
    this.selectVideo.next(vid);
  }
}
```

```
    }  
}
```



Ao invés de outputs: ['selectVideo'] no @Component, pode-se usar diretamente @output() selectVideo = new EventEmitter()

Perceba que adicionamos no import a classe EventEmitter. Também adicionamos uma nova diretiva ao @Component que é o outputs, indicando que SelectVideo é um evento de saída do componente. No método onSelect, ao invés do console.log usamos a variável selectVideo para disparar o evento, através do método next, repassando o vídeo que o usuário selecionou no evento.

Com o VideoListComponent disparando o evento, podemos voltar ao AppComponent para capturá-lo. Isso será realizado em duas etapas. Primeiro, no template:

```
<h1 class="jumbotron">  
  {{title}}  
</h1>  
<video-list [videos]="videos"  
  (selectVideo)="onSelectVideo($event)">  
</video-list>
```

No template usamos o evento (selectVideo) chamando o callback onSelectVideo, e repassando \$event. Agora precisamos criar o método onSelectVideo na classe AppComponent:

```
import {Component} from 'angular2/core'  
import {Config} from './config.service'  
import {Video} from './video'  
import {VideoListComponent} from './videolist.component'  
  
@Component({  
  selector: 'my-app',  
  templateUrl: 'app/app.component.html',  
  directives: [VideoListComponent]  
})  
export class AppComponent {
```

```
title = Config.TITLE_PAGE;
videos : Array<Video>;
```

```
constructor(){
    this.videos = [
        new Video(1,"Test","www.test.com","Test Description"),
        new Video(2,"Test 2","www.test2.com","Test Description")
    ]
}

onSelectVideo(video){
    console.log(JSON.stringify(video));
}
```

O resultado até agora é o mesmo em termos de interface, mas agora o vídeo selecionado pelo usuário está na classe AppComponent, que é a classe que controla toda a aplicação. O componente VideoListComponent cumpre apenas o papel de mostrar os vídeos e disparar eventos caso um dos itens seja selecionado.



Nos tempaltes, usamos [] para repassar variáveis ao componente, e usamos () para indicar eventos.

3.2.12 Exibindo os detalhes do vídeo

Vamos criar um novo componente, chamado VideoDetailComponent, que a princípio exibe apenas o título do vídeo:

app/videodetail.component.ts

```
import {Component} from 'angular2/core'
import {Video} from './video'

@Component({
  selector: 'video-detail',
  templateUrl: 'app/videodetail.component.html',
  inputs: ['video']
})
export class VideoDetailComponent{}
```

A classe VideoDetailComponent possui o seletor video-detail, o template videodetail.component.html e inputs: ['video'] que será a propriedade que representa o vídeo selecionado.

Neste momento, o template possui apenas a informação do título do livro:

app/videodetail.component.html

```
<h2>{{video.title}}</h2>
```

Agora alteramos a aplicação principal, AppComponent, onde criaremos uma propriedade chamada selectedVideo, que irá controlar o vídeo selecionado pelo evento do VideoListComponent, veja:

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'

@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [VideoListComponent]
})
export class AppComponent {
```

```
title = Config.TITLE_PAGE;
videos : Array<Video>;
selectedVideo: Video;

constructor(){
    this.videos = [
        new Video(1, "Test", "www.test.com", "Test Description"),
        new Video(2, "Test 2", "www.test2.com", "Test Description")
    ]
}

onSelectVideo(video){
    //console.log(JSON.stringify(video));
    this.selectedVideo = video;
}
}
```

Com a propriedade `selectedVideo` preenchida, podemos controlar a visualização do componente `VideoDetailComponent`, através do template da aplicação:

app/app.component.html

```
<h1 class="jumbotron">
    {{title}}
</h1>

<video-list [videos]="videos"
            (selectVideo)="onSelectVideo($event)">
</video-list>

<video-detail
    *ngIf="selectedVideo"
    [video]="selectedVideo">
</video-detail>
```

Neste template, temos a inclusão do `<video-detail>`. Usamos a diretiva `*ngIf` para controlar se o componente é exibido ou não na página. O asterisco é necessário para indicar ao angular

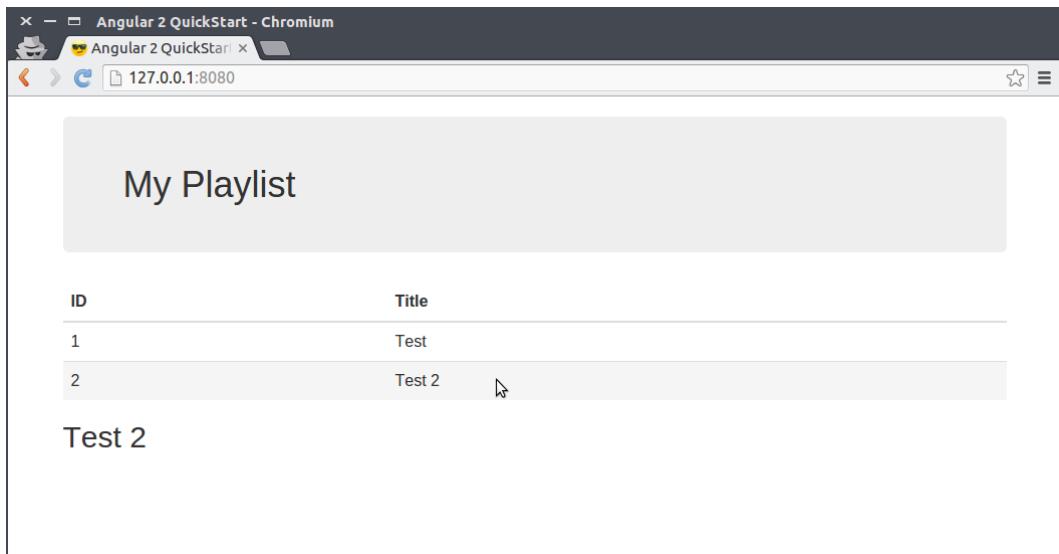
que este controle irá alterar a DOM de alguma forma. Tanto `ngIf` e `ngFor` possuem esta característica.

Em `[video]="selectedVideo"` temos a propriedade `video` do `VideoComponentDetail` recebendo o valor de `selectedVideo` do componente `AppComponent`.

Antes de testar a aplicação, é necessário voltar ao `app.component.ts` e inserir o `VideoDetailComponent` na propriedade `directives` do `@Component`, já que este controle está sendo adicionando pelo template e o Angular precisa que a classe esteja carregada:

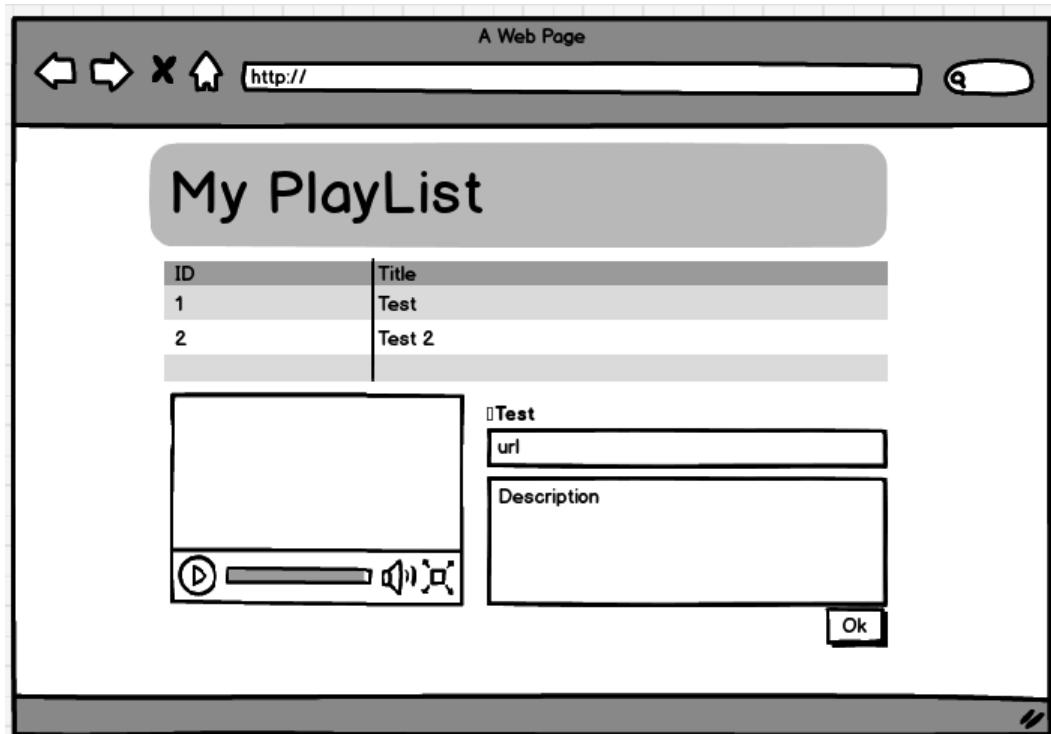
```
import {Component} from 'angular2/core'  
import {Config} from './config.service'  
import {Video} from './video'  
import {VideoListComponent} from './videolist.component'  
import {VideoDetailComponent} from './videodetail.component'  
  
@Component({  
    selector: 'my-app',  
    templateUrl: 'app/app.component.html',  
    directives: [VideoListComponent, VideoDetailComponent]  
})  
export class AppComponent {  
    ...  
}
```

Com o `VideoDetailComponent` adicionado em `directives`, podemos testar a aplicação e clicar nas linhas da tabela de vídeos para que o componente `VideoDetail` seja exibido, conforme a figura a seguir:



3.2.13 Editando os dados do vídeo selecionado

Agora vamos alterar o template do `VideoDetailComponent` para algo mais agradável. O desenho a seguir ilustra como a tela deverá ser desenhada:



Para implementar esta tela, usaremos alguns conceitos do Bootstrap, a biblioteca css que estiliza a aplicação. Também usaremos a diretiva [(model)]="field" que implementa o *Databind* entre o campo de texto e o valor da propriedade.

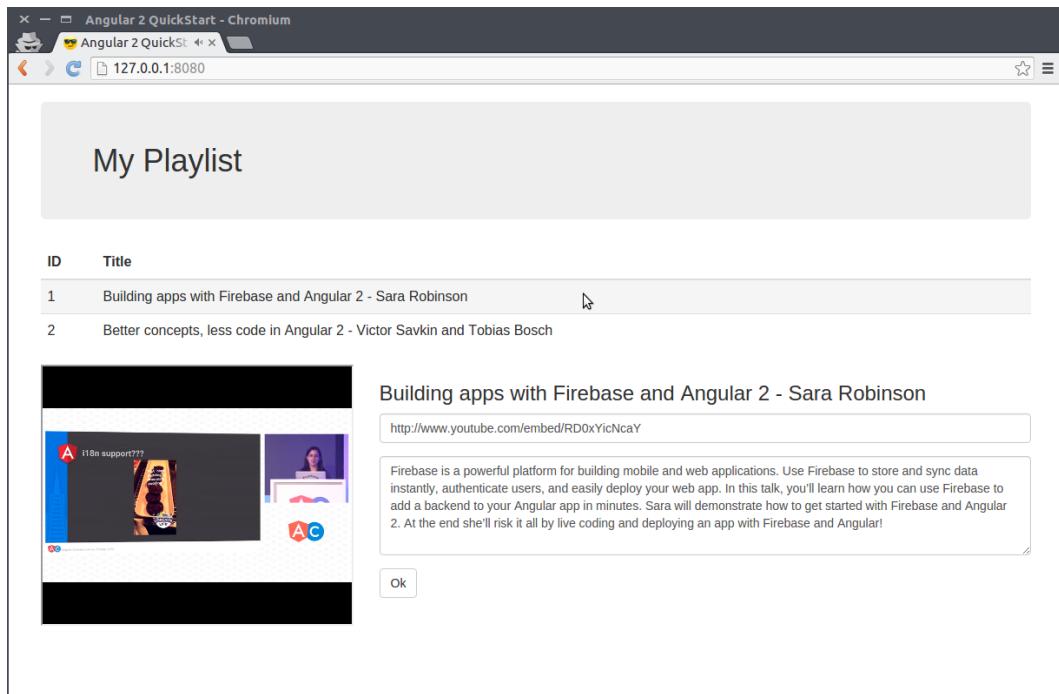
videodetail.component.html

```
<div class="row">
  <div class="col-md-4">
    <iframe width="100%" height="300" src="{{video.url}}>
  </iframe>
</div>
<div class="col-md-8">
  <form>
    <h3>{{video.title}}</h3>
    <div class="form-group">
      <input type="input"
            class="form-control"
            id="url"
```

```
        required
        placeholder="url"
        [(ngModel)]="video.url"
      >
</div>
<div class="form-group">
<textarea class="form-control" rows="5" [(ngModel)]="video.desc">
</textarea>
</div>
<button type="button"
        class="btn btn-default"
        (click)="onButtonOkClick()"
      >Ok</button>
</form>
</div>
</div>
```

Neste formulário os campos `url` e `desc` estão relacionados ao modelo através do recurso de databind do angular, provido pelo uso do `[()]`. Este recurso é chamado de `Two Way DataBind` pois ao alterar o valor da caixa de texto, o mesmo é alterado na propriedade `selectedVideo`, que por sua vez altera o array de vídeos.

O resultado até este momento é exibido a seguir:



3.2.14 Editando o título

Um recurso simples que podemos adotar é usar o evento (`click`) no título do formulário para que o mesmo seja alterado do `<h3>` para o `<input>`. Para controlar este efeito, deve-se criar o campo do título e adicionar uma variável no componente `VideoDetailComponent`, da seguinte forma:

videodetail.component.html

```
<div class="row">
  <div class="col-md-4">
    <iframe width="100%" height="300" src="{{video.url}}>
  </iframe>
</div>
  <div class="col-md-8">
    <form>
      <h3 *ngIf="!editTitle" (click)="onTitleClick()">
        {{video.title}}</h3>
```

```
<div *ngIf="editTitle" class="form-group">
  <input type="input"
    class="form-control"
    id="title"
    required
    placeholder="title"
    [(ngModel)]="video.title"
  >
</div>
<div class="form-group">
  <input type="input"
    class="form-control"
    id="url"
    required
    placeholder="url"
    [(ngModel)]="video.url"
  >
</div>
....
```

A variável `editTitle` irá controlar qual elemento estará visível. Por padrão, o elemento `<h3>` aparece primeiro, e se o usuário clicar no elemento, o evento `onTitleClick` será disparado. No componente, temos:

videodetail.component.ts

```
import {Component} from 'angular2/core'
import {Video} from './video'

@Component({
  selector: 'video-detail',
  templateUrl: 'app/videodetail.component.html',
  inputs: ['video']
})
export class VideoDetailComponent{
  private editTitle:boolean = false;
```

```
onTitleClick(){
    this.editTitle=true;
}
onButtonOkClick(){
    //todo
}
ngOnChanges(){
    this.editTitle=false;
}
}
```

No componente, quando `onTitleClick` é executado, alteramos o valor da propriedade `editTable`, e desta forma o `<h3>` é escondido e o campo `<input>` do título aparece. Adicionamos também o evento `ngOnChanges` que é executado sempre que os dados do componente alteram, ou seja, quando a propriedade `video` é alterada, o evento será disparado e a variável `editTitle` retorna para false.

A última implementação do componente é o botão Ok, que deve fechar o formulário. Na verdade, o botão Ok vai disparar o evento `close` e a aplicação irá tratar o evento.

videodetail.component.ts

```
import {Component, EventEmitter} from 'angular2/core'
import {Video} from './video'

@Component({
    selector: 'video-detail',
    templateUrl: 'app/videodetail.component.html',
    inputs: ['video'],
    outputs: ['closeForm']
})
export class VideoDetailComponent{
    private closeForm = new EventEmitter();
    private editTitle:boolean = false;
    onTitleClick(){
        this.editTitle=true;
    }
    onButtonOkClick(){
        this.closeForm.next({});
    }
}
```

```
    }
    ngOnChanges(){
        this.editTitle=false;
    }
}
```

Para adicionar o evento, importamos a classe `EventEmitter` e criamos o evento chamado `closeForm`. No método `onButtonOkClick`, o evento é disparado.

No `AppComponent` adicionamos o evento no template:

```
<video-detail
    *ngIf="selectedVideo"
    [video]="selectedVideo"
    (closeForm)="onCloseDetailForm($event)"
>
</video-detail>
```

E configuramos o método `onCloseDetailForm` da seguinte forma:

app/app.component.ts

```
imports....
export class AppComponent {
    ...
    onCloseDetailForm(event){
        this.selectedVideo = null;
    }
    ...
}
```

Ao alterar o valor da variável `selectedVideo` para `null`, o componente `VideoDetailComponent` ficará invisível, graças ao `*ngIf="selectedVideo"`.

3.2.15 Criando um novo item

Para finalizar a aplicação, precisamos criar um botão para adicionar um novo vídeo.

app/app.component.html

```
<h1 class="jumbotron">
    {{title}}
</h1>

<video-list [videos]="videos"
    (selectVideo)="onSelectVideo($event)">
</video-list>

<video-detail
    *ngIf="selectedVideo"
    [video]="selectedVideo"
    (closeForm)="onCloseDetailForm($event)"
    >
</video-detail>

<button type="button"
    class="btn btn-default"
    (click)="newVideo()">New</button>
```

Agora basta implementar o método newVideo na classe AppComponent:

```
imports....
export class AppComponent{

    ...
    newVideo(){
        var v : Video = new Video(this.videos.length+1,"A new video");
        this.videos.push(v);
        this.selectedVideo = v;
    }
    ...
}
```

3.2.16 Algumas considerações

Esta pequena aplicação usa os conceitos mais simples do Angular 2. Por exemplo, para preencher os dados da lista de vídeos, usamos um simples array que foi populado no

AppComponent. Em aplicações reais os dados são requisitados ao servidor, e é usado um service para esta tarefa. Isso será abordado em um próximo capítulo.

No formulário também abordamos o básico, mas existem diversas funcionalidades embutidas no Angular 2 que auxiliam o tratamento de erros do formulário, no qual veremos em um capítulo posterior.

3.3 Criando Componentes

Uma das vantagens de trabalhar com componentes no Angular 2 é a possibilidade de reutilização dos mesmos, ou seja, quando criamos um componente podemos usá-lo em qualquer lugar da aplicação. Uma das funcionalidades principais da componentização é a criação de componentes hierárquicos, ou seja, a possibilidade de adicionarmos componentes dentro de outros componentes, de forma livre.

Para o próximo exemplo, copie o diretório “AngularBase” e cole como “AngularPanel”, pois exibiremos um exemplo de como criar um componente do tipo Panel, o mesmo do *Bootstrap*. Após copiar o projeto, adicione o bootstrap pelo comando:

```
$ npm i bootstrap -S
```

Adicione a biblioteca bootstrap no arquivo index.html, conforme o código a seguir:

index.html

```
<html>

  <head>
    <title>Angular 2 QuickStart</title>

    <!-- 1. Load libraries -->
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
  <pt>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.m\in.css">
```

```
<!-- 2. Configure SystemJS -->
<script>
    System.config({
        packages: {
            app: {
                format: 'register',
                defaultExtension: 'js'
            }
        }
    });
    System.import('app/boot')
        .then(null, console.error.bind(console));
</script>

</head>

<!-- 3. Display the application -->
<body>
    <div class="container">
        <my-app>Loading...</my-app>
    </div>
</body>

</html>
```

Vamos alterar o template do AppComponent para o seguinte modelo:

```
import {Component} from 'angular2/core'

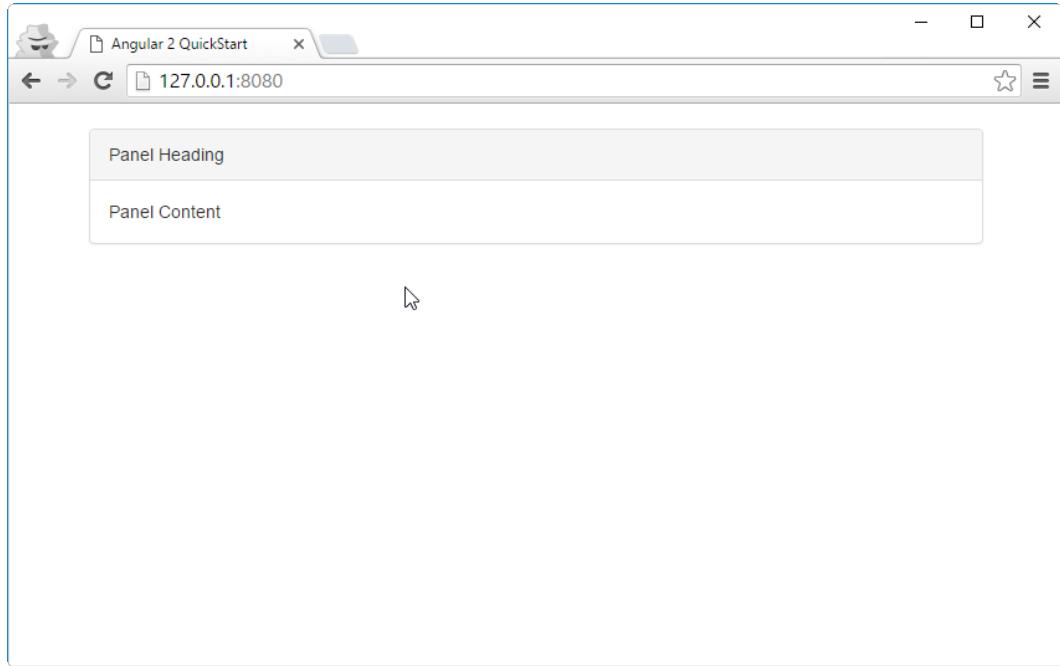
@Component({
    selector: 'my-app',
    templateUrl: 'app/app.component.html'
})
export class AppComponent { }
```

Então incluimos o seguinte template no app.component.html:

```
<br/>

<div class="panel panel-default">
  <div class="panel-heading">Panel Heading</div>
  <div class="panel-body">Panel Content</div>
</div>
```

Até este momento temos um Panel do Bootstrap, semelhante a imagem:



Agora vamos criar o componente `Panel`, no qual irá pertencer ao diretório `container`, e que será uma *library*.

app/container/panel.ts

```
import {Component} from 'angular2/core'

@Component({
  selector: 'panel',
  templateUrl: 'app/container/panel.html'
})
export class Panel { }
```

O template para o Panel é, por enquanto, o mesmo código html de um Panel do bootstrap:

```
<div class="panel panel-default">
  <div class="panel-heading">Panel Heading</div>
  <div class="panel-body">Panel Content</div>
</div>
```

Com o componente criado, podemos adicioná-lo ao componente AppComponent. Deve-se alterar o template:

app/app.component.html

```
<br/>
<panel></panel>
```

E também devemos adicionar o carregamento da diretiva Panel no componente AppComponent:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Panel} from './container'

@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [Panel] // <<<< Add panel to directives
})
export class AppComponent {}
```

Veja que importamos o Panel da seguinte forma:

```
import {Panel} from './container'
```

Isso significa que temos uma library configurada no arquivo container.ts, da seguinte forma:

app/container.ts

```
export * from './container/panel'
```

Ao atualizarmos a aplicação (lembre-se que executar o comando tsc e usar o live-server) veremos a mesma tela, só que agora temos o componente Panel pronto. Agora podemos criar a propriedade title no componente Panel, da seguinte forma:

app/container/panel.ts

```
import {Component} from 'angular2/core'

@Component({
  selector: 'panel',
  templateUrl: 'app/container/panel.html',
  inputs: ['title']
})
export class Panel {}
```

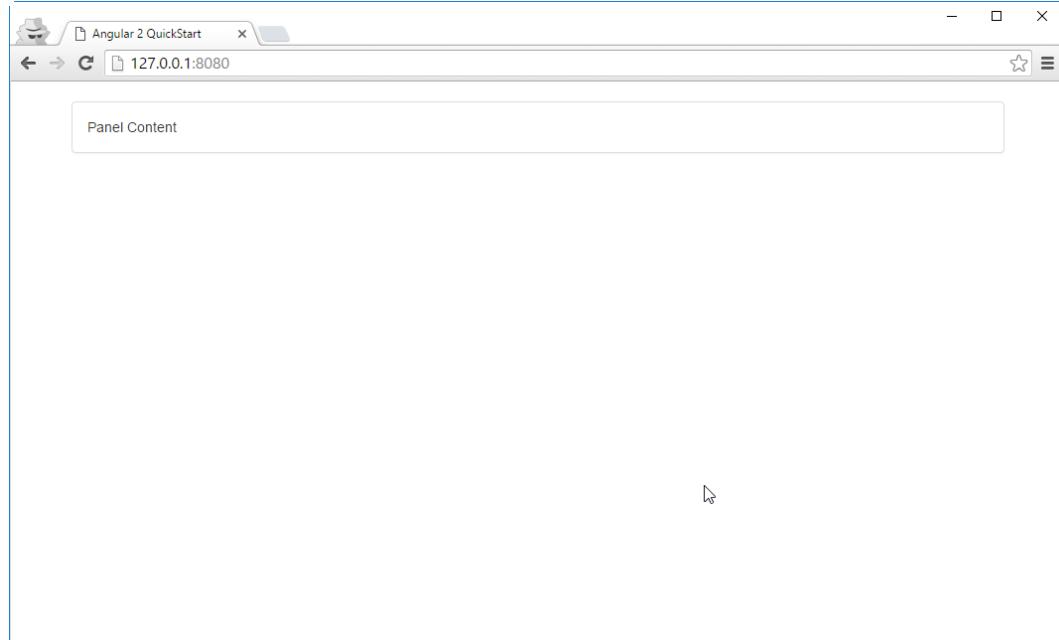
Perceba que incluímos na diretiva inputs do @Component o valor title, configurando assim uma variável de entrada no componente. Podemos usá-lo no template, da seguinte forma:

app/container/panel.html

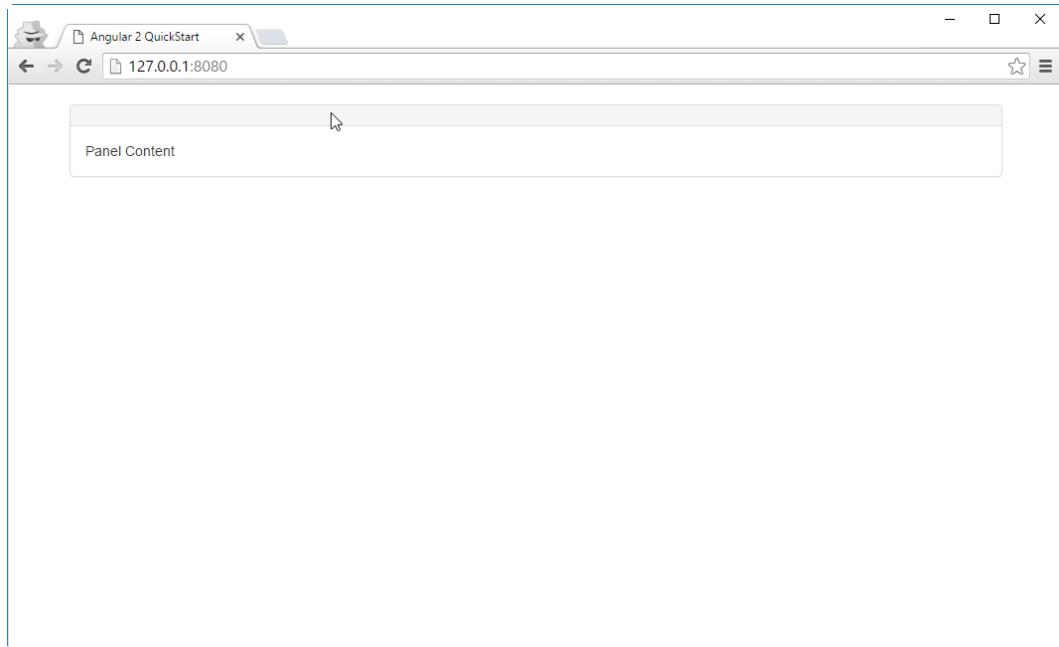
```
<div class="panel panel-default">
  <div class="panel-heading" *ngIf="title">{{title}}</div>
  <div class="panel-body">Panel Content</div>
</div>
```

Aqui usamos a diretiva `*ngIf="title"` configurando que a `<div>` somente será inserida se houver algum valor para a variável `title`. Nesta div, incluímos a variável através do databind `{{title}}`.

Ao recompilarmos a aplicação, temos a seguinte interface:



Se, por exemplo, o `*ngIf` não estivesse presente na `div`, teríamos a seguinte saída:

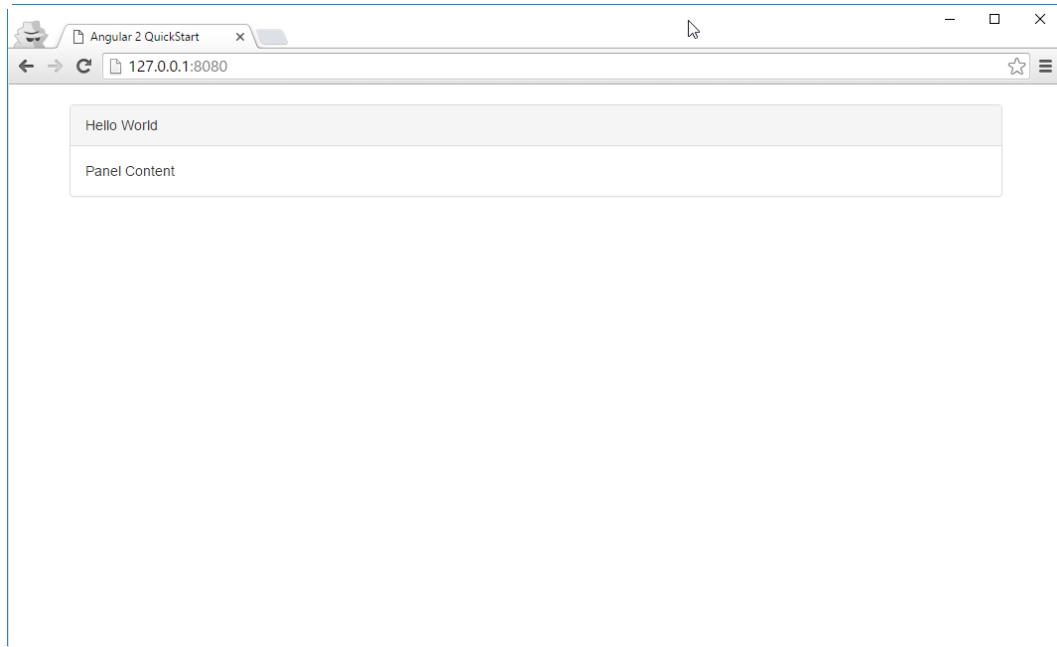


Agora altere o template do AppComponent, da seguinte forma:

app/app.component.html

```
<br/>
<panel title="Hello World"></panel>
```

E a interface fica semelhante a figura a seguir:



3.4 Componentes Hierárquicos

O componente Panel possui uma área onde podemos inserir um texto, ou até mesmo outro componente. Esta “área” é definida pelo elemento `<ng-content></ng-content>`, adicionado no componente Panel:

app/container/panel.html

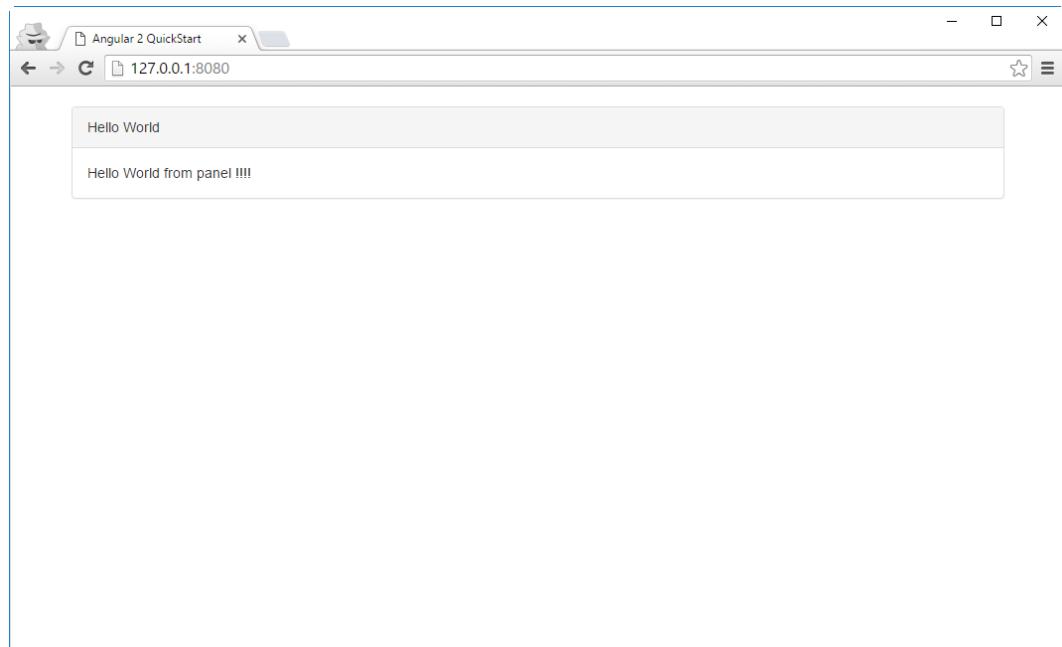
```
<div class="panel panel-default">
  <div class="panel-heading" >{{title}}</div>
  <div class="panel-body"><ng-content></ng-content></div>
</div>
```

No componente principal, temos a seguinte alteração:

app/app.component.html

```
<br/>  
  
<panel title="Hello World">  
    Hello World from panel !!!!  
</panel>
```

Após recompilar os componentes, temos o desenho do Panel semelhante a figura a seguir:



Pode-se inclusive adicionar componentes dentro de componentes, o que permite a criação de telas com a vantagem de reutilização de código html, veja:

app/app.component.html

```
<br/>

<panel title="Hello World">
  <panel title="Step 1">
    Open a terminal
  </panel>
  <panel title="Step 2">
    Say hello world!
  </panel>
</panel>
```

Este template produz o seguinte resultado:

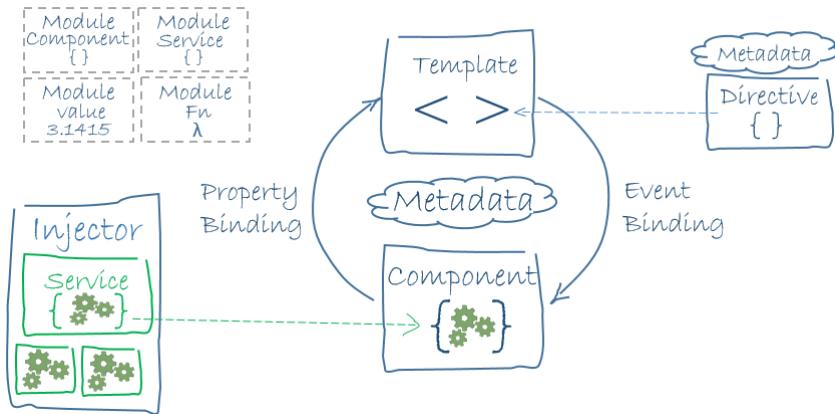


4. Um pouco de teoria

Neste capítulo estaremos revendo um pouco da teoria sobre o Angular 2. Já tivemos um primeiro contato com o framework através do exemplo do capítulo anterior, então agora iremos fixar o que foi aprendido até este momento conhecendo os principais conceitos do framework.

4.1 Visão Geral

Na documentação oficial do framework temos uma imagem que exibe tudo aquilo que você deve conhecer sobre o Angular 2. A imagem é exibida a seguir:



Através desta imagem podemos destacar alguns pontos chave que o leitor deve conhecer para que possa dominar por completo o framework. Estes pontos são:

- Modulo
- Componente
- Template
- Metadado
- Serviço
- Diretivas
- Injeção de dependência

4.2 Módulo (module)

Todas as aplicações do angular podem ser compostas por diversos módulos, que são carregados a medida que os componentes são carregados. Um módulo é compreendido pelo **nome do arquivo** sem a sua extensão.

No exemplo do capítulo anterior (*AngularPlaylist*), temos exemplos de módulos com o mesmo nome da classe, ou seja, temos:

`app.component.ts`

```
export class AppComponent{  
}
```

e

`config.service.ts`

```
export class Config{  
}
```

A palavra `export` configura que a classe será **pública** ao módulo. Após criar a classe pública ao módulo, podemos importá-la, como fizemos em diversos pontos da aplicação:

```
import {Config} from './config.service'
```

O framework Angular 2 também trabalha com módulos, como por exemplo:

```
import {Component} from 'angular2/core'
```

A diferença entre importar um módulo do Angular ou o módulo da aplicação está na definição do diretório. Quando usamos `angular2/core` estamos referenciando o módulo do Angular que está no diretório `node_modules`.

4.2.1 Library Module

Quando criamos várias classes para somente um módulo, estamos criando uma Library (biblioteca), o que é perfeitamente normal no Angular 2. Uma biblioteca pode conter componentes, diretivas, services etc. O uso de bibliotecas não significa que deve-se criar várias classes em um mesmo arquivo, mas que podemos agrupar as suas chamadas a partir de um único ponto.

Suponha uma aplicação Angular em que as classes de modelo (que representado um objeto) estão agrupadas no diretório model, como no exemplo a seguir:

app/model/video.ts

```
export class Video{
    id:number;
    title:string;
    url:string;
    desc:string;
}
```

e

app/model/video.ts

```
import {Video} from './video'
export class Playlist{
    videos:Array<Video>;
    count():number{
        return this.videos.length;
    }
}
```

Na classe principal da aplicação, se quisermos utilizar estas classes do modelo, devemos importá-las da seguinte forma:

app/app.component.ts

```
import {Video} from './model/video'  
import {Playlist} from './model/playlist'
```

Até este momento o código está 100% correto, mas suponha que você deseja agrupar estas classes em uma Library. Então criamos o arquivo `model.ts` e usamos o seguinte código:

app/model.ts

```
export * from './model/video'  
export * from './model/playlist'
```

Com a library `model` criada, a importação das classes `Video` e `Playlist` se resume a:

app/app.component.ts

```
import {Video, Playlist} from './model'
```

O que deixa o código muito mais limpo visualmente.

4.3 Componente (component)

Um componente no Angular é qualquer parte visual da sua aplicação. Na maioria das vezes, um componente possui o *decorator* `@Component`, que contém informações sobre o componente como a sua diretiva, o template, o estilo css e outras. Todas as propriedades são definidas pela sua *api*, disponível [neste link¹](#).

Um componente no Angular 2 possui o conceito de design reativo, isto é, as alterações de visualização no componente não são definidas acessando a DOM diretamente, mas sim alterando os estados do componente. No exemplo do capítulo anterior, quando o usuário clicava na lista de vídeos, surgia então um novo componente na tela exibindo os detalhes daquele vídeo. Para tornar o componente visível na tela, em nenhum momento acessamos a DOM do documento HTML e alteramos a propriedade `display`. Usamos a diretiva `*ngIf` para isso:

¹<https://angular.io/docs/ts/latest/api/core/Component-decorator.html>

```
<video-detail
  *ngIf="selectedVideo"
>
</video-detail>
```

Componentes são a base das aplicações em Angular 2. Sempre que estivermos criando uma aplicação, devemos pensar em como quebrá-la em componentes e fazer com que comuniquem entre si.

4.4 Template

Uma parte importante do componente é o template, que define como ele será desenhado na aplicação. Um template possui código HTML, diretivas, chamada a eventos e também outros templates. Um exemplo completo de template foi visto no capítulo anterior:

```
<h1 class="jumbotron">
  {{title}}
</h1>

<video-list [videos]="videos"
  (selectVideo)="onSelectVideo($event)">
</video-list>

<video-detail
  *ngIf="selectedVideo"
  [video]="selectedVideo"
  (closeForm)="onCloseDetailForm($event)"
>
</video-detail>

<button type="button"
  class="btn btn-default"
  (click)="newVideo()">New</button>
```

Perceba que existem diversas notações no template que definem um comportamento específico. Vamos listar cada um deles a seguir:

4.4.1 Interpolation (Uso de {{ }})

O `{{title}}` é um *databind* do template. Ele vai usar o valor da variável `title` da classe no template. Pode-se usar este *databind* em *tags* e atributos *html*, como no exemplo a seguir:

```
<h3>
  {{title}}
  
</h3>
```

4.4.2 Template Expressions

Além de inserir valores, expressões podem ser usadas para se obter os mais diversos resultados. Pode-se usar `{{1+1}}` para se obter o valor 2, por exemplo. Ou então pode-se usar `{{meuarray.length}}` para obter a quantidades de itens de um array.

4.5 Property Bind

Uma propriedade no componente pode ser ligada a um evento ou método do componente. Vamos voltar a este exemplo específico:

```
<video-detail
  *ngIf="selectedVideo"
  [video]="selectedVideo"
  (closeForm)="onCloseDetailForm($event)"
>
</video-detail>
```

Aqui temos três exemplos de *property bind* que podem ser usados para três situações diferentes:

- O uso o `*ngIf` irá determinar se o componente estará presente ou não na aplicação. O uso do asterisco `*` indica ao Angular que esta propriedade pode alterar a DOM da página. Isso vai alterar a forma como o Angular trata este componente, para que ele seja otimizado.

- O uso do [video]=selectedVideo indica que o valor de selectedVideo será repassado para a variável video do VideoDetailComponent.
- Já (closeForm)="onCloseDetailForm(\$event)" indica um evento que ocorrerá no componente VideoDetailComponent e que executará o método onCloseDetailForm. A propriedade \$event deverá estar sempre presente.

Além destes temos outro chamado de TwoWay DataBind, que é indicado através de [(target)], como no exemplo a seguir:

```
<input type="input"
       [(ngModel)]="video.title">
```

4.5.1 Laços

Outro *template expression* que merece destaque são os laços, no qual usamos a diretiva *ngFor. Perceba o uso do asterisco novamente indicando que esta diretiva altera a DOM da aplicação. A diretiva deve ser inserida no elemento que irá se repetir. No exemplo do capítulo anterior, usamos:

```
<table class="table table-hover">
  <thead>
    <tr>
      <th>ID</th>
      <th>Title</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="#v of videos" (click)="onSelect(v)">
      <td>{{v.id}}</td>
      <td>{{v.title}}</td>
    </tr>
  </tbody>
</table>
```

No elemento <tr> criamos o laço e usamos a expressão #v of videos, o que significa: pegue cada elemento de videos, armazene na variável v e renderize <tr>. Caso deseje utilizar o índice do array, altere a expressão para: *ngFor="#v of videos, #i=index".

4.5.2 Pipes (Operador |)

Um pipe adiciona uma transformação na expressão. O exemplo a seguir ilustra este comportamento:

```
<div>{{ title | uppercase }}</div>
```

Pode-se inclusive formatar uma variável com *json*, por exemplo:

```
<div>{{currentHero | json}}</div>
```

A saída seria:

```
{ "firstName": "Hercules",
  "lastName": "Son of Zeus",
  "birthdate": "1970-02-25T08:00:00.000Z",
  "url": "http://www.imdb.com/title/tt0065832/",
  "rate": 325, "id": 1 }
```

Para formatar uma data, usa-se {{ birthday | date:"MM/dd/yy" }} onde *birthday* é um objeto do tipo Date.

4.6 Metadata (annotation)

Os metadados são usados para fornecer informações a uma classe, no caso do Angular são usados *decorators*, conforme o exemplo a seguir:

```
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
})
export class AppComponent { }
```

Neste exemplo, temos o uso do *selector* que define o seletor que será usado no documento html, ou seja, ao inserirmos <my-app></my-app> no documento html, o componente *AppComponent* será renderizado.

A propriedade *templateUrl* define uma url que indica o caminho do template daquele componente. Pode-se também usar *template* diretamente como metadata, como no exemplo a seguir:

```
@Component({  
  selector: 'my-app',  
  template: ' <h1>My First Angular 2 App</h1>'  
})  
export class AppComponent { }
```

4.7 Serviço (Service)

Uma classe `service` tem como responsabilidades prover dados, sejam eles vindo de um servidor ou não. Na verdade a classe `service` é uma classe TypeScript comum, mas que possui apenas o conceito de manipulação de dados. No capítulo sobre comunicação Ajax veremos mais detalhes de como a classe funciona.

4.8 Injeção de dependência

O conceito de injeção de dependência não é proveniente do Angular 2, mas sim da orientação a objetos. O Angular usa este conceito para facilitar a instanciação de classes pelos componentes, de forma a prover a instância da classe automaticamente.

Uma classe “injetada” fica disponível globalmente à aplicação. Ou seja, se alterarmos a propriedade de uma classe que está injetada, esta alteração continuará ativa se a classe for injetada em outro lugar.

Para “injetar” uma classe em outra, é preciso informá-la no `bootstrap` da aplicação:

```
bootstrap(AppComponent, [ConfigService]);
```

Ou definir a classe no decorador `@Component`:

```
@Component({  
  providers: [ConfigService]  
})  
export class AppComponent { ... }
```

Após a configuração, pode-se usar a seguinte sintaxe para injetar a instância da classe:

```
export class AppComponent{  
    constructor(private _config: ConfigService){ }  
}
```

No capítulo anterior, usamos uma classe de configuração chamada `Config` na qual usamos uma propriedade estática chamada `TITLE_PAGE`. Vamos alterar este comportamento, injetando a classe `Config` no `AppComponent`.

Primeiro, retiramos o `static` da variável `TITLE_PAGE`:

`app/config.service.ts`

```
export class Config{  
    /**  
     * Título da página da aplicação  
     */  
    TITLE_PAGE : string = "My Playlist";  
}
```

Então alteramos o `bootstrap` indicando que a classe `Config` pode ser injetada em qualquer classe da aplicação:

`boot.ts`

```
import {bootstrap} from 'angular2/platform/browser'  
import {Config} from './config.service'  
import {AppComponent} from './app.component'  
  
bootstrap(AppComponent, [Config]);
```

Após adicionar a classe `Config` no `bootstrap`, alteramos a classe `AppComponent`, que ao invés de referenciar o título diretamente, será referenciado no seu construtor, utilizando a classe `Config` injetada.

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Config} from './config.service'
import {Video} from './video'
import {VideoListComponent} from './videolist.component'
import {VideoDetailComponent} from './videodetail.component'

@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html',
  directives: [VideoListComponent, VideoDetailComponent]
})
export class AppComponent {

  title:string;
  videos : Array<Video>;
  selectedVideo: Video;

  constructor(_config:Config){
    this.title = _config.TITLE_PAGE;
    ...
  }
  ...
}
```

4.8.1 Uso do @Injectable()

Em algumas classes onde não há nenhum *metadata* (@Component por exemplo) é preciso adicionar o *metadata* @Injectable() principalmente se esta classe usa injeção de dependência. No exemplo a seguir, a classe TestService gera um erro de execução, pois não existe nenhum *metadata* atrelado a ela, e a mesma usa injeção de dependência com a classe Http:

```
import {Http} from 'angular2/http'

export class TestService{
  constructor(http:Http){
    http.get('.....');
  }
}
```

Para que este problema seja resolvido, basta inserir o metadata `@Injectable()`:

```
import {Http} from 'angular2/http'
import {Injectable} from 'angular2/core'
@Injectable()
export class TestService{
  constructor(http:Http){
    http.get('.....');
  }
}
```

5. Formulários

Neste capítulo tratamos as funcionalidades que o Angular 2 provê no desenvolvimento de um formulário web. Formulários são usado para entrada de dados, e é necessário controlar como esta entrada é realizada e se os dados informados pelo usuário são válidos.

Com Angular 2, é possível prover as seguintes funcionalidades:

- Databind entre objetos e campos do formulário
- Capturar as alterações no formulário
- Validar entrada de forma coerente
- Capturar eventos
- Exibir mensagens de erro

5.1 Criando o projeto inicial

Neste capítulo, iremos copiar a aplicação `AngularBase` e colar como `AngularForms`. Após copiar a aplicação, adicione a biblioteca bootstrap:

```
$ npm i bootstrap --D
```

Após a instalação do Bootstrap pelo `npm`, alteramos o arquivo `index.html` para:

`index.html`

```
<html>

  <head>
    <title>Angular 2 QuickStart</title>

    <!-- 1. Load libraries -->
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
```

```
<script src="node_modules/rxjs/bundles/Rx.js"></script>
<script src="node_modules/angular2/bundles/angular2.dev.js"></script>
<link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.m\in.css">

<!-- 2. Configure SystemJS -->
<script>
System.config({
  packages: {
    app: {
      format: 'register',
      defaultExtension: 'js'
    }
  }
});
System.import('app/boot')
  .then(null, console.error.bind(console));
</script>

</head>

<!-- 3. Display the application -->
<body>
  <div class="container">
    <my-app>Loading...</my-app>
  </div>
</body>

</html>
```

Antes de criarmos o primeiro formulário, vamos criar a classe Person que ficará disponível na Library model. Para isso, devemos criar o arquivo app/model/person.ts e o arquivo app/model.ts, da seguinte forma:

app/model/person.ts

```
export class Person {
  constructor(
    public id: number,
    public name: string,
    public email: string,
    public birthdate?:string
  ) { }
}
```

Esta é uma classe simples que define quatro campos, sendo que o campo `birthdate` é opcional. A classe `model.ts` exporta a classe `Person`, formando assim uma library:

app/model.ts

```
export * from './model/Person'
```

Agora criamos um simples service chamado “Mock”, que conterá a informação que será exibida no formulário, veja:

app/mock.ts

```
import {Person} from './model'
export class Mock{
  public mike = new Person(1,"Mike","mike@gmail");
}
```

E vamos preparar a classe `Mock` para ser injetada em outras classes. Isso é realizado no bootstrap:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {Mock} from './mock'

bootstrap(AppComponent, [Mock]);
```

Com o service Mock pronto para ser usado, podemos retornar a classe AppComponent e usá-lo da seguinte forma:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Mock} from './mock'
import {Person} from './model'

@Component({
  selector: 'my-app',
  template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent {
  user:Person;

  constructor(_mock:Mock){
    this.user = _mock.mike;
  }
}
```

Com esta configuração podemos criar um simples formulário e adicioná-lo ao componente AppComponent, alterando a propriedade template para templateUrl, com o seguinte template:

app/app.component.html

```
<form>
  <input type="hidden" id="id" name="id"/>
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" required>
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input type="text" class="form-control" required>
  </div>
  <button type="submit" class="btn btn-default">Submit</button>
</form>
```

Até este momento os dados do Mock ainda não foram exibidos, isso porque não configuramos o *Two-Way Databind*. Para configurá-lo, devemos usar a seguinte sintaxe: `[(ngModel)]="model.name"`. No formulário, temos:

app/app.component.html

```
<form>
  <input type="hidden" id="id" name="id" [(ngModel)]="user.id"/>
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control"
           required [(ngModel)]="user.name">
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input type="text" class="form-control"
           required [(ngModel)]="user.email">
  </div>
  <button type="submit" class="btn btn-default">Submit</button>
</form>
```

Somente com esta modificação já temos os dados sendo exibidos no formulário:

The image shows a user interface for a form. At the top, there is a label 'Name' followed by a text input field containing the value 'Mike'. Below it is another label 'Email' followed by a text input field containing 'mike@gmail.com'. At the bottom of the form is a single button labeled 'Submit'.

Uma simples forma de analisar a variável `user` da classe `AppComponent` é usar o seguinte template acima do formulário:

app/app.component.html

```
{ {user | json} }

<form>
  <input type="hidden" id="id" name="id" [(ngModel)]="user.id"/>
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control"
           required [(ngModel)]="user.name">
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input type="text" class="form-control"
           required [(ngModel)]="user.email">
  </div>
  <button type="submit" class="btn btn-default">Submit</button>
</form>
```

Este template irá exibir a variável `user` e o pipe “`|`” irá formatar em json. O resultado é semelhante a figura a seguir:

```
{ "id": 1, "name": "Mike CHANGEEE", "email": "mike@gmail" }
```

Name

Email

Submit

Perceba que, ao alterar o valor da caixa de texto, o valor do objeto user altera automaticamente.

5.2 Uso do ngControl

Permitir o data-bind do controle é apenas uma das funcionalidades que o *Angular* provê. Também é possível rastrear o estado de cada componente permitindo assim que se saiba quando um componente foi alterado, quando está inválido etc. Além de checar o estado atual do controle, é possível adicionar estilos *css* à ele, de forma que podemos destacar os estados do controle, como um erro por exemplo.

Para habilitar esta validação, basta usar a diretiva `ngControl`, repassando a propriedade a ser observada, veja:

app/app.component.html

```
 {{user | json}}
```

```
<form>
```

```
  <input type="hidden" id="id" name="id" [(ngModel)]="user.id"/>
```

```
  <div class="form-group">
```

```
    <label for="name">Name</label>
```

```
    <input type="text" class="form-control"
```

```
      required [(ngModel)]="user.name"
```

```
      ngControl="name"
```

```
    >
```

```
  </div>
```

```
  <div class="form-group">
```

```
    <label for="email">Email</label>
```

```
<input type="text" class="form-control"
       required [(ngModel)]="user.email"
       ngControl="email"
     >
</div>
<button type="submit" class="btn btn-default">Submit</button>
</form>
```

Ao adicionarmos o `ngControl`, este passa a observar o controle, e controlar os estilos CSS do campo de acordo com o comportamento do controle. Por exemplo, o campo `name` possui a instrução `required`, que define o campo como obrigatório. Se este campo não estiver preenchido, a classe CSS `ng-invalid` será adicionada à ele. Podemos então usar um pouco de CSS para exibir uma informação ao usuário. Primeiro, crie o arquivo `styles.css` com o seguinte código:

`styles.css`

```
.ng-invalid {
  border-left: 5px solid #a94442;
}
```

Depois, adicione o arquivo `styles.css` no `<head>` do `index.html`:

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>

    <!-- 1. Load libraries -->
    ...
    <link rel="stylesheet" href="styles.css">

    <!-- 2. Configure SystemJS -->
    ...
```

Após adicionar o estilo, recompile a aplicação e acesse o formulário, retire o texto do campo Name e acesse o campo Email. Perceba que o estilo foi aplicado, com uma borda vermelha, conforme a imagem a seguir:

```
{ "id": 1, "name": "", "email": "mike@gmail" }  
Name  
  
Email  
  
Submit
```

Volte a preencher o campo e veja que o estilo foi removido.

5.3 Exibindo uma mensagem de erro

Para exibir uma mensagem de erro, é necessário criar uma `<div>` cuja visibilidade possa ser controlada através de uma variável de controle. Para criar esta variável, use a `*` seguida do nome da variável e do seu valor. No caso do campo `name`, temos:

```
<div class="form-group">  
  <label for="name">Name</label>  
  <input type="text" class="form-control"  
    required [(ngModel)]="user.name"  
    ngControl="name"  
    #name="ngForm"  
  >  
  <div [hidden]="name.valid" class="alert alert-danger">  
    Name is required  
  </div>  
</div>
```

Perceba que criamos `#name="ngForm"`, ou seja, criamos a variável `name` no campo `input` apontando para `ngForm`, que é o formulário em questão. Depois criamos uma `div` e usamos a diretiva `[hidden]` para controlar a sua visibilidade. O resultado desta implementação é exibida a seguir:

The screenshot shows a simple web form with two fields: 'Name' and 'Email'. The 'Name' field is empty and has a red border, with a red error message 'Name is required' displayed below it. The 'Email' field contains the value 'mike@gmail.com' and has a green border. Below the form is a small navigation icon.

Name

Name is required

Email

mike@gmail.com

Submit

5.4 Desabilitando o botão de submit do formulário

É possível desabilitar o botão de *Submit* do formulário caso existe algum problema com o formulário. Para isso, precisamos criar uma variável que representa o `ngForm`, isso é feito na tag `<form>`:

```
<form #f="ngForm">
```

No botão `submit` podemos usar a diretiva `[disabled]` em conjunto com a variável `#f` criada:

```
<button [disabled]="!f.valid"
  type="submit"
  class="btn btn-default">Submit</button>
```

Quando o formulário estiver inválido por algum motivo, `f.valid` será falso e `!f.valid` verdadeiro, desabilitando o botão, conforme a imagem a seguir:

Name
Name is required

Email
mike@gmail.com

Submit

5.5 Submit do formulário

Quando o usuário clicar no botão submit, o mesmo será postado, o que não é utilizado pelo Angular 2, já que toda a aplicação geralmente é armazenada em uma única página. Geralmente, quando um formulário é submetido, usamos Ajax para enviar estas informações ao servidor.

No Angular 2 podemos usar a diretiva (`ngSubmit`) repassando um método que será executado no componente, como no exemplo a seguir:

app/app.component.html

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
```

No componente, temos:

app/app.component.ts

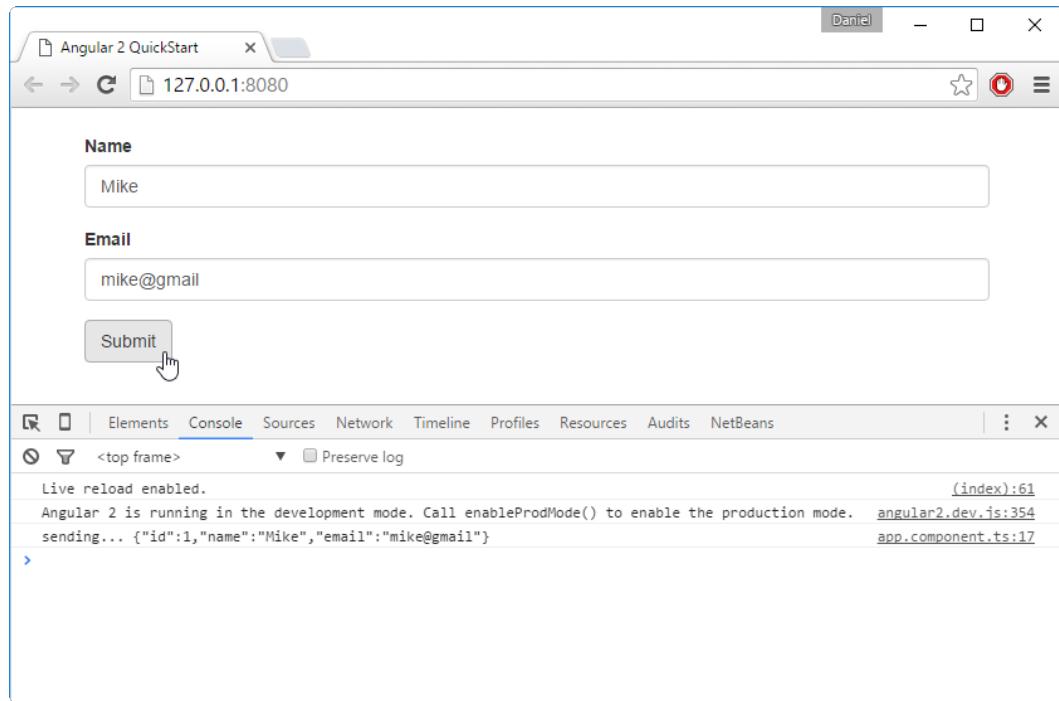
```
import {Component} from 'angular2/core'  
import {Mock} from './mock'  
import {Person} from './model'  
  
@Component({  
  selector: 'my-app',  
  templateUrl: 'app/app.component.html'  
})  
export class AppComponent {
```

```
user:Person;

constructor(_mock:Mock){
    this.user = _mock.mike;
}

onSubmit(f){
    console.log("sending..." + JSON.stringify(user));
}
}
```

Ao clicar no botão `submit`, temos uma resposta semelhante a figura a seguir:



5.6 Controlando a visibilidade do formulário

Uma prática simples que ajuda o usuário a obter uma resposta ao envio do formulário é criar uma variável de controle que indica se o formulário foi enviado. Com esta variável podemos

usar `[hidden]` para definir o que estará visível ao usuário. O exemplo a seguir ilustra este processo:

app/app.component.html

```
<div [hidden]="submitted">
  <form>
    ...
  </form>
</div>
<div [hidden]="!submitted">
  Sending... {{user|json}}
</div>
```

A variável `submitted` é controlada no componente `AppComponent`:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {Mock} from './mock'
import {Person} from './model'

@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent {
  user:Person;
  submitted:boolean;

  constructor(_mock:Mock){
    this.submitted = false;
    this.user = _mock.mike;
  }

  onSubmit(f){
    this.submitted = true;
    console.log("sending... " + JSON.stringify(this.user));
  }
}
```

}
}

6. Conexão com o servidor

Toda aplicação real necessita persistir dados em um banco de dados, e no desenvolvimento web é usado uma arquitetura no estilo cliente/servidor, ou seja, a aplicação Angular acessa o servidor através de uma url, que por sua vez responde ao cliente.

O servidor web pode ser constituído de qualquer linguagem (java, php, .Net), mas a comunicação entre o servidor e o cliente deve seguir um padrão de dados. Esta padrão é o *json*, um formato de dados em texto que pode representar um objeto ou um array de objetos.

6.1 Criando o projeto

Copie o diretório AngularBase e cole como AngularHttp. O primeiro teste que faremos é acessar uma url qualquer e obter dados via JSON. Podemos, a princípio, criar o seguinte arquivo no projeto recém criado:

users.json

```
[  
  {"id":1,"name":"Mike"},  
  {"id":2,"name":"Joe"},  
  {"id":3,"name":"Adam"}]
```

Após criar o arquivo, execute o ‘live-server’ e accesse a seguinte url: ‘<http://localhost:8080/users.json>’, obtendo a seguinte resposta:



6.2 Uso da classe Http

Para que seja possível utilizar a classe `Http` para acessar uma url via ajax, é necessário realizar os seguintes procedimentos:

- Adicionar o arquivo `http.dev.js` na lista de scripts do arquivo `index.html`:

```
<!-- 1. Load libraries -->
<script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>
<script src="node_modules/rxjs/bundles/Rx.js"></script>
<script src="node_modules/angular2/bundles/angular2.dev.js"></script>
<script src="node_modules/angular2/bundles/http.dev.js"></script>
```

- Adicionar os seguintes *imports* no componente que usa `Http`

```
import {Http, HTTP_PROVIDERS} from 'angular2/http'  
import 'rxjs/add/operator/map'
```

Perceba que estamos importando também `rxjs/add/operator/map` que é necessário na última versão do Angular (beta).

- Adicionar no `@Component` a propriedade `HTTP_PROVIDERS`, importada anteriormente que define diversas classes do Angular que podem ser injetadas no componente.

```
@Component({  
    ...  
    providers: [HTTP_PROVIDERS],  
    ...  
})
```

Como abordado anteriormente, a propriedade `providers` pode ser substituída no bootstrap da aplicação, no arquivo `boot.ts`, tornando a classe `Http` disponível para qualquer classe da aplicação.

- Injete a classe `Http` no componente:

```
export class AppComponent {  
  
    constructor(private _http:Http){  
  
    }  
}
```

Pode-se inclusive usar uma variável de classe, como por exemplo:

```
export class AppComponent {  
    private _http;  
    constructor(_http:Http){  
        this._http = _http;  
    }  
}
```

ou somente:

```
export class AppComponent {  
    constructor(private _http:Http){  
        this._http = _http;  
    }  
}
```

Neste primeiro exemplo, iremos usar `_http` somente no construtor.

Com a classe devidamente injetada, vamos usá-la para acessar o servidor via ajax. Como usamos o `live-server`, ao usar o endereço `./users.json`, será acessado uma URL semelhante a esta: `http://localhost:8080/users.json`.

Para acessar este endereço via o método GET, usamos:

```
constructor(_http:Http){  
    _http.get("./users.json").....  
}
```

Ou seja, usamos `_http.get(url)` para acesso a url e quando o servidor retorna com as informações, chamamos dois métodos que tem funcionalidades distintas. O primeiro é o método `map` que irá formatar o texto de retorno do servidor e o segundo é o `subscribe` que irá atribuir o texto formatado a uma variável qualquer. Resumindo, temos:

```
export class AppComponent {  
  users:Array<any>;  
  constructor(_http:Http){  
    _http.get("./users.json")  
      .map(res => res.json())  
      .subscribe(users => this.users = users);  
  }  
}
```

Neste exemplo, o método `map` formata a resposta `res` em json, isso é possível porque o servidor retorna um texto no formato json. O método `subscribe` atribuiu a resposta já formatada em json para a variável `this.users`, criada na classe.



Pode-se, ao invés de atribuir o valor à variável, chamar um método da própria classe, por exemplo: `.subscribe(users => this.onGetUsers(users));`

Com a variável `this.users` devidamente preenchida, podemos usar o template para exibir os dados. No exemplo a seguir, temos a classe `AppComponent` completa:

```
import {Component} from 'angular2/core'  
import {Http, HTTP_PROVIDERS} from 'angular2/http'  
import 'rxjs/add/operator/map'  
  
@Component({  
  selector: 'my-app',  
  providers: [HTTP_PROVIDERS],  
  template: `  
    <ul>  
      <li *ngFor="#u of users">  
        {{u.id}} - {{u.name}}  
      </li>  
    </ul>  
`  
)  
export class AppComponent {  
  users:Array<any>;
```

```
constructor(_http:Http){  
    _http.get("./users.json")  
        .map(res => res.json())  
        .subscribe(users => this.users = users);  
}  
}
```

A novidade na classe é o template, que realiza um `ngFor` no elemento `` através da variável `users`, que é preenchida pelo `Http`. O resultado deste componente é exibido a seguir:

The screenshot shows a browser window titled "Angular 2 QuickStart" at the URL "127.0.0.1:8080". The page content displays a list of users:

- 1 - Mike
- 2 - Joe
- 3 - Adam

Below the browser window is the "Network" tab of the developer tools. The timeline shows a single request to "users.json" with a duration of approximately 100ms. The "Response" tab shows the JSON data returned:

```
[{"id": 1, "name": "Mike"}, {"id": 2, "name": "Joe"}, {"id": 3, "name": "Adam"}]
```

The "Headers" tab shows the following headers:

Name	Value
Content-Type	application/json
Date	Mon, 11 Jul 2016 14:44:11 GMT
Content-Length	113

6.3 Utilizando services

No exemplo anterior usamos a classe `Http` diretamente no componente, acessando o servidor e obtendo o array de `users`, que foi atribuído à variável `users`. Vamos refatorar o projeto para colocar cada funcionalidade em seu devido lugar.

Uma classe **service** é responsável em prover dados a aplicação, então fica natural criarmos a classe `UserService` que terá a responsabilidade de acessar o servidor e retornar com os dados.

6.4 Organização do projeto

Neste contexto, entra uma observação sobre a organização de suas classes no projeto. Perceba que temos três alternativas a escolher, sendo elas:

1. Criar todas as classes da aplicação no diretório `app`
2. Criar o diretório `services` e o arquivo `user.ts` neste diretório. Criar a library `services` no diretório `app`.
3. Criar o diretório `user` e o arquivo `user.service.ts` neste diretório. Criar a library `services` no diretório `app`.

A pior opção em termos de organização é a 1a, use-a somente quando estiver aprendendo ou testando algo muito pequeno. Um projeto em Angular 2 divide-se em muitos arquivos e armazená-los em somente um diretório `app` não é viável.

A opção 2 é boa para projetos pequenos, pois assim temos `services/classes/componentes/templates/estilos` organizados em cada lugar, por exemplo:

```
app
|- components
  |- user.component.ts
  |- product.component.ts
  |- employee.component.ts
|- templates
  |- user.html
  |- product.html
```

```
| - employee.html  
|-  
|-services  
| - user.ts  
| - product.ts  
| - employee.ts  
|-  
|-styles  
| - app.css  
| - user.css  
| - product.css  
| - employee.css  
|-services.ts  
|-app.component.ts  
|-boot.ts
```

....

Perceba que nesta organização os arquivos que representam as entidades da aplicação estão misturados. Ou seja, para que você trabalhe com products deverá abrir diversos arquivos em vários diretórios. Isso se torna custoso quando temos muitos arquivos em um mesmo diretório. A segunda opção é melhor que a primeira, mas funciona bem apenas em projetos pequenos.

A 3a opção consiste em separar as entidades da aplicação em diretórios distintos, da seguinte forma:

```
app  
|-user  
| - user.service.ts  
| - user.component.ts  
| - user.template.html  
| - user.style.css  
|-product  
| - product.service.ts  
| - product.component.ts  
| - product.template.html  
| - product.style.css
```

```
| -employee
|   | - employee.service.ts
|   | - employee.component.ts
|   | - employee.template.html
|   | - employee.style.css
|-services.ts
|-app.component.ts
|-boot.ts
```

Na terceira opção, separamos cada entidade em um diretório, adicionando os arquivos relacionados àquela entidade. Desta forma, quando o projeto tornar-se demasiadamente grande, teremos muitos diretórios, mas cada um deles representa um pedaço único da aplicação.

6.5 Model user

Antes de criarmos a classe service, perceba que no app.component a variável users é do tipo `Array<any>`, ou seja, um array de objetos de qualquer tipo. Vamos melhorar um pouco este código criando um objeto que representa o user, no qual chamaremos de model User.

app/user/user.ts

```
export class User{
    constructor(
        public id:number,
        public name:string
    ){}
}
```

Nesta classe usamos `public id` dentro do `constructor` criando assim uma variável pública. Em projetos maiores é possível encapsular estas variáveis. Ao criar o model User, podemos adicioná-la a library model, da seguinte forma:

app/model.ts

```
export * from './user/user'
```

6.6 Service user

Podemos criar a classe UserService da seguinte forma:

app/user/user.service.ts

```
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map'
import {User} from '../model'

@Injectable()
export class UserService {
    constructor(private http: Http) { }

    public getUsers() {
        return this.http
            .get('./users.json')
            .map(res => res.json());
    }
}
```

Nesta classe, temos os imports que já vimos no início do capítulo. Uma novidade é o import do User que aplica um caminho diferente: `../model`. O uso do `.. /` sobe um nível de diretório para encontrar a classe `model.ts`.

Usamos `@Injectable()` para configurar a injeção de dependência na classe. Com isso, será possível injetar a classe `Http` na classe `UserService`. Quando criamos o construtor, injetamos a instância da classe `Http` na variável `http`, que pode ser usada em outros métodos da classe. Para que possamos injetar a classe `Http`, devemos usar o metadata providers, ou alterar o bootstrap da aplicação. Como `Http` é uma classe que será injetada em diversas outras classes, vamos adicioná-la no bootstrap da aplicação:

```
import {bootstrap} from 'angular2/platform/browser'  
import {AppComponent} from './app.component'  
import {Http, HTTP_PROVIDERS} from 'angular2/http'  
  
bootstrap(AppComponent, [HTTP_PROVIDERS]);
```

Voltando a classe `UserService`, temos o método `getUsers()`:

```
public getUsers() {  
    return this.http  
        .get('./users.json')  
        .map(res => res.json());  
}
```

Este método usa a classe `Http` para fazer uma requisição GET ao endereço `./users.json`, e na resposta do servidor o método `map` é chamado, onde usamos o método `.json()` para formatar a resposta que é o texto `[{id:1,name:'mike'}....]` em Json.

Perceba que estamos retornando o `this.http.get(...).map(...)` para quem chamou o método `getUsers()`. O código que chama `getUsers()` que fica responsável em tratar este retorno. Isso é realizado no `AppComponent`.

Para criar uma library de services, criamos o seguinte arquivo:

`app/service.ts`

```
export * from './user/user.service'
```

6.7 Alterando o componente `AppComponent`

Agora o `AppComponent` irá usar a classe `UserService`, da seguinte forma:

app/app.component.ts

```
import {Component} from 'angular2/core'
import {User} from './model'
import {UserService} from './service'

@Component({
  selector: 'my-app',
  providers: [UserService],
  template: `
    <ul>
      <li *ngFor="#u of users">
        {{u.id}} - {{u.name}}
      </li>
    </ul>
  `
})
export class AppComponent {
  public users:Array<User>;
  constructor(userService:UserService){
    userService.getUsers()
      .subscribe(users => this.users = users);
  }
}
```

No AppComponent importamos a classe UserService e a usamos como um *provider* no @Component. Isso significa que podemos utilizar injeção de dependência no construtor. O construtor irá chamar o método userService.getUsers() que irá retornar uma instância da classe Observable, na qual podemos utilizar o método subscribe para preencher o valor da variável this.users.

6.8 Enviando dados

Para enviar dados do cliente ao servidor, faremos uma requisição POST da seguinte forma:

```
@Injectable()
export class UserService {

    ...
    public addUser(u:User){
        return this.http
            .post('./addUser', JSON.stringify(u))
            .map(res => res.json());
    }
    ...
}
```

Ao invés de `http.get`, usamos `http.post` e no segundo parâmetro do `.post` repassamos os dados que serão enviados ao servidor. Como ainda não estamos tratando dados no servidor, veremos mais exemplos de alteração de dados nos próximos capítulos.

7. Routes

A definição de rotas no Angular 2 segue o conceito de divisão da sua aplicação em partes menores que podem ser carregadas via Ajax, pelo próprio framework.

7.1 Aplicação AngularRoutes

Copie a aplicação AngularBase e cole como AngularRoutes. O primeiro procedimento que deve ser realizado é adicionar a inclusão do arquivo `router.dev.js` no arquivo `index.html`, da seguinte forma:

`AngularRoutes/index.html`

```
<html>

  <head>
    <title>Angular 2 QuickStart</title>

    <!-- 1. Load libraries -->
    <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="node_modules/rxjs/bundles/Rx.js"></script>
    <script src="node_modules/angular2/bundles/angular2.dev.js"></script>
    <script src="node_modules/angular2/bundles/router.dev.js"></script>
    <base href="/">

    <!-- 2. Configure SystemJS -->
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
    </script>
  </head>
  <body>
    <h1>Welcome to angular2!</h1>
    <p>This is a sample web application. You can edit the code below or in <a href="https://github.com/Microsoft/TypeScript-Angular2-Quickstart">https://github.com/Microsoft/TypeScript-Angular2-Quickstart</a> and reload this page to see your changes reflected in the browser.
    </p>
  </body>
</html>
```

```
        }
    }
});
System.import('app/boot')
    .then(null, console.error.bind(console));
</script>

</head>

<!-- 3. Display the application -->
<body>
    <my-app>Loading...</my-app>
</body>

</html>
```



Perceba que existem os arquivos com a extensão “.dev.js” e “.min.js”. Quando a aplicação está em desenvolvimento, usamos a extensão “.dev” para que erros de programação sejam facilmente identificados. Em ambiente de produção, usamos “.min” para obter uma melhor performance.



Importante: Inclua também `<base href="/">` dentro da seção `<head>` do documento html.

Após incluir o Router, deve-se adicionar a variável ROUTER_PROVIDERS no boot da aplicação, da seguinte forma:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {ROUTER_PROVIDERS} from 'angular2/router'

bootstrap(AppComponent, [ROUTER_PROVIDERS]);
```

Através destas duas configurações, o Router é habilitado na aplicação.

7.2 Dividindo a aplicação em partes

Uma aplicação pode ser dividida em diversas partes, como por exemplo: home, login e dashboard. Cada parte possui seu componente e template, que são configurados através do Router.

Primeiro, crie os seguintes componentes:

- HomeComponent
- LoginComponent
- DashboardComponent

```
import {Component} from 'angular2/core'
@Component({
  templateUrl: 'app/home/home.html'
})
export class HomeComponent { }
```

```
import {Component} from 'angular2/core'
@Component({
  templateUrl: 'app/login/login.html'
})
export class LoginComponent { }
```

```
import {Component} from 'angular2/core'  
@Component({  
  templateUrl: 'app/dashboard/dashboard.html'  
})  
export class DashboardComponent { }
```

Crie os templates de cada componente somente com o nome de cada componente.

7.3 Criando a área onde os componentes serão carregados

É preciso definir uma área onde os componentes serão carregados. Esta definição é feita com a tag <router-outlet>, da seguinte forma:

app/app.component.ts

```
import {Component} from 'angular2/core'  
import {ROUTER_DIRECTIVES} from 'angular2/router'  
  
@Component({  
  selector: 'my-app',  
  directives: [ROUTER_DIRECTIVES],  
  template: `  
    <h1>My First Angular 2 App</h1>  
    <div><router-outlet></router-outlet></div>  
`  
})  
export class AppComponent { }
```

7.4 Configurando o router

No componente app.component vamos configurar o roteamento através do metadata @RouteConfig. Várias modificações devem ser feitas no app.component, veja:

```
import {Component} from 'angular2/core'
import {ROUTER_DIRECTIVES, RouteConfig} from 'angular2/router'
import { HomeComponent } from './home/home.component'
import { DashboardComponent } from './dashboard/dashboard.component'
import { LoginComponent } from './login/login.component'

@Component({
  selector: 'my-app',
  directives: [ROUTER_DIRECTIVES],
  template: `
    <h1>My First Angular 2 App</h1>
    <div><router-outlet></router-outlet></div>
  `
})
@RouteConfig([
  { path: '/home', name: 'Home', component: HomeComponent,
    useAsDefault: true },
  { path: '/login', name: 'Login', component: LoginComponent },
  { path: '/dashboard', name: 'Dashboard', component: DashboardComponent }
])
export class AppComponent { }
```

Os *imports* devem incluir ROUTER_DIRECTIVES e RouteConfig, além dos componentes que criamos: HomeComponent, DashboardComponent e LoginComponent.

No @RouteConfig definimos as rotas, onde cada uma delas possui pelo menos três parâmetros:

- path: é o caminho que deverá “casar” com a url.
- name: é o nome da rota.
- component: é o componente que será carregado

Ainda existe a propriedade useAsDefault: true que define a rota padrão. Para testar as rotas, vamos criar um pequeno menu que aponta para os três componentes que criamos.

7.5 Criando links para as rotas

Vamos criar um pequeno menu que contém os links para as rotas. A princípio este menu fica no próprio template do AppComponent, veja:

app/app.component.ts

```
@Component({
  selector: 'my-app',
  directives: [ROUTER_DIRECTIVES],
  template: `
    <h1>My First Angular 2 App</h1>
    <ul>
      <li><a [routerLink]="/Home">Home</a></li>
      <li><a [routerLink]="/Login">Login</a></li>
      <li><a [routerLink]="/Dashboard">Dashboard</a></li>
    </ul>
    <div><router-outlet></router-outlet></div>
  `
})
```

Usamos `[routerLink]` para definir um link para a rota, onde devemos informar o nome da rota.

7.6 Repassando parâmetros

É possível repassar parâmetros entre as rotas. Primeiro, deve-se adicionar o parâmetro no link do Router, como no exemplo a seguir:

```
{ path: '/user/:1', name: 'User', component: UserComponent },
```

Para repassar o parâmetro, pode-se usar o `[routerLink]` da seguinte forma:

```
<a [routerLink]="'[User', {id:1}]">Princess Crisis</a>
```

Ou então através do TypeScript (com um router injetado na classe):

```
this._router.navigate( ['User', { id: user.id }]);
```

Para obter o valor do parâmetro repassado, é preciso utilizar a classe `RouteParams`:

```
import {RouteParams, Router} from 'angular2/router'

@Component({.....})
export class SomeClass{
  constructor(
    private _router: Router,
    private _routeParams: RouteParams
  ) { }
  ngOnInit() {
    var id = this._routeParams.get('id');
  }
}
```

8. Exemplo Final - Servidor

Após revermos todos os conceitos relevantes do Angular 2, vamos criar um exemplo funcional de como integrar o Angular 2 a uma api. O objetivo desta aplicação é criar um simples blog com *Posts* e *Usuários*, onde é necessário realizar um login para que o usuário possa criar um Post.

8.1 Criando o servidor RESTful

Usaremos uma solução 100% Node.js, utilizando as seguintes tecnologias:

- **express**: É o servidor web que ficará responsável em receber as requisições web vindas do navegador e respondê-las corretamente. Não usaremos o *live-server*, mas o **express** tem a funcionalidade de autoloading através da biblioteca **nodemon**.
- **body-parser**: É uma biblioteca que obtém os dados JSON de uma requisição POST.
- **mongoose**: É uma adaptador para o banco de dados MongoDB, que é um banco NoSql que possui funcionalidades quase tão boas quanto a um banco de dados relacional.
- **jsonwebtoken**: É uma biblioteca node usada para autenticação via web token. Usaremos esta biblioteca para o login do usuário.

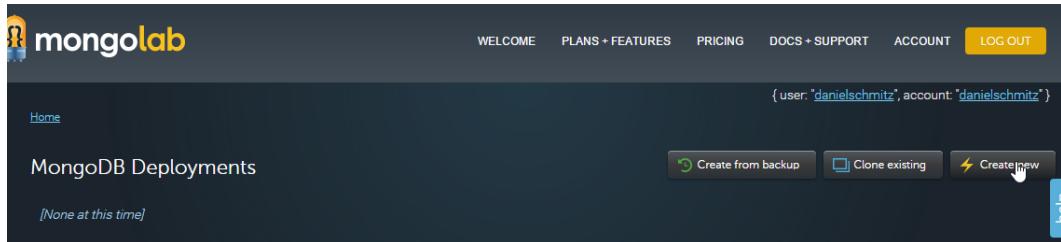
Todas estas tecnologias podem ser instaladas via **npm**, conforme será visto a seguir.

8.2 O banco de dados MongoDB

O banco de dados MongoDB possui uma premissa bem diferente dos bancos de dados relacionais (aqueles em que usamos SQL), sendo orientados a documentos auto contidos (NoSql). Resumindo, os dados são armazenados no formato JSON. Você pode instalar o MongoDB em seu computador e usá-lo, mas nesta obra estaremos utilizando o serviço <https://mongolab.com/>¹ que possui uma conta gratuita para bancos públicos (para testes).

¹<https://mongolab.com/>

Acesse o link <https://mongolab.com/welcome/>² e clique no botão Sign Up. Faça o cadastro no site e logue (será necessário confirmar o seu email). Após o login, na tela de administração, clique no botão Create New, conforme a imagem a seguir:



Na próxima tela, escolha a aba Single-node e o plano Sandbox, que é gratuito, conforme a figura a seguir:

A screenshot of the Mongolab Plan selection screen. At the top, it says "Plan (view pricing page) :". Below that, there are two tabs: "Single-node" (highlighted with a blue background and white text) and "Replica set cluster". A red arrow points to the "Single-node" tab. Underneath the tabs, it says "These plan(s) are perfect for development/testing/staging environments as well as for utility instances that do not require high-availability." Then, it shows the "Standard Line" section with the heading "Standard Line". It says "The most economical plans for production applications running on AWS. Plans come standard with 2 data nodes plus an arbiter node." A red arrow points to the "Standard Line" heading. Below that, there's a table of plans:

Plan	Storage	Cost
<input checked="" type="radio"/> Sandbox (shared, 0.5 GB)		FREE
<input type="radio"/> M3 Single-node (7.5 GB, 120 GB SSD block storage)		\$420
<input type="radio"/> M4 Single-node (15 GB, 240 GB SSD block storage)		\$835
<input type="radio"/> M5 Single-node (34.2 GB, 480 GB SSD block storage)		\$1310
<input type="radio"/> M6 Single-node (68.4 GB, 700 GB SSD block storage)		\$2045

Ainda nesta tela, forneça o nome do banco de dados. Pode ser blog e clique no botão Create new MongoDB deployment. Na próxima tela, com o banco de dados criado, acesse-o e verifique se a mensagem “A database user is required....” surge, conforme a imagem a seguir:

²<https://mongolab.com/welcome/>

[Home](#)**Database: blogdb****To connect using the mongo shell:**

```
% mongo ds045465.mongolab.com:45465/blogdb -u <dbuser> -p <dbpassword>
```

To connect using a driver via the standard MongoDB URI ([what's this?](#)):

```
mongodb://<dbuser>:<dbpassword>@ds045465.mongolab.com:45465/blogdb
```



A database user is required to connect to this database. [Click here](#) to create a new one.



Clique no link e adicione um usuário qualquer (login e senha) que irá acessar este banco, conforme a imagem a seguir:

The screenshot shows the Mongolab database interface for the 'blogdb' database. At the top, there are connection instructions:

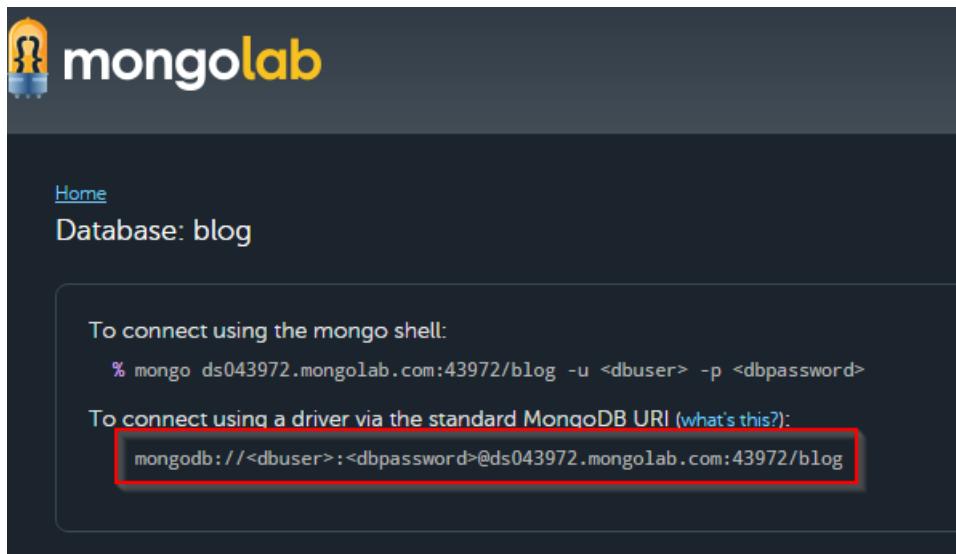
- To connect using the mongo shell:
% mongo ds045465.mongolab.com:45465/blogdb -u <dbuser> -p <dbpassword>
- To connect using a driver via the standard MongoDB URI ([what's this?](#)):
mongodb://<dbuser>:<dbpassword>@ds045465.mongolab.com:45465/blogdb

Below these are navigation tabs: Collections, Users (which is highlighted in blue), Stats, Backups, and Tools.

Database Users

NAME	READ ONLY?
user	false

Após criar o usuário, iremos usar a URI de conexão conforme indicado na sua tela de administração:



8.3 Criando o projeto

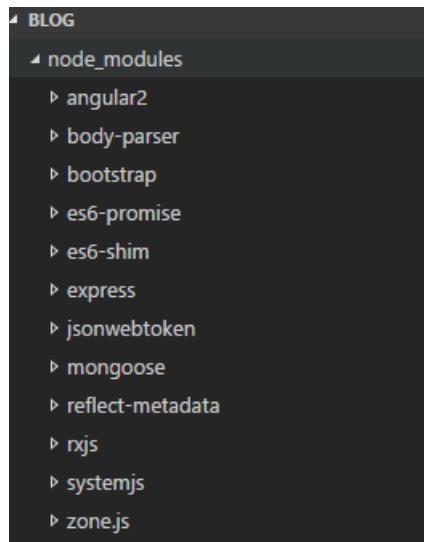
Crie o diretório “blog”, e inicialize o arquivo de configuração de projetos com o seguinte comando:

```
\blog $ npm init
```

Após inserir as informações sobre o projeto, o arquivo `package.json` é criado. Vamos usar o `npm` para instalar todas as bibliotecas que utilizaremos no projeto:

```
$ npm i express body-parser jsonwebtoken mongoose angular2 systemjs bootstrap\np --S
```

Após a instalação de todas as bibliotecas, a pasta `node_modules` deverá possuir os seguintes projetos:



8.4 Estrutura do projeto

A estrutura do projeto está focada na seguinte arquitetura:

```
blog
| - .vscode (Configurações do Visual Studio Code)
| - node_modules (Módulos do node da aplicação)
| - app    (Aplicação - arquivos TypeScript)
| - model (Arquivos de modelo do banco de dados MongoDB)
| - public (Pasta pública com os arquivos estáticos do projeto)
|     | -- index.html (Página html principal da aplicação)
| - api   (Pasta virtual com a api da aplicação)
| - package.json
| - tsconfig.json (configura como os arquivos TypeScript são compilados)
| - server.js (Servidor Web Express)
```

A pasta app conterá a aplicação Angular 2 em TypeScript, sendo que quando a aplicação for compilada, ela será copiada automaticamente para a pasta public. Isso significa que os arquivos com a extensão .ts estarão localizados na pasta app e os arquivos com a extensão .js e .js.map estarão localizados na pasta public.

8.5 Configurando os modelos do MondoDB

O Arquivo `server.js` contém tudo que é necessário para que a aplicação funcione como uma aplicação web. Iremos explicar passo a passo o que cada comando significa. Antes, vamos abordar os arquivos que representam o modelo MongoDB:

/model/user.js

```
var mongoose      = require('mongoose');
var Schema        = mongoose.Schema;

var userSchema = new Schema({
  name:  String,
  login: String,
  password: String
});

module.exports = mongoose.model('User', userSchema);
```

O modelo User é criado com o auxílio da biblioteca mongoose. Através do Schema criamos um modelo (como se fosse uma tabela) chamada User, que possui os campos name, login e password.

A criação dos Posts é exibida a seguir:

/model/post.js

```
var mongoose      = require('mongoose');
var Schema        = mongoose.Schema;

var postSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  user: {type: mongoose.Schema.Types.ObjectId, ref: 'User'},
  date: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Post', postSchema);
```

Na criação do modelo Post, usamos quase todos os mesmos conceitos do User, exceto pelo relacionamento entre Post e User, onde configuramos que o Post o possui uma referência ao modelo User.

8.6 Configurando o servidor Express

Crie o arquivo `server.js` para que possamos inserir todo o código necessário para criar a *api*. Vamos, passo a passo, explicar como este servidor é configurado.

`server.js`

```
1 var express = require('express');
2 var app = express();
3 var bodyParser = require('body-parser');
4 var jwt = require('jsonwebtoken');
```

Inicialmente referenciamos as bibliotecas que usaremos no decorrer do código. Também é criada a variável `app` que contém a instância do servidor *express*.

`server.js`

```
5 //secret key (use any big text)
6 var secretKey = "MySuperSecretKey";
```

Na linha 6 criamos uma variável chamada `secretKey` que será usada em conjunto com o módulo `jsonwebtoken`, para que possamos gerar um token de acesso ao usuário, quando ele logar. Em um servidor de produção, você deverá alterar o `MySuperSecretKey` por qualquer outro texto.

server.js

```
7 //Database in the cloud
8 var mongoose = require('mongoose');
9 mongoose.connect('mongodb://USER:PASSWORD@URL/blog', function (err) {
10   if (err) { console.error("error! " + err) }
11 });



---


```

Importamos a biblioteca mongoose na linha 8 e usamos o comando connect para conectar no banco de dados do serviço mongolab. Lembre de alterar o endereço de conexão com o que foi criado por você.

server.js

```
12 //bodyparser to read json post data
13 app.use(bodyParser.urlencoded({ extended: true }));
14 app.use(bodyParser.json());



---


```

Nas linhas 13 e 14 configuramos o bodyParser, através do método use do express, que está representado pela variável app. O bodyParser irá obter os dados de uma requisição em JSON e formatá-los para que possamos usar na aplicação.

server.js

```
15 //Load mongodb model schema
16 var Post = require('./model/post');
17 var User = require('./model/user');



---


```

Nas linhas 16 e 17 importamos os models que foram criados e que referenciam Post e User. Estes esquemas (“Schema”) serão referenciados pelas variáveis Post e User.

server.js

```
15 //create a REST ROUTER
16 var router = express.Router();



---


```

A linha 16 cria o router, que é a preparação para o express se comportar como uma API. Um router é responsável em obter as requisições e executar um determinado código dependendo do formato da requisição. Geralmente temos quatro tipos de requisição:

- GET: Usada para obter dados. Pode ser acessada pelo navegador através de uma URL.
- POST: Usada para inserir dados, geralmente vindos de um formulário.
- DELETE: Usado para excluir dados
- PUT: Pode ser usado para editar dados. Não iremos usar PUT neste projeto, mas isso não lhe impede de usá-lo.

Além do tipo de requisição também temos a url e a passagem parâmetros, que veremos mais adiante.

server.js

```
17 //Static files
18 app.use('/', express.static(__dirname+'/public'));
19 app.use('/libs', express.static(__dirname+'/node_modules/bootstrap/dist'));
20 app.use('/libs', express.static(__dirname+'/node_modules/systemjs/dist'));
21 app.use('/libs', express.static(__dirname+'/node_modules/rxjs/bundles/'));
22 app.use('/libs', express.static(__dirname+'/node_modules/angular2/bundles'));
```

Na linha 18 configuramos o diretório public como estático, ou seja, todo o conteúdo neste diretório será tratado como um arquivo que, quando requisitado, deverá ser entregue ao requisitante. Este conceito é semelhante ao diretório “webroot” de outros servidores web. Veja que, no caso do express, os diretórios que não pertencem ao express.static não poderão ser acessados pelo navegador web. Isso melhora a segurança do servidor inibindo a exibição de arquivos que devem ser executados somente no servidor, como o arquivo model/user.js, por exemplo.

Também usamos a variável __dirname que retorna o caminho completo até o arquivo server.js. Isso é necessário para um futuro deploy da aplicação em servidores “reais”.

Nas linhas 19 à 22, configuramos o diretório estático /libs no qual irá apontar para as bibliotecas javascript do diretório node_modules. Estamos adicionando as bibliotecas do Bootstrap, SystemJS, rxjs e Angular 2, todas elas que serão adicionadas no arquivo public/index.html do projeto, no próximo capítulo.

server.js

```
23 //middleware: run in all requests
24 router.use(function (req, res, next) {
25     console.warn(req.method + " " + req.url +
26                   " with " + JSON.stringify(req.body));
27     next();
28 });
```

Na linha 24 criamos uma funcionalidade chamada “middleware”, que é um pedaço de código que vai ser executado em toda a requisição que o express receber. Na linha 25 usamos o método `console.warn` para enviar uma notificação ao console, exibindo o tipo de método, a url e os parâmetros Json. Esta informação é usada apenas em ambiente de desenvolvimento, pode-se comentá-la em ambiente de produção. O resultado produzido na linha 24 é algo semelhante ao texto a seguir:

```
POST /login with {"login":"foo","password":"bar"}
```

O método `JSON.stringify`, caso não conheça, obtém um objeto JSON e retorna a sua representação no formato texto.

Na linha 27 usamos o método `next()` para que a requisição continue o seu fluxo.

server.js

```
29 //middleware: auth
30 var auth = function (req, res, next) {
31     var token = req.body.token || req.query.token
32             || req.headers['x-access-token'];
33     if (token) {
34         jwt.verify(token, secretKey, function (err, decoded) {
35             if (err) {
36                 return res.status(403).send({
37                     success: false,
38                     message: 'Access denied'
39                 });
40             } else {
41                 req.decoded = decoded;
```

```
42         next();
43     }
44   });
45 }
46 else {
47   return res.status(403).send({
48     success: false,
49     message: 'Access denied'
50   });
51 }
52 }
```

Na linha 30 temos outro middleware, chamado de auth, que é um pouco mais complexo e tem como objetivo verificar se o token fornecido pela requisição é válido. Quando o usuário logar no site, o cliente receberá um token que será usado em toda a requisição. Esta forma de processamento é diferente em relação ao gerenciamento de sessões no servidor, muito comum em autenticação com outras linguagens como PHP e Java.

Na linha 31 criamos a variável token que tenta receber o conteúdo do token vindo do cliente. No caso do blog, sempre que precisamos repassar o token ao servidor, iremos utilizar o cabeçalho http repassando a variável x-access-token.

Na linha 34 usa-se o método jwt.verify para analisar o token, repassado pelo cliente. Veja que a variável secretKey é usada neste contexto, e que no terceiro parâmetro do método verify é repassado um *callback*.

Na linha 35 verificamos se o *callback* possui algum erro. Em caso positivo, o token repassado não é válido e na linha 36 retornarmos o erro através do método res.status(403).send() onde o status 403 é uma informação de acesso não autorizado (Erro http, assim como 404 é o *not found*).

Na linha 40 o token é válido, pois nenhum erro foi encontrado. O objeto decodificado é armazenado na variável req.decoded para que possa ser utilizada posteriormente e o método next irá continuar o fluxo de execução da requisição.

A linha 46 é executada se não houver um token sendo repassado pelo cliente, retornando também um erro do tipo 403.

server.js

```
53 //simple GET / test
54 router.get('/', function (req, res) {
55     res.json({ message: 'hello world!' });
56});
```

Na linha 54 temos um exemplo de como o router do express funciona. Através do método `router.get` configuramos a url “/”, que quando chamada irá executar o *callback* que repassamos no segundo parâmetro. Este *callback* configura a resposta do router, através do método `res.json`, retornando o objeto `{ message: 'hello world!' }`.

server.js

```
56 router.route('/users')
57     .get(auth, function (req, res) {
58         User.find(function (err, users) {
59             if (err)
60                 res.send(err);
61             res.json(users);
62         });
63     })
64     .post(function (req, res) {
65         var user = new User();
66         user.name = req.body.name;
67         user.login = req.body.login;
68         user.password = req.body.password;
69
70         user.save(function (err) {
71             if (err)
72                 res.send(err);
73             res.json(user);
74         })
75     });

```

Na linha 56 começamos a configurar o roteamento dos usuários, que será acessado inicialmente pela url “/users”. Na linha 57 configuramos uma requisição GET à url /users,

adicionando como *middleware* o método `auth`, que foi criado na linha 29. Isso significa que, antes de executar o `callback` do método “GET /users” iremos verificar se o token repassado pelo cliente é válido. Se for válido, o `callback` é executado e na linha 58 usamos o schema `User` para encontrar todos os usuários do banco. Na linha 61 retornamos este array de usuário para o cliente.

Na linha 64 configuramos o método POST `/users` que tem como finalidade cadastrar o usuário. Perceba que neste método não usamos o middleware `auth`, ou seja, para executá-lo não é preciso estar autenticado. Na linha 65 criamos uma variável que usa as propriedades do “Schema” `User` para salvar o registro. Os dados que o cliente repassou ao express são acessados através da variável `req.body`, que está devidamente preenchida graças ao `body-parser`.

O método `user.save` salva o registro no banco, e é usado o `res.json` para retornar o objeto `user` ao cliente.

server.js

```
76 router.route('/login').post(function (req, res) {
77   if (req.body.isNew) {
78     User.findOne({ login: req.body.login }, 'name')
79       .exec(function (err, user) {
80         if (err) res.send(err);
81         if (user != null) {
82           res.status(400).send('Login Existente');
83         }
84       else {
85         var newUser = new User();
86         newUser.name = req.body.name;
87         newUser.login = req.body.login;
88         newUser.password = req.body.password;
89         newUser.save(function (err) {
90           if (err) res.send(err);
91           var token = jwt.sign(newUser, secretKey, {
92             expiresIn: "1 day"
93           });
94           res.json({ user: newUser, token: token });
95         });
96       }
97     });
}
```

```
98     } else {
99         User.findOne({ login: req.body.login,
100             password: req.body.password }, 'name')
101         .exec(function (err, user) {
102             if (err) res.send(err);
103             if (user != null) {
104                 var token = jwt.sign(user, secretKey, {
105                     expiresIn: "1 day"
106                 });
107                 res.json({ user: user, token: token });
108             }else{
109                 res.status(400).send('Login/Senha incorretos');
110             }
111         });
112     }
113 });


---


```

Na linha 76 temos a funcionalidade para o Login, acessado através da url /login. Quando o cliente faz a requisição “POST /login” verificamos na linha 77 se a propriedade isNew é verdadeira, pois é através dela que estamos controlando se o usuário está tentando logar ou está criando um novo cadastro.

Na linha 78 usamos o método findOne repassando o filtro {login:req.body.login} para verificar se o login que o usuário preencher não existe. O segundo parâmetro deste método são os campos que deverão ser retornados, caso um usuário seja encontrado. O método .exec irá executar o findOne e o callback será chamado, onde podemos retornar um erro, já que não é possível cadastrar o mesmo login.

Se req.body.isNew for falso, o código na linha 99 é executado e fazemos a pesquisa ao banco pelo login e senha. Se houver um usuário com estas informações, usamos o método jwt.sign na linha 103 para criar o token de autenticação do usuário, e o retornarmos na linha 106. Se não houver um usuário no banco com o mesmo login e senha, retornarmos o erro na linha 108.

server.js

```
114 router.route('/posts/:post_id?')
115     .get(function (req, res) {
116         Post
117             .find()
118                 .sort([['date', 'descending']])
119                 .populate('user', 'name')
120                 .exec(function (err, posts) {
121                     if (err)
122                         res.send(err);
123                     res.json(posts);
124                 });
125     })
126     .post(auth, function (req, res) {
127         var post = new Post();
128         post.title = req.body.title;
129         post.text = req.body.text;
130         post.user = req.body.user._id;
131         if (post.title==null)
132             res.status(400).send('Título não pode ser nulo');
133         post.save(function (err) {
134             if (err)
135                 res.send(err);
136             res.json(post);
137         });
138     })
139     .delete(auth, function (req, res) {
140         Post.remove({
141             _id: req.params.post_id
142         }, function(err, post) {
143             if (err)
144                 res.send(err);
145             res.json({ message: 'Successfully deleted' });
146         });
147     });

```

Na linha 114 usamos /posts/:post_id? para determinar a url para obtenção de posts.

O uso do `:post_id` adiciona uma variável a url, por exemplo `/posts/5`. Já que usamos `/posts/:post_id?`, o uso do `?` torna a variável opcional.

Na linha 115 estamos tratando o método “GET /posts” que irá obter todos os posts do banco de dados. Na linha 118 usamos o método `sort` para ordenar os posts, e na linha 119 usamos o método `populate` para adicionar uma referência ao modelo `user`, que é o autor do Post. Esta referência é possível porque adicionamos na definição do *schema* de Post.

Na linha 126 criamos o método “POST /posts” que irá adicionar um Post. Criamos uma validação na linha 131 e usamos o método `Post.save()` para salvar o post no banco de dados.

Na linha 139 adicionamos o método “DELETE /posts/”, onde o post é apagado do banco de dados. Para apagá-lo, usamos o método `Post.remove` na linha 140, repassando o `id` (chave) do Post e usando o parâmetro `req.params.post_id`, que veio da url `/posts/:post_id?`.

server.js

```
148 //register router
149 app.use('/api', router);
150 //start server
151 var port = process.env.PORT || 8080;
152 app.listen(port);
153 console.log('Listen: ' + port);
```

Finalizando o script do servidor, apontamos a variável `router` para o endereço `/api`, na linha 149. Com isso, toda a api será exposta na url “/api”. Por exemplo, para obter todos os posts do banco de dados, deve-se fazer uma chamada GET ao endereço “/api/posts”. Nas linhas 151 e 152 definimos a porta em que o servidor express estará “escutando” e na linha 153 informamos via `console.log` qual a porta foi escolhida.

8.7 Testando o servidor

Para testar o servidor Web, podemos simplesmente executar o seguinte comando:

```
$ node server.js
```

Onde temos uma simples resposta: “Listen: 8080”. Se houver alguma alteração no arquivo `server.js`, esta alteração não será refletida na execução corrente do servidor, será necessário terminar a execução e reiniciar. Para evitar este retrabalho, vamos instalar a biblioteca `nodemon` que irá recarregar o servidor sempre que o arquivo `server.js` for editado.

```
$ npm install nodemon -g
```

Após a instalação, execute:

```
$ nodemon server.js
```

```
[nodemon] 1.8.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node server.js`
Listen: 8080
```

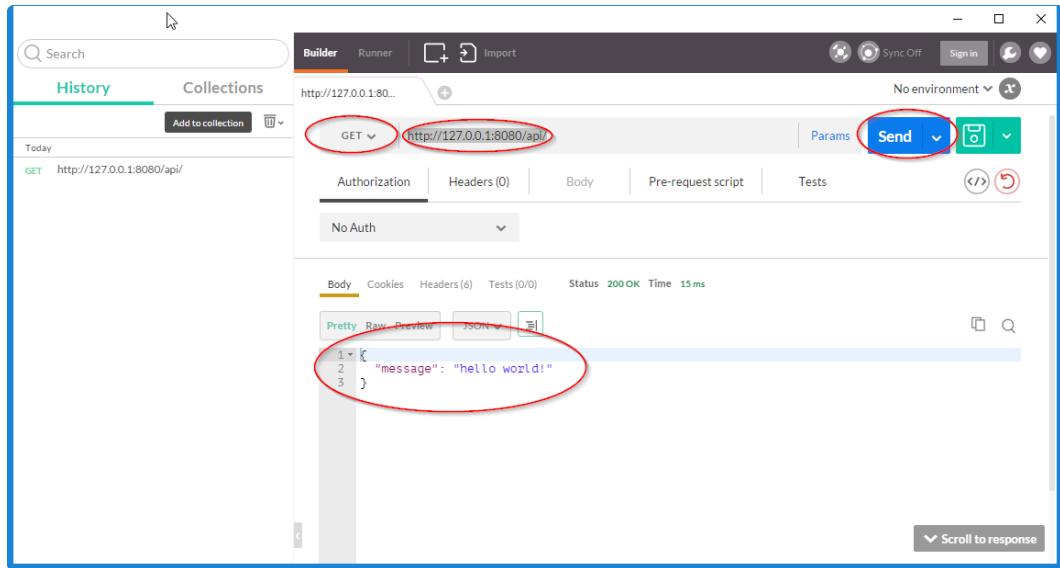
A resposta já indica que, sempre quando o arquivo `server.js` for atualizado, o comando `node server.js` também será.

8.8 Testando a api sem o Angular

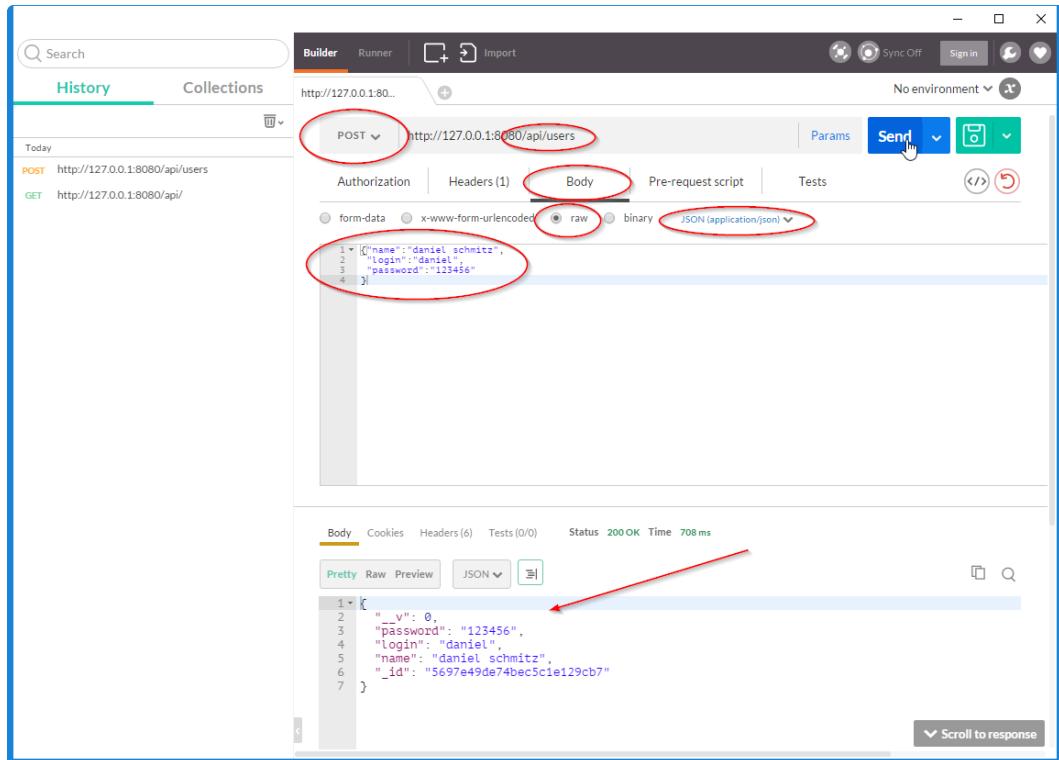
Pode-se testar a api que acabamos de criar, enviando e recebendo dados, através de um programa capaz de realizar chamadas Get/Post ao servidor. Um destes programas se chama **Postman**³, que pode ser instalado com um plugin para o Google Chrome.

Por exemplo, para testar o endereço `http://127.0.0.1:8080/api/`, configuramos o Postman da seguinte forma:

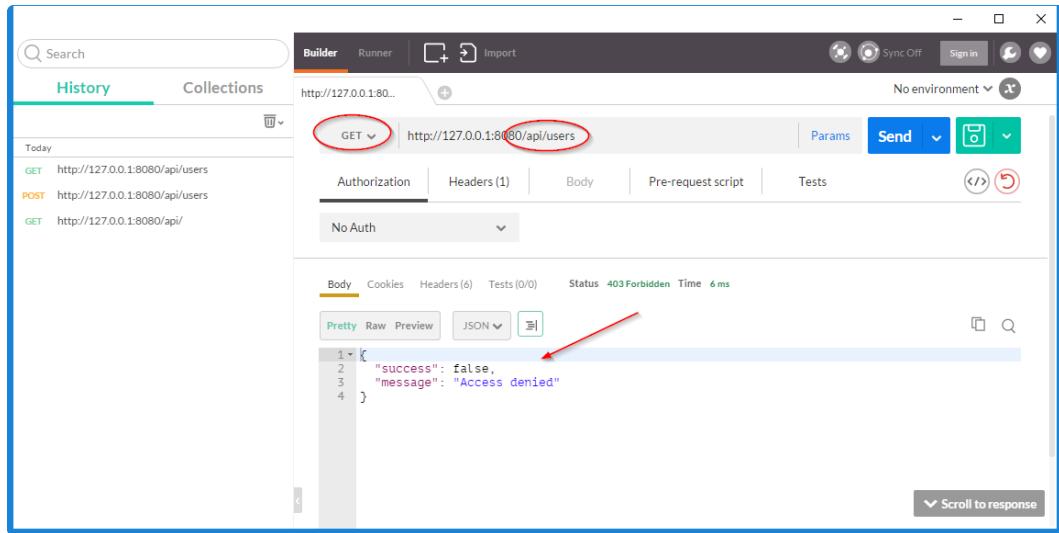
³<https://www.getpostman.com/>



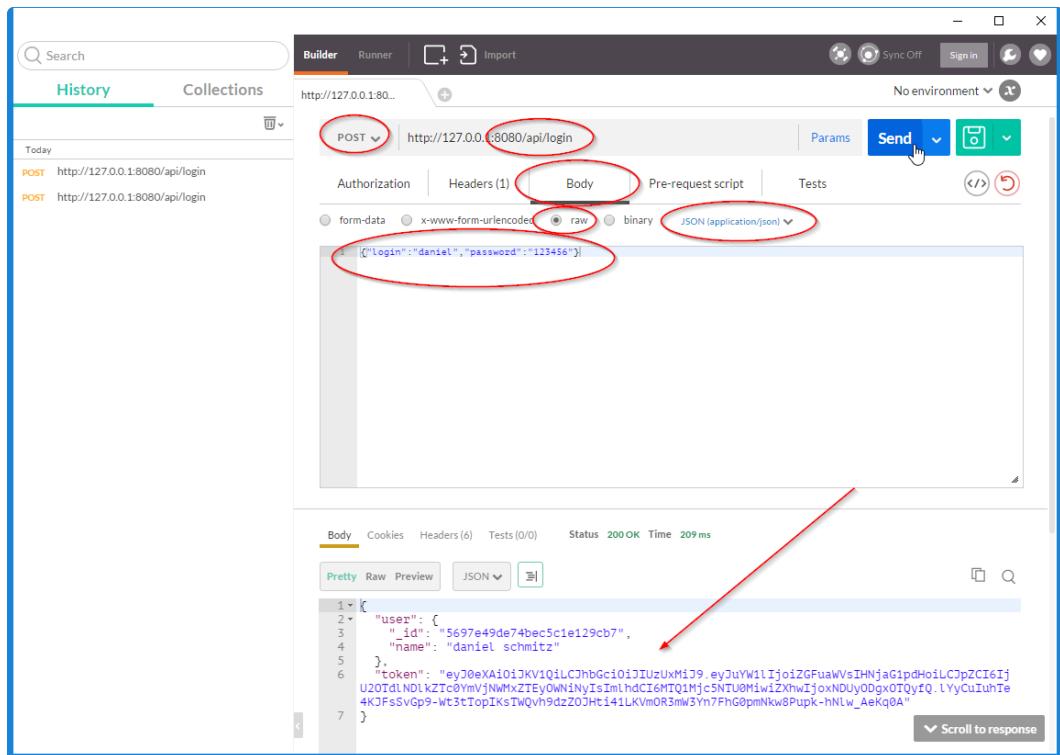
Perceba que obtemos a resposta “hello world”, conforme configurado no servidor. Para criar um usuário, podemos realizar um POST à url /users repassando os seguintes dados:



Para testar o login, vamos tentar acessar a url /api/users. Como configuramos que esta url deve passar pelo *middleware*, o token não será encontrado e um erro será gerado:

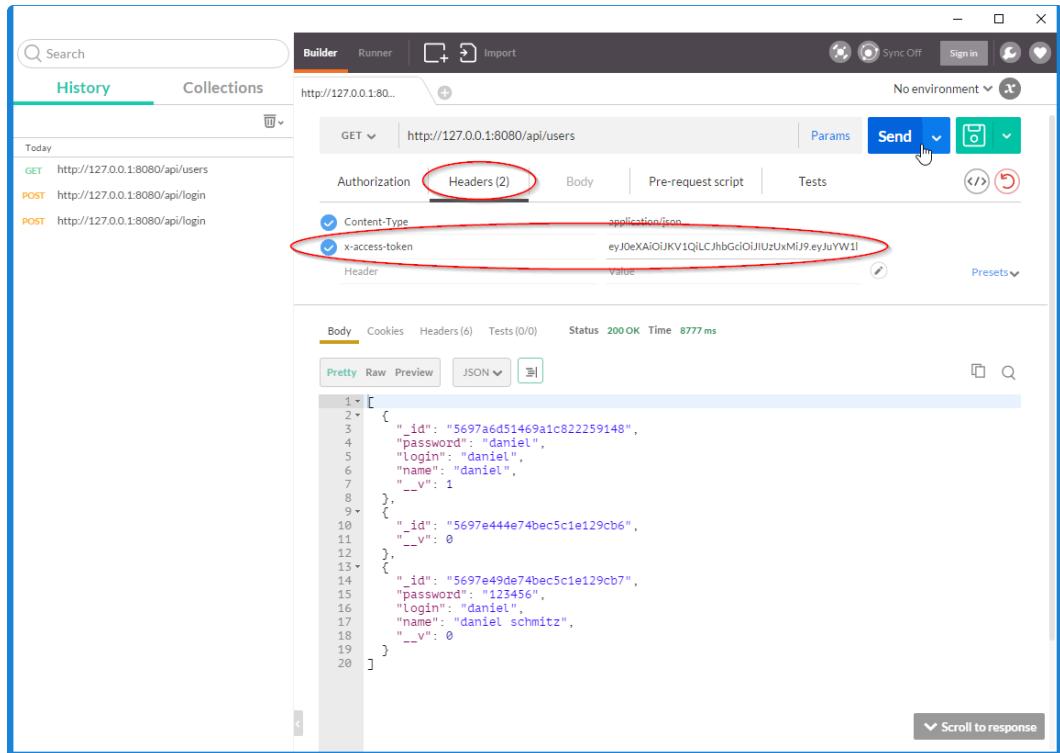


Para realizar o login, acessamos a URL /api/login, da seguinte forma:



Veja que ao repassarmos `login` e `password`, o token é gerado e retornado para o `postman`. Copie e cole este token para que possamos utilizá-lo nas próximas chamadas ao servidor. No Angular, iremos armazenar este token em uma variável.

Com o token, é possível retornar a chamada `GET /users` e repassá-lo no cabeçalho da requisição HTTP, conforme a imagem a seguir:



Veja que com o token os dados sobre os usuários são retornados. Experimente alterar algum caractere do token e refazer a chamada, para obter o erro “Failed to authenticate”.

9. Exemplo Final - Cliente

Com o servidor pronto, podemos dar início a criação da estrutura do Angular 2. Ao invés de copiar e colar o projeto “AngularBase”, vamos recriar cada passo da instalação do projeto para que possamos compreender melhor o seu funcionamento.

Voltando ao servidor express, configuramos os arquivos estáticos javascript da seguinte forma:

```
...
app.use('/', express.static(__dirname+'/public'));
app.use('/libs', express.static(__dirname+'/node_modules/bootstrap/dist'));
app.use('/libs', express.static(__dirname+'/node_modules/systemjs/dist'));
app.use('/libs', express.static(__dirname+'/node_modules/rxjs/bundles/'));
app.use('/libs', express.static(__dirname+'/node_modules/angular2/bundles'));
...
...
```

9.1 Arquivos iniciais

Crie o arquivo public/index.html, inicialmente com o seguinte código:

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Blog</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <base href="/"> <!-- <<< IMPORTANTE --->

  <!-- LIBS -->
  <script src="libs/angular2-polyfills.js"></script>
  <script src="libs/system.src.js"></script>
  <script src="libs/Rx.js"></script>
```

```
<script src="libs/angular2.dev.js"></script>
<script src="libs/http.dev.js"></script>
<script src="libs/router.dev.js"></script>
<link rel="stylesheet" href="libs/css/bootstrap.min.css">

<!-- BOOT -->
<script>
System.config({
  packages: {
    app: {
      format: 'register',
      defaultExtension: 'js'
    }
  }
});
System.import('app/boot')
  .then(null, console.error.bind(console));
</script>
</head>
<body>
  <my-app>Loading...</my-app>
</body>
</html>
```

O documento html que contém a configuração inicial da aplicação blog possui algumas diferenças em relação aos outros que criamos durante esta obra. Primeiro, estamos adicionando todas as bibliotecas utilizando o caminho /libs, que fisicamente não existe, mas foi criado virtualmente no express. Isso é importante para não expor toda a pasta node_modules ao acesso público.

Após adicionar as bibliotecas, iniciamos a configuração do `System.config`, que é idêntica aos exemplos anteriores, mas possui um detalhe **muito** importante. Como o arquivo `index.html` está na pasta `public`, ao executarmos `System.import('app/boot')` estamos na verdade executando o arquivo `public/app/boot.js`, que a princípio ainda não existe. Perceba que não podemos configurar o `import` com o caminho `../app/boot` para acessar o diretório `blog/app`, pois este não é visível para acesso. Isso significa que neste projeto teremos os arquivos TypeScript no diretório `/blog/app` e os arquivos `.js` e `.js.map` (são os arquivos compilados do TypeScript) no diretório `blog/public/app`. Para que esta configuração seja

possível, precisamos configurar o arquivo `tsconfig.json` da seguinte forma:

blog/tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "system",  
    "moduleResolution": "node",  
    "sourceMap": true,  
    "emitDecoratorMetadata": true,  
    "experimentalDecorators": true,  
    "removeComments": false,  
    "noImplicitAny": false,  
    "watch": true,  
    "outDir": "./public/app"  
  },  
  "exclude": [  
    "node_modules"  
  ]  
}
```

Veja que o arquivo `tsconfig.json` é muito semelhante aos anteriores, tendo como diferença o atributo `outDir`, onde indicamos um diretório em que os arquivos compilados serão criados. Neste contexto usamos o caminho `./public/app`.

Os arquivos `app/boot.ts` e `app/component.ts` podem ser criados, a princípio, com o mesmo código da aplicação `AngularBase`:

app/boot.ts

```
import {bootstrap} from 'angular2/platform/browser'  
import {AppComponent} from './app.component'  
import {Http, HTTP_PROVIDERS} from 'angular2/http'  
  
bootstrap(AppComponent, [HTTP_PROVIDERS]);
```

Veja que o `bootstrap` já carrega `HTTP_PROVIDERS`, pois iremos injetar `Http` no `services`.

A classe `AppComponent`, inicialmente, pode ser a seguinte:

```
import {Component} from 'angular2/core'

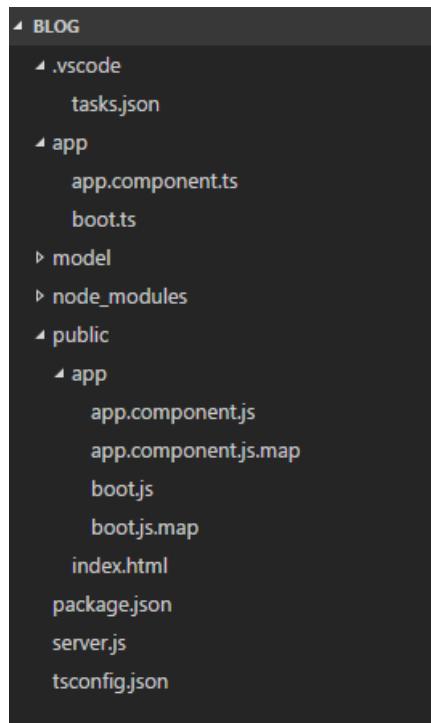
@Component({
  selector: 'my-app',
  template: '<h1>My Blog</h1>'
})
export class AppComponent { }
```

Antes de compilar a aplicação, se estiver utilizando o *Visual Studio Code*, configure o arquivo `.vscode/tasks.json` (ele será criado na primeira vez que pressionar `ctrl+shift+b`) com o seguinte texto:

`.vscode/tasks.json`

```
{
  "version": "0.1.0",
  "command": "tsc",
  "isShellCommand": true,
  "showOutput": "silent",
  "problemMatcher": "$tsc"
}
```

Ao compilarmos a aplicação, pressionando `ctrl+shift+b` no *Visual Studio Code*, ou digitando `tsc` no terminal (no diretório `blog`), a compilação é feita e os arquivos `js` e `map` são gerados em `public/app`. A estrutura de diretórios é igual a exibida a seguir:

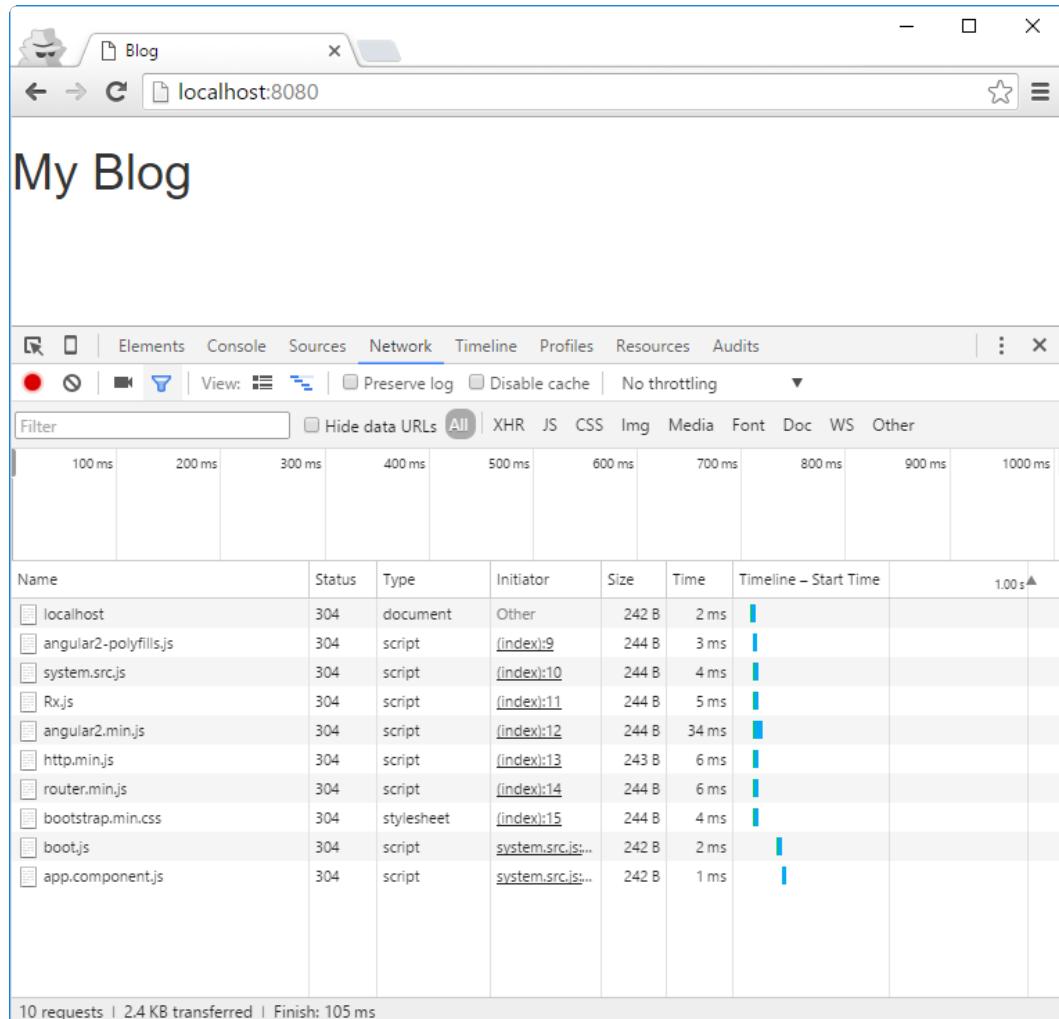


Para testar no navegador, use o terminal e no diretório blog digite:

```
$ nodemon server.js

[nodemon] 1.8.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node server.js`
Listen: 8080
```

Acesse <http://localhost:8080/> e obtenha o resultado semelhante a figura a seguir:



9.2 Preparando o Template base da aplicação

O arquivo `index.html` possui a chamada ao elemento `<my-app>` e é nele que configuramos a base da aplicação, que está instanciada na classe `AppComponent`. Geralmente todo o HTML da aplicação (incluindo menu e área de conteúdo) é adicionada ao `AppComponent`, e para ele criaremos um arquivo de template chamado `appComponent.html` que ficará localizado no diretório `public`. Isso é necessário porque somente o diretório `public` é visível a aplicação. Crie o arquivo `public/appComponent.html`, com o seguinte texto:

```
public/appComponent.html
```

```
<h1>My Blog</h1>
```

E alterarmos o AppComponent para:

```
import {Component} from 'angular2/core'

@Component({
  selector: 'my-app',
  templateUrl: 'appComponent.html'
})
export class AppComponent { }
```

Teremos o mesmo resultado, mas com o template separado do componente. Neste template, podemos usar algumas classes css do *bootstrap* para estilizar a aplicação. A princípio, podemos adicionar o seguinte html:

```
public/appComponent.html
```

```
<nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button"
        class="navbar-toggle collapsed"
        data-toggle="collapse"
        data-target="#navbar"
        aria-expanded="false"
        aria-controls="navbar">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#"> BLOG </a>
    </div>
    <div id="navbar" class="collapse navbar-collapse">
```

```
<ul class="nav navbar-nav">
  <!-- -->
  <!-- MENU -->
  <!-- -->
</ul>
</div>
</div>
</nav>
<div style="padding-top:50px">
  CONTEUDO
</div>
```

Veja que separamos a aplicação em um menu e o seu conteúdo. Iremos a partir deste ponto implementar o *router*.

9.3 Implementando o roteamento (Router)

Para implementar o router na aplicação, precisamos realizar as seguintes configurações:

- Dividir a aplicação em componentes
- Configurar o AppComponent
- Incluir o menu
- Configurar o html incluindo `<router-outlet>`

9.3.1 Criando componentes

Cada tela de uma aplicação deve ser um componente. Neste pequeno blog, teremos três telas:

1. HomeComponent: Exibe todos os posts
2. LoginComponent: Formulário de login
3. AddPostComponent: Adiciona um post

Os componentes podem ser criados no diretório `app/components`, a princípio sem nenhuma informação:

app/component/home.ts

```
import {Component} from 'angular2/core'

@Component({
  template: `HomeComponent`
})
export class HomeComponent {  
}  
}
```

app/component/login.ts

```
import {Component} from 'angular2/core'

@Component({
  template: `LoginComponent`
})
export class LoginComponent {  
}  
}
```

app/component/addpost.ts

```
import {Component} from 'angular2/core'

@Component({
  template: `AddPostComponent`
})
export class AddPostComponent {  
}  
}
```

Para definir uma “library” destes componentes, crie o arquivo app/component.ts com o seguinte código:

```
export * from './component/home'  
export * from './component/login'  
export * from './component/addpost'
```

Através desta library, podemos referenciar as classes de componente da seguinte forma:

```
import { HomeComponent, LoginComponent, AddPostComponent } from './component'
```

O que é melhor do que importar desta forma:

```
import { HomeComponent } from './component/home'  
import { HomeComponent } from './component/home'  
import { AddPostComponent } from './component/addpost'
```

9.3.2 Configurando o @RouteConfig

Com os componentes prontos, podemos voltar ao AppComponent e configurar o router através da diretiva RouterConfig:

```
import { Component } from 'angular2/core'  
import { ROUTER_DIRECTIVES, ROUTER_PROVIDERS, RouteConfig } from 'angular2/router'  
  
import { HomeComponent, LoginComponent, AddPostComponent } from './component'  
  
@Component({  
    selector: 'my-app',  
    providers: [ROUTER_PROVIDERS],  
    directives: [ROUTER_DIRECTIVES],  
    templateUrl:'appComponent.html',  
  
})  
@RouteConfig([  
    { path: '/', name: 'Home', component: HomeComponent,  
        useAsDefault: true },  
    { path: '/Login', name: 'Login', component: LoginComponent },  
    { path: '/Addpost', name: 'AddPost', component: AddPostComponent }
```

```
])  
export class AppComponent {  
  
}
```

Veja que importamos ROUTER_DIRECTIVES e ROUTER_PROVIDERS que devem ser adicionados as propriedades directives e providers do @Component. Depois usamos o @RouteConfig para criar três rotas, ligando cada uma delas aos componentes criados.

9.3.3 Configurando o menu

O menu da aplicação deve estar integrado as rotas. Para isso, edite o arquivo public\appComponent.html incluindo o seguinte menu:

public\appComponent.html

```
<nav class="navbar navbar-inverse navbar-fixed-top">  
  <div class="container">  
    <div class="navbar-header">  
      <button type="button"  
        class="navbar-toggle collapsed"  
        data-toggle="collapse"  
        data-target="#navbar"  
        aria-expanded="false"  
        aria-controls="navbar">  
        <span class="sr-only">Toggle navigation</span>  
        <span class="icon-bar"></span>  
        <span class="icon-bar"></span>  
        <span class="icon-bar"></span>  
      </button>  
      <a class="navbar-brand" href="#">{{title}}</a>  
    </div>  
    <div id="navbar" class="collapse navbar-collapse">  
      <ul class="nav navbar-nav">  
  
        <li><a [routerLink]="/Home">Home</a></li>  
        <li><a [routerLink]="/Login">Login</a></li>  
        <li><a [routerLink]="/AddPost">Add Post</a></li>
```

```
</ul>
</div><!-- .nav-collapse -->
</div>
</nav>
<div style="padding-top:50px">
    CONTEUDO
</div>
```

9.3.4 Configurando o router-outlet

Para finalizar a criação do Router, é preciso informar onde os componentes serão carregados. Isso é realizado através da tag `<router-outlet>` que deve ser inserida no template que contém a configuração do router:

public\appComponent.html

```
<nav class="navbar navbar-inverse navbar-fixed-top">
    <!-- ... MENU ... -->
</nav>
<div style="padding-top:50px">
    <router-outlet></router-outlet>
</div>
```

Para testar a aplicação, recarregue a página e clique nos menus para verificar o conteúdo sendo carregado.

9.4 Exibindo Posts

No componente HomeComponent, vamos exibir os posts que estão cadastrados no banco. Para isso, faremos uma chamada GET ao endereço `/api/posts`, que não necessita de autenticação. Ao invés de programar esta chamada no componente, vamos criar um service, que será chamado de PostService e o model Post que contém a definição de um post, veja:

```
import {User} from './model'
export class Post{
    public _id:string;
    public title: string;
    public user: User;
    public body: string;
    public date: Date
    constructor(){}
}
```

E a classe user.ts:

```
export class User{
    public _id:string;
    public name: string;
    public email: string;
    public password: string;
    public isNew: Boolean;
    constructor(){}
}
```

E vamos criar a library também:

app/model.ts

```
export * from './model/post'
export * from './model/user'
```

O service usa a classe Http para fazer a chamada ao servidor:

app/service/post.ts

```
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map' // <<<<<
```

```
@Injectable()
export class PostService {
    constructor(private http: Http) { }
    public getPosts() {
        return this.http
            .get('./api/posts')
            .map(res => res.json());
    }
}
```

Usamos o recurso de injeção de dependência da classe `Http`, que foi configurado no bootstrap da aplicação (`app/boot.ts`). O método `getPosts` irá fazer uma requisição ajax ao endereço `./api/posts` e retornar uma instância da classe `Observable` do Angular 2, que será tratada na classe de quem realizou a chamada ao método `getPosts()`.

Como queremos exibir os *posts* na página principal, devemos configurar o `HomeController` para acessar o este service. Isso é feito da seguinte forma:

app/component/home.ts

```
import {Component} from 'angular2/core'
import {PostService} from '../service/post'
import {Post} from '../model'

@Component({
    providers: [PostService],
    template: "HomeComponent"
})
export class HomeComponent {
    public posts: Array<Post>;
    constructor(private postService: PostService) {
        postService.getPosts().subscribe(
            p => this.posts = p,

```

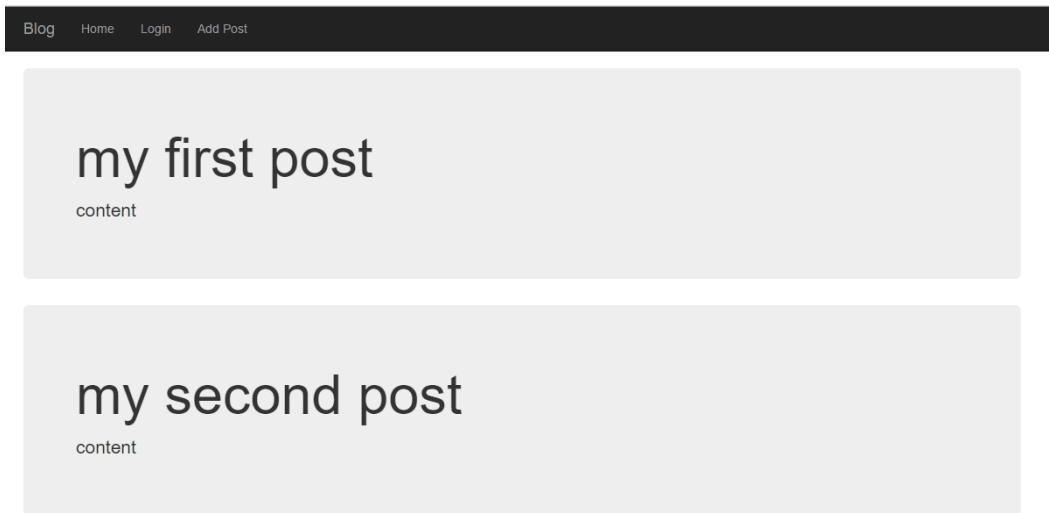
```
        err => console.log(err)
    );
}
}
```

Importamos PostService e o adicionamos aos providers do componente. Com isso podemos injetá-lo no método construtor. Após injetá-lo, chamamos postService.getPosts() que, quando responder com os posts, executa o método subscribe onde é possível obter os posts ou tratar algum eventual erro. Repare que os posts serão armazenados na variável this.posts, na qual podemos usá-la para exibir os posts na página.

Para finalizar, alteramos o template do componente para:

```
// imports...
@Component({
    providers: [PostService],
    template: `
        <div class="jumbotron" *ngFor="#p of posts">
            <h1>{{p.title}}</h1>
            <p>content</p>
        </div>
    `
})
export class HomeComponent { .... }
```

Usamos *ngFor para realizar um loop entre os itens de posts e, por enquanto, exibimos o título do post. O resultado é semelhante a imagem a seguir:



9.5 Login

A tela de login é definida pelo LoginComponent, que possui a seguinte estrutura:

app/component/login.ts

```
import {Component} from 'angular2/core'  
import {User} from '../model'  
import {UserService} from '../service/user'  
import {LoginService} from '../service/login'  
import {Router} from 'angular2/router'  
  
@Component({  
  providers: [UserService],  
  template: `<  
    >>TEMPLATE<<  
  `,  
})  
export class LoginComponent {  
  private user:User=new User();  
  private showLoading:boolean = false;  
  private errorMessage:string = null;
```

```
constructor(private userService:UserService,
            private loginService:LoginService,
            private router:Router){

}

onClick(event){
    event.preventDefault();
    this.showLoading = true;
    this.errorMessage = null;
    this.userService.insert(this.user).subscribe(
        result => this.onLoginResult(result),
        error => this.onLoginError(error)
    );
}

onLoginResult(result){
    console.log(result);
    this.loginService.setLogin(result.user,result.token);
    this.router.navigate( ['Home'] );
}

onLoginError(error){
    this.showLoading = false;
    this.errorMessage = error._body;
}

}
```

Antes de exibir o template (que é um pouco mais complexo que o template do HomeController), vamos analisar como o LoginComponent funciona. Nos imports, percebe-se que estamos utilizando UserService e LoginService, classes que tem como objetivo prover dados a classe do componente. Veremos estas classes em breve!

No LoginComponent, temos duas variáveis que controlam o template da aplicação. Por exemplo, showLoading irá controlar se uma mensagem “Carregando” será exibida, como no exemplo a seguir:

```
<div class="col-md-4 col-md-offset-4" *ngIf="showLoading">
    Aguarde...
</div>
```

Com o template acima, é possível configurar a sua visualização apenas definindo o valor da variável `showLoading` para `true` ou `false`.

O template completo do `LoginComponent` é exibido a seguir:

```
<div class="col-md-4 col-md-offset-4" *ngIf="!showLoading">

    <div *ngIf="errorMessage" class="alert alert-danger" role="alert">
        {{errorMessage}}
    </div>

    <form ngForm>
        <div class="form-group">
            <label for="login">Login</label>
            <input type="text" class="form-control"
                id="login"
                required
                placeholder="Login"
                [(ngModel)]="user.login">
        </div>
        <div class="form-group">
            <label for="password">Password</label>
            <input type="password" class="form-control"
                id="password"
                required
                placeholder="Password"
                [(ngModel)]="user.password">
        </div>
        <div class="checkbox">
            <label>
                <input id="createAccount" type="checkbox"
                    [(ngModel)]="user.isNew"> Create Account?
            </label>
        </div>
        <div class="form-group" *ngIf="user.isNew">
            <label for="login">Your Name</label>
            <input type="text" class="form-control"
                id="name">
        </div>
    </form>
</div>
```

```
placeholder="Your Name"
[(ngModel)]="user.name">
</div>

<button type="submit" class="btn btn-default pull-right"
(click)="onClick($event)">Login</button>

</form>
</div>

<div class="col-md-4 col-md-offset-4"
*ngIf="showLoading">
Aguarde...
</div>
```

9.6 Services

Algumas classes como `LoginService` e `HeadersService` podem ser usadas em vários componentes, então eles são injetados no arquivo `boot.ts`, veja:

`app/boot.ts`

```
import {bootstrap} from 'angular2/platform/browser'
import {AppComponent} from './app.component'
import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {LoginService} from './service/login'
import {HeadersService} from './service/headers'

bootstrap(AppComponent, [HeadersService, HTTP_PROVIDERS, LoginService]);
```

Existem alguns services que são usados em poucas classes, como no exemplo do `UserService`, então ele é injetado na classe através do `providers`, como foi visto no `LoginComponent`.

9.6.1 LoginService

A classe `LoginService` provê algumas informações sobre o login do usuário:

app/service/login.ts

```
import {Injectable} from 'angular2/core'
import {User} from '../model'

@Injectable()
export class LoginService {
    private user:User=null;
    private token:string=null;
    constructor() { }
    setLogin(u:User,t:string){
        this.user = u;
        this.token = t;
    }
    getToken():string{
        return this.token;
    }
    getUser(){
        return this.user;
    }
    isLoggedIn(){
        return this.user!=null && this.token!=null;
    }
    logout(){
        this.user = null;
        this.token = null;
    }
}
```

9.6.2 UserService

A classe UserService tem a responsabilidade de inserir um usuário.

```
import {Http, HTTP_PROVIDERS, Headers} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map'
import {User} from '../model'
import {HeadersService} from './headers'

@Injectable()
export class UserService {
  constructor(private _http: Http, private _header:HeadersService) {

  }
  public insert(u:User) {
    return this._http
      .post('./api/login',
        JSON.stringify(u),
        this._header.getJsonHeaders())
      .map(res => res.json());
  }
}
```

9.6.3 HeadersService

Esta classe é usada para fornecer informações de cabeçalho *http* às requisições que o cliente faz ao servidor:

```
import {Injectable} from 'angular2/core'
import {Headers} from 'angular2/http'

@Injectable()
export class HeadersService {
  constructor(){}
  getJsonHeaders(token?:string){
    var headers = new Headers();
    headers.append('Content-Type', 'application/json');
    if (token)
      headers.append('x-access-token',token)
  }
}
```

```
    return {headers: headers};  
}
```

O método `getJsonHeaders` retorna um cabeçalho configurado para o formato JSON, e caso a propriedade `token` seja repassada, ela é adicionada ao cabeçalho. Toda requisição *Http* realizada do cliente ao servidor possui um cabeçalho no qual podemos repassar algumas informações. Neste caso, estamos repassando o `Content-Type` e o `x-access-token`.

9.7 Conectando no servidor

Vamos analisar como o Angular conecta no servidor e obtém informações. Na `LoginComponent`, temos o botão de Login com a seguinte ação: `(click)="onClick($event)"`. Quando o usuário clica no botão, o método `onClick` da classe `LoginComponent` é disparado, que possui o seguinte código:

app/component/login.ts

```
import {Component} from 'angular2/core'  
import {User} from '../model'  
import {UserService} from '../service/user'  
import {LoginService} from '../service/login'  
import {Router} from 'angular2/router';  
  
@Component({  
  providers: [UserService],  
  template: `  
  
    <div class="col-md-4 col-md-offset-4" *ngIf="!showLoading">  
  
      <div *ngIf="errorMessage" class="alert alert-danger" role="alert">  
        {{errorMessage}}  
      </div>  
  
      <form ngForm>  
        <div class="form-group">  
          <label for="login">Login</label>  
          <input type="text" class="form-control" id="login" required plac\
```

```
eholder="Login" [(ngModel)]="user.login">
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input type="password" class="form-control" id="password" required placeholder="Password" [(ngModel)]="user.password">
    </div>
    <div class="checkbox">
        <label>
            <input id="createAccount" type="checkbox" [(ngModel)]="user.isNew" value="true" checked="checked" /> Create Account?
        </label>
    </div>
    <div class="form-group" *ngIf="user.isNew">
        <label for="login">Your Name</label>
        <input type="text" class="form-control" id="name" placeholder="Your Name" [(ngModel)]="user.name">
    </div>

        <button type="submit" class="btn btn-default pull-right" (click)="onClick($event)">Login</button>

    </form>
</div>

<div class="col-md-4 col-md-offset-4" *ngIf="showLoading">
    Aguarde...
</div>
<br/>
})
```

```
export class LoginComponent {
    private user:User=new User();
    private showLoading:boolean = false;
    private errorMessage:string = null;
    constructor(private userService:UserService,
                private loginService>LoginService,
                private router:Router){
```

```
}

onClick(event){
    event.preventDefault();
    this.showLoading = true;
    this.errorMessage = null;
    this.userService.insert(this.user).subscribe(
        result => this.onLoginResult(result),
        error => this.onLoginError(error)
    );
}

onLoginResult(result){
    console.log(result);
    this.loginService.setLogin(result.user,result.token);
    this.router.navigate( ['Home' ] );
}

onLoginError(error){
    this.showLoading = false;
    this.errorMessage = error._body;
}

}
```

Inicialmente o método `onClick` chama `event.preventDefault()`, fazendo com que o envio do formulário seja cancelado (já que vamos usar ajax). Então exibimos uma mensagem de “Carregando” para mostrar ao usuário que algo está sendo processado. É importante também esconder o formulário, para que o usuário não clique duas vezes no botão. O método `insert` da classe `userService` é chamado, método este que vai conectar no servidor enviando `this.user` no formato json. Quando o service retornar com os dados do servidor, o método `subscribe()` é executado e ele possui dois callbacks: `result` e `error`. Para cada um deles criamos um método no próprio componente para que possamos tratá-lo.

O método `onLoginResult` é executado quando o usuário está devidamente autenticado. Usamos a classe `LoginService` para informar este login, e ele poderá ser usado em qualquer lugar da aplicação (essa é uma das vantagens da injeção de dependência). Depois usamos o `router` para navegar de volta ao Home.

Caso ocorra algum erro o método `onLoginError` será disparado. Então escondemos a mensagem de “Carregando...” e exibimos a mensagem de erro. Veja que, apenas por preencher a variável `this.errorMessage`, o `div` será visível na tela.

9.8 Posts

Após o usuário logar, ele poderá clicar no link “AddPost”, que será redirecionado para o component AddPostComponent.

app/component/addPost.ts

```
import {Component} from 'angular2/core'
import {Post} from '../model'
import {LoginService} from '../service/login'
import {Router} from 'angular2/router'
import {PostService} from '../service/post'

@Component({
  providers: [PostService],
  template: `<div>
    <h2>Add Post</h2>
    <form>
      <input type="text" placeholder="Title" [(ngModel)]="post.title" />
      <input type="text" placeholder="Content" [(ngModel)]="post.content" />
      <button (click)="onClick($event)">Add</button>
    </form>
  </div>`})
export class AddPostComponent {
  private post:Post = new Post();
  private errorMessage:string = null;
  private showLoading:boolean = false;

  constructor(private _loginService:LoginService,
              private _router:Router,
              private _postService:PostService) {
    if (!_loginService.isLoggedIn())
      this._router.navigate( ['Login'] );
    this.post.user = this._loginService.getUser();
  }

  onClick(event){
    event.preventDefault();
    this.showLoading = true;
    this.errorMessage = null;
    this._postService.insert(this.post).subscribe(
      result => this.onInsertPostResult(result),
      error => this.onError(error)
    );
  }

  onInsertPostResult(result) {
    this._router.navigate(['Post', result.id]);
  }

  onError(error) {
    this.errorMessage = error.message;
  }
}
```

```
        error => this.onInsertPostError(error)
    );
}

onInsertPostResult(result){
    this._router.navigate( [ 'Home' ] );
}

onInsertPostError(error){
    this.showLoading = false;
    this.errorMessage = error._body;

}

}
```

O componente AddPostComponent tem várias semelhanças com o formulário de Login. A forma como exibimos erros e a mensagem e “Carregando...” é a mesma. Uma novidade é que usamos o service LoginService para verificar se o usuário está logado, e e caso negativo redirecionamos para o formulário de login.

Uma diferença pequena em relação ao componente LoginComponent é que usamos o nome das variáveis injetadas (LoginService, PostService) com o prefixo `_`, ou seja, usamos `_postService` e `_loginService`. Você pode escolher o que mais combina com o seu projeto.

O template desta tela é apresentado a seguir:

```
<div class="col-md-4 col-md-offset-4" *ngIf="showLoading">
    Aguarde...
</div>
<div class="col-md-8 col-md-offset-2" *ngIf="!showLoading">
<div *ngIf="errorMessage" class="alert alert-danger" role="alert">
    {{errorMessage}}
</div>
<div class="panel panel-default">
    <div class="panel-heading">Add Post</div>
    <div class="panel-body">
        <form ngForm>
            <div class="form-group">
                <label for="title">Title</label>
                <input type="text" class="form-control"
```

```

        id="title" required placeholder="Title"
        [(ngModel)]="post.title">
</div>
<div class="form-group">
    <label for="text">Text</label>
    <textarea rows=10 cols=100 class="form-control"
        id="text" [(ngModel)]="post.text"></textarea>
</div>
<button type="submit" class="btn btn-default pull-right"
    (click)="onClick($event)">Create</button>
</form>
</div>
</div>
</div>

```

9.8.1 PostService

A classe PostService é responsável em conectar no servidor e realizar três operações básicas: obter posts, incluir posts e excluir posts:

app/service/post.ts

```

import {Http, HTTP_PROVIDERS} from 'angular2/http'
import {Injectable} from 'angular2/core'
import 'rxjs/add/operator/map'
import {Post} from '../model'
import {HeadersService} from '../service/headers'
import {LoginService} from '../service/login'

@Injectable()
export class PostService {
    constructor(private _http: Http,
                private _headerService:HeadersService,
                private _loginService:LoginService) { }

    public getPosts() {
        return this._http
            .get('../api/posts')
            .map(res => res.json());
    }
}

```

```
public insert(p:Post){  
    return this._http  
        .post('./api/posts', JSON.stringify(p),  
            this._headerService.  
                getJsonHeaders(this._loginService.getToken()))  
        .map(res => res.json());  
}  
public delete(p:Post){  
    return this._http  
        .delete('./api/posts/' + p._id ,  
            this._headerService.  
                getJsonHeaders(this._loginService.getToken()))  
        .map(res => res.json());  
}  
}
```

O método `getPosts` realiza uma chamada “GET /api/posts” ao servidor. O método `insert` irá realizar uma chamada “POST /api/posts”, mas aqui temos um detalhe importante:

```
.post('./api/posts',  
      JSON.stringify(p),  
      this._headerService.  
          getJsonHeaders(this._loginService.getToken())  
    )
```

Perceba que usamos a classe `HeaderService` para chamar o método `getJsonHeaders`, que irá formatar o cabeçalho da requisição (Requisição HTTP) como JSON (*application/json*). Pode-se conferir este comportamento analisando a requisição na aba networking do *Google Chrome Developer Tools*. Além desta formatação, repassamos como parâmetro o `Token`, pois neste estágio o usuário tem que estar logado.

O método `delete` adiciona na url o `id` do post, juntamente com o `Token` através do `HeaderService`.

9.9 Refatorando a tela inicial

Para finalizar este pequeno sistema, vamos retornar ao template do `HomeComponent`, que antes exibia o título dos posts e adicionar mais algumas funcionalidades, veja:

app/component/home.ts

```
import {Component} from 'angular2/core'
import {PostService} from '../service/post'
import {Post} from '../model'
import {LoginService} from '../service/login'
import {User} from '../model';

@Component({
  providers: [PostService], // OBS: LoginService at boot.ts
  template: `
    <div class="alert alert-info" *ngIf="showLoading">
      Aguarde...
    </div>
    <div *ngIf="!showLoading">
      <div *ngIf="_loginService.isLoggedIn()" class="alert alert-success">
        Olá {{_loginService.getUser().name}}
        <a href="#" (click)="logout($event)" class="pull-right">
          Sair</a>
      </div>
      <div class="jumbotron" *ngFor="#p of posts">
        <h1>{{p.title}}</h1>
        <p>{{p.text}}</p>
        <p>Por: {{p.user?.name}}</p>
        <a href="#" (click)="deletePost(p)" *ngIf="checkPost(p)">Apagar</a>
      </div>
    </div>
  `
})
export class HomeComponent {

  private posts: Array<Post>;
  private showLoading:boolean=false;
```

```
constructor(private _postService: PostService,
            private _loginService:LoginService) {
    this.loadAllPosts();
}

loadAllPosts(){
    this.showLoading = true;
    this._postService.getPosts().subscribe(
        p => this.onLoadAllPostsResult(p),
        err => console.log(err)
    );
}

onLoadAllPostsResult(p){
    this.posts = p;
    this.showLoading = false;
}

logout(event){
    this._loginService.logout();
}

checkPost(p:Post):boolean{
    try {
        if (p.user == null) return false;
        if (!this._loginService.isLoggedIn()) return false;
        return p.user._id==this._loginService.getUser()._id;
    } catch (error) {
        return false;
    }
    return false;
}

deletePost(p){
    this._postService.delete(p).subscribe(
        result => this.onDeletePostResult(result),
        error => this.onDeletePostError(error)
    )
}

onDeletePostResult(result){
    this.loadAllPosts();
}

onDeletePostError(error){
```

```
        console.log(error);
    }
}
```

Existem duas diferenças importantes no componente. A primeira é que incluímos uma barra exibido se o usuário está logado:

```
<div *ngIf="_loginService.isLoggedIn()"
      class="alert alert-success">
  Olá {{_loginService.getUser().name}}
  <a href="#" (click)="logout($event)"
      class="pull-right" >
    Sair
  </a>
</div>
```

Veja que usamos `LoginService` para controlar tanto a visibilidade quanto as informações do login. Também criamos um método `logout` que irá chamar o método `this._loginService.logout()`, atualizando toda a tela novamente.

Outro detalhe é que incluímos o botão “Excluir” em cada Post:

```
<a href="#" (click)="deletePost(p)"
    *ngIf="checkPost(p)">Apagar</a>
```

A visibilidade do link é controlada pelo método `checkPost` que retorna verdadeiro se, e somente se, o usuário logado é o dono do Post.

Após estas modificações, a tela principal do sistema fica semelhante à figura a seguir:

The screenshot shows a web-based blog application interface. At the top, there is a dark header bar with three items: "Blog", "Home", and "Add Post". Below this is a light green navigation bar containing the text "Olá Mike" and a small user icon, along with a "Sair" (Logout) link. The main content area displays two blog posts. The first post, titled "Wow, hello World", contains the text "Hi, i am mike :)" and "Por: Mike". It also has a blue "Apagar" (Delete) link. The second post, titled "My first post", contains the text "this is my first post as "Daniel" user" and "Por: Daniel Schmitz". Both posts are displayed in a light gray box.

9.10 Conclusão

Ainda existem alguns detalhes a serem realizados no projeto blog, no qual deixaremos como exercício para o leitor. Por exemplo, a edição de Posts, a troca de senha, envio de senha por email, inclusão de comentários etc. Todos estes processos usam os mesmos conceitos que aprendermos no decorrer da obra.

10. Utilizando Sublime Text

Em toda nossa obra usamos o Visual Studio Code como IDE para a criação de nossos projetos. Durante o lançamento do livro, alguns leitores nos pediram para comentar um pouco também sobre a integração do Angular 2 com o Sublime Text, pois esse era o editor de código que mais usam.

10.1 Instalação

Se deseja instalar o Sublime Text, recomendamos a versão 3, que pode ser encontrada em (neste link)[<http://www.sublimetext.com/3>]. Faça o download de acordo com a versão do seu sistema operacional, e após instalar o editor, você deverá instalar um plugin chamado “Package Control”. Este plugin é encontrado (neste link)[<https://packagecontrol.io/installation#st3>]. O processo de instalação dele é realizado da seguinte forma:

- Copie o texto de instalação, certifique-se que a aba “Sublime Text 3” está ativa.
- Com o editor aberto, vá até `View > Show Console`
- Com o console aberto, cole o texto que copiou no site e pressione enter

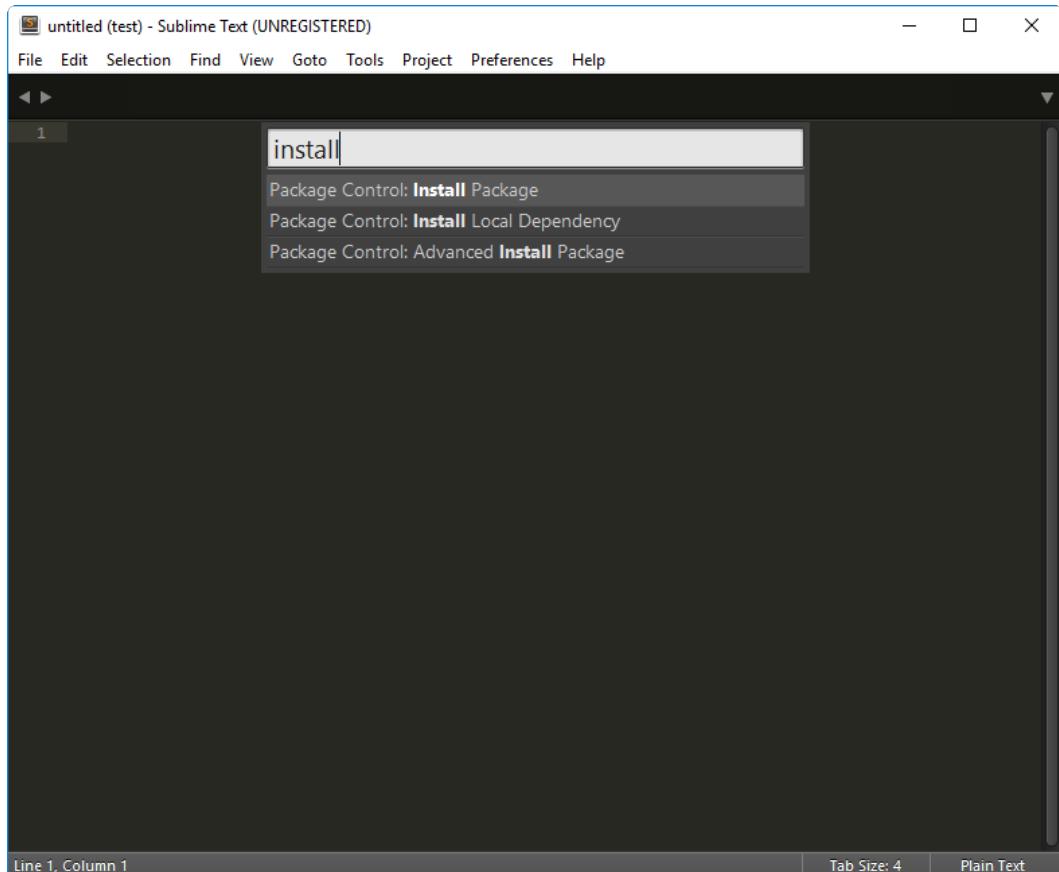
Aguarde alguns até a instalação terminar. Se surgir alguma mensagem de erro, reinicie o editor para que as bibliotecas sejam recarregadas.



Lembre-se que precisamos ter node e typescript instalados, conforme visto nos capítulos anteriores.

10.2 Adicionando suporte a linguagem TypeScript

Com o Package Control instalado, vamos instalar outro plugin que dará suporte a linguagem TypeScript . Pressione `Ctrl+Shift+p` e digite “install” até surgir a ação “Package Control: Install Package”, semelhante a figura a seguir:



Surge a lista de plugins que podem ser instalados no Sublime Text. Escreva Typescript e instale o plugin. Após a instalação, já podemos testar o TypeScript. Acesse File>open Folder e abra um diretório vazio. Adicione o arquivo (File > New File) index.ts e inclua algum código TypeScript, como no exemplo a seguir:

```
class HelloWorld{
    constructor(){

    }
    static say(){
        return "Hello World";
    }
}
```

Após a criação da classe, pressione **ctrl+b** para iniciar o processo de compilação. Na caixa de texto **Build Parameters** que abre, deixe em branco e pressione enter. O Plugin irá executar o comando **tsc** internamente, assim como é feito no **Visual Studio Code** e o arquivo com a extensão **.js** será criado.

10.3 Automatizando a build TypeScript

Para que o processo de build seja automatizado, podemos criar o arquivo **tsconfig.json** com o seguinte código:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false,
    "watch": true,
    "outDir": "./public/app"
  },
  "exclude": [
    "node_modules"
  ]
}
```

Após criar o arquivo, compile a aplicação com `ctrl+b` e veja que o diretório `public` foi criado, conforme a propriedade `outDir` da configuração.

11. Publicando a aplicação em um servidor cloud

Como etapa final desta obra, precisamos viabilizar a publicação da aplicação em um servidor real, no qual poderemos disponibilizar a aplicação para todos. Até esse momento, criamos a aplicação localmente e usamos o comando `node server.js` para iniciar o servidor express. Usamos Node e Mongo para servir a aplicação em Angular 2, e iremos partir deste mesmo princípio para publicá-la.



Este capítulo requer um pouco de conhecimento em ambientes Linux. Expressões como root, sudo, ssh, nano, devem ser compreendidas para que o leitor possa criar a sua aplicação em um servidor real.

Para que possamos publicar a aplicação, é necessário escolher um serviço de hospedagem cloud, que possa oferecer acesso direto ao servidor. Isso é diferente dos planos de hospedagem que oferecem cPanel ou Plesk.

Dos mais variados serviços existentes no mercado, nós escolhemos a *Digital Ocean* por possuir uma interface simples de administração e um preço bom para o plano mais simples (atualmente 5 dólares). Também existe um cupom de desconto que lhe dará \$10.00 dólares em crédito. Desta forma, você poderá usar o serviço da *Digital Ocean* com dois meses gratuitos.

Nesta obra, criaremos o domínio ‘practicalangular2.com’, você deve trocar para o seu domínio sempre que for necessário. Usamos o serviço `name.com` para criar e gerenciar domínios, caso queira usá-lo também, use o cupom [https://www.name.com/referral/2794d9¹](https://www.name.com/referral/2794d9) para ganhar \$5.00 em créditos na criação do seu próprio domínio. Pode-se criar qualquer domínio, no site que você está acostumado a registrá-los.

11.1 Criando a conta na Digital Ocean

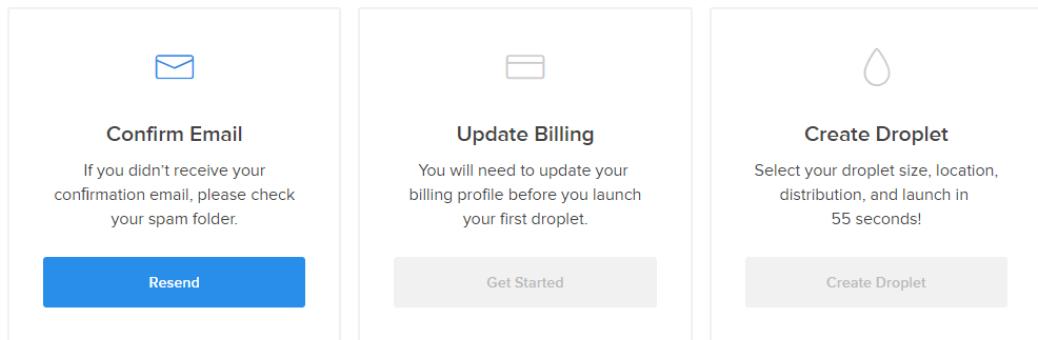
O primeiro passo para criar uma conta na Digital Ocean é usar o cupom de desconto, clicando no link [https://m.do.co/c/0812e52d3f24²](https://m.do.co/c/0812e52d3f24). Após clicar no link, clique no botão “Sign Up” para

¹<https://www.name.com/referral/2794d9>

²<https://m.do.co/c/0812e52d3f24>

criar a sua conta. Após fornecer o seu email e senha, surge a tela de configuração semelhante a exibida a seguir:

Welcome to DigitalOcean! Please confirm your email.



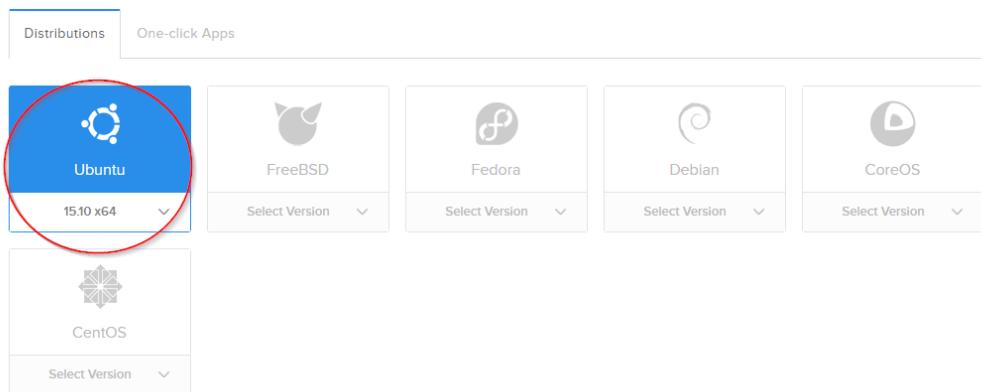
Nesta tela você irá confirmar o seu email, depois atualizar as suas informações de pagamento (necessita cartão de crédito ou paypal) e então clicar no botão **Create Droplet**. Na Digital Ocean, um *droplet* é um servidor que estará sempre ativo, e será nele que iremos publicar a aplicação.

11.2 Criando o droplet (servidor)

Clique no botão **Create Droplet** para iniciar o processo de criação do servidor. A primeira escolha a fazer é sobre a imagem do servidor, ou seja, qual distribuição iremos usar. Escolha a versão **Ubuntu 15.10 x64**, conforme a figura a seguir:

Create Droplets

Choose an image [?](#)



Após escolher a distribuição, escolha o preço. Vamos optar pelo mais barato, de \$5.00, com isso pode-se testar o serviço por dois meses:

Choose a size

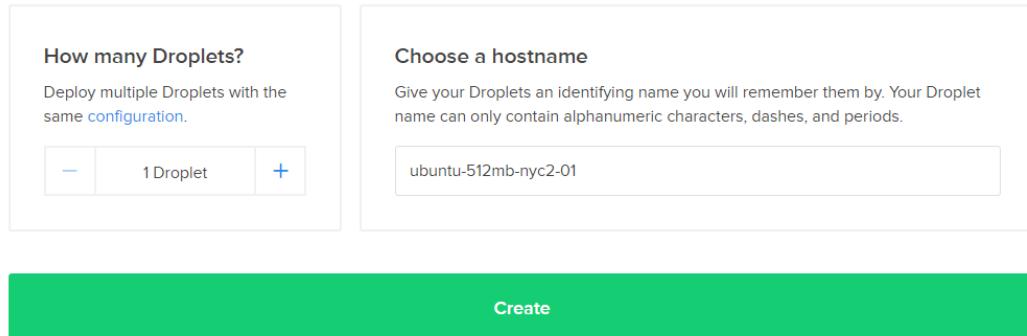
\$5/mo \$0.007/hour 512 MB / 1 CPU 20 GB SSD Disk 1000 GB Transfer	\$10/mo \$0.015/hour 1 GB / 1 CPU 30 GB SSD Disk 2 TB Transfer	\$20/mo \$0.030/hour 2 GB / 2 CPUs 40 GB SSD Disk 3 TB Transfer	\$40/mo \$0.060/hour 4 GB / 2 CPUs 60 GB SSD Disk 4 TB Transfer	\$80/mo \$0.119/hour 8 GB / 4 CPUs 80 GB SSD Disk 5 TB Transfer
\$160/mo \$0.238/hour 16 GB / 8 CPUs 160 GB SSD Disk 6 TB Transfer	\$320/mo \$0.476/hour 32 GB / 12 CPUs 320 GB SSD Disk 7 TB Transfer	\$480/mo \$0.714/hour 48 GB / 16 CPUs 480 GB SSD Disk 8 TB Transfer	\$640/mo \$0.952/hour 64 GB / 20 CPUs 640 GB SSD Disk 9 TB Transfer	

O próximo passo é escolher o local onde o servidor será instalado. Escolha de acordo com a sua região mais próxima. Em **Select additional options** e **Add your SSH keys** deixe o valor padrão.

No item **How many Droplets?** escolha 1 Droplet e no item **Choose a hostname** você deverá escolher o nome do seu servidor. Não é necessário inserir o nome do domínio, então

deixaremos o valor padrão, conforme a imagem a seguir:

Finalize and create



Clique no botão Create para criar o seu servidor. Aguarde alguns minutos e verifique na sua caixa de email informações sobre o servidor recém criado, como a senha de root e o seu ip.

11.3 Configurando o acesso SSH

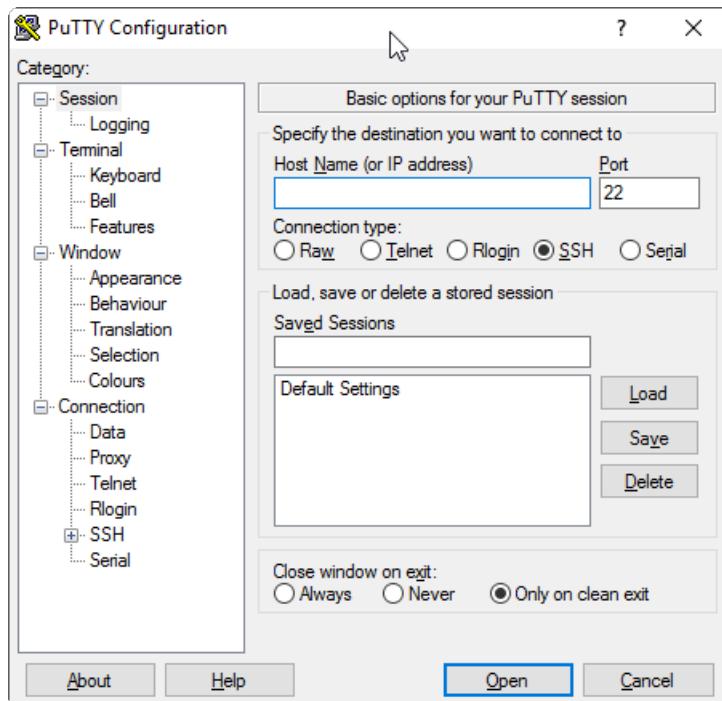
Após criar o servidor, você receberá um email com informações como o IP Adress, username e password. Você usará estas informações para realizar uma conexão ssh no servidor.

Para usuários Windows, recomendamos a instalação do programa [Putty](#)³. Para usuários Linux, o acesso SSH é realizado através da linha de comando `ssh user@ip`. Como usuário Linux sabemos que você possui o conhecimento necessário para operar o SSH, então vamos dar um foco maior aos usuários Windows.

Após instalar o [Putty](#)⁴, abra-o para ter uma interface semelhante a figura a seguir:

³<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

⁴<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

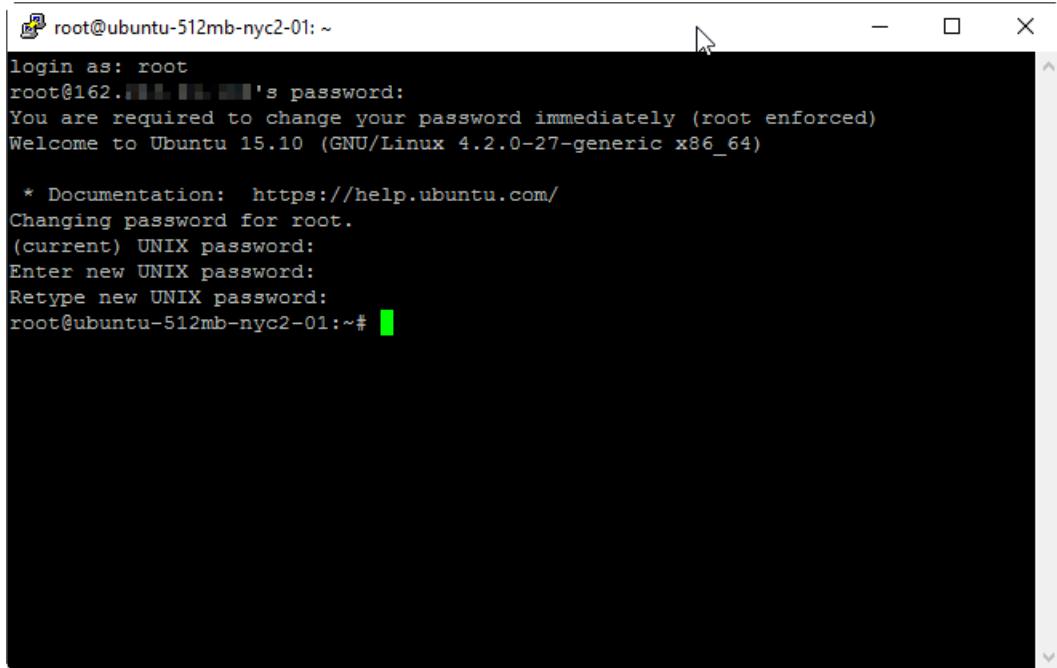


No campo host name (or ip address), insira o ip do seu servidor (está no email que a Digital Ocean lhe enviou ao criar o droplet). Em Port deixe a 22, que é a porta padrão de conexão SSH. Clique no botão Open e clique em Yes na mensagem que surge pelo Putty. Surge então a tela de login do seu servidor, no qual você deve informar o login e a senha que está no email.



Dica: Copie a senha e cole no putty usando o botão direito do mouse. Basta clicar uma vez no botão direito do mouse para automaticamente colar o texto.

Após logar, será necessário redefinir a sua senha de root. use uma senha forte com letras e números, pois o usuário root tem controle total sobre o seu servidor. O resultado do login até a mudança de senha é semelhante a figura à seguir:



```
root@ubuntu-512mb-nyc2-01: ~
login as: root
root@162.***.***.***'s password:
You are required to change your password immediately (root enforced)
Welcome to Ubuntu 15.10 (GNU/Linux 4.2.0-27-generic x86_64)

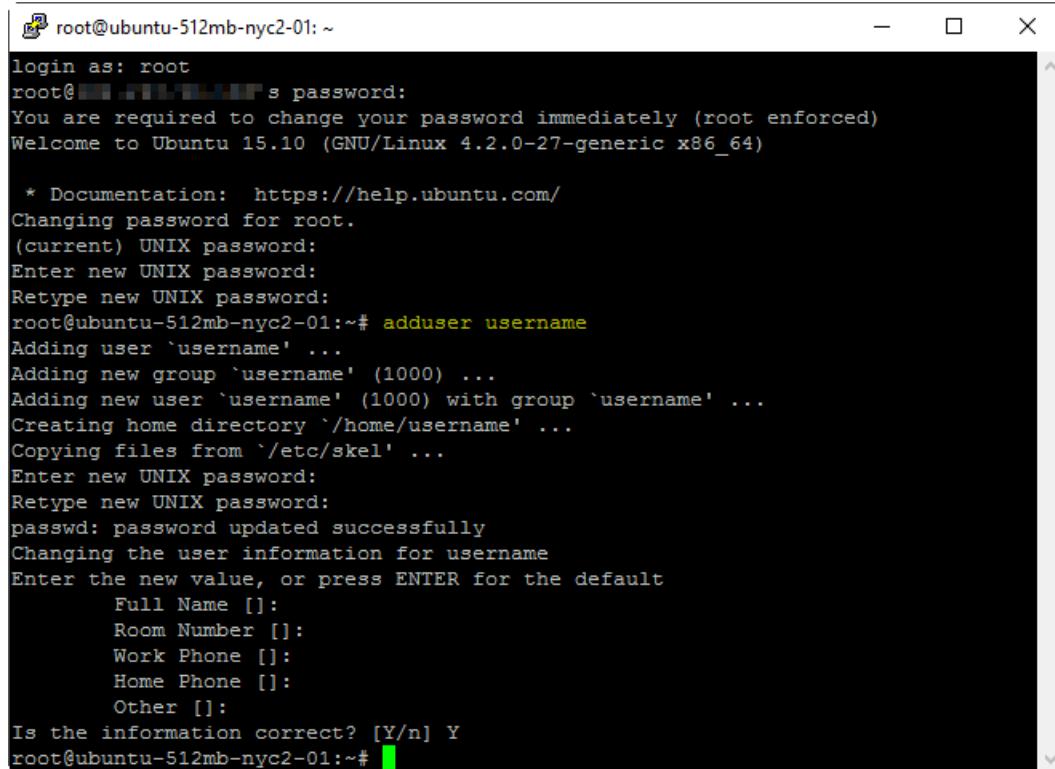
 * Documentation: https://help.ubuntu.com/
Changing password for root.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
root@ubuntu-512mb-nyc2-01:~#
```

11.4 Criando o usuário

A primeira ação deve ser feita no novo servidor é criar um usuário com um acesso mais restrito ao sistema. Será neste usuário que incluiremos também o sistema blog que criamos no capítulo 8.

Para incluir um usuário, digite:

```
# adduser username
```



```
root@ubuntu-512mb-nyc2-01: ~
login as: root
root@[REDACTED] s password:
You are required to change your password immediately (root enforced)
Welcome to Ubuntu 15.10 (GNU/Linux 4.2.0-27-generic x86_64)

 * Documentation: https://help.ubuntu.com/
Changing password for root.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
root@ubuntu-512mb-nyc2-01:~# adduser username
Adding user `username' ...
Adding new group `username' (1000) ...
Adding new user `username' (1000) with group `username' ...
Creating home directory `/home/username' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for username
Enter the new value, or press ENTER for the default
    Full Name []:
    Room Number []:
    Work Phone []:
    Home Phone []:
    Other []:
Is the information correct? [Y/n] Y
root@ubuntu-512mb-nyc2-01:~#
```



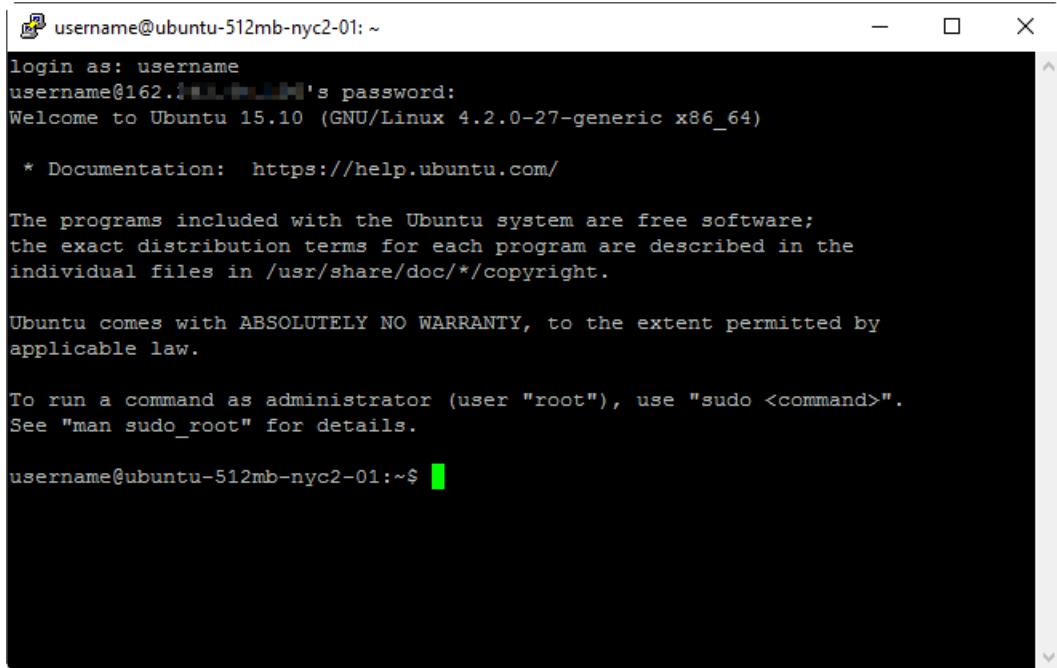
Pode-se usar o nome de usuário que desejar, insira o seu nome por exemplo. Nesta obra, usaremos `username`.

Após criar o usuário, vamos dar um pouco de poderes a ele pelo comando `sudo`. Se você não sabe o que é `sudo`, verifique neste [link⁵](#) o seu significado.

```
# adduser username sudo
Adding user `username' to group `sudo'
Done.
```

Com o usuário criado, podemos relogar com ele. Feche o putty (ou digite `exit`) e logue novamente, agora com o usuário que acabou de criar:

⁵<https://en.wikipedia.org/wiki/Sudo>

A screenshot of a terminal window titled "username@ubuntu-512mb-nyc2-01: ~". The window shows the standard Ubuntu 15.10 login screen with the following text:

```
login as: username
username@162.111.111.111's password:
Welcome to Ubuntu 15.10 (GNU/Linux 4.2.0-27-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

username@ubuntu-512mb-nyc2-01:~$
```

Através deste usuário faremos todas as alterações no servidor necessárias para que possamos instalar a aplicação blog.

11.5 Instalando o git

Usaremos o git para obter o código fonte da aplicação blog. Para instalar o git, faça:

```
$ sudo apt-get install git
```

Após instalar o git, podemos obter o código fonte da aplicação com o seguinte comando:

```
$ git clone https://github.com/danielschmitz/angular2-codigos.git
```

Todo o código fonte da obra será baixado, mas precisamos apenas do diretório blog que está no capítulo 8. Vamos copiar o diretório blog para o diretório raiz do usuário. Para fazer isso, execute o seguinte comando:

```
$ cp angular2-codigos/08/blog /home/username/ -R
```

A aplicação blog na qual queremos publicar estará no diretório /home/username/blog, conforme a figura a seguir:



The screenshot shows a terminal window on an Ubuntu system. The command 'ls' is run in the directory '/home/username/'. The output shows two subdirectories: 'angular2-codes' and 'blog'. Inside the 'blog' directory, files 'app', 'model', 'package.json', 'public', 'server.js', and 'tsconfig.json' are listed. The terminal window has a dark background with light-colored text. The title bar shows the user's name and the host.

```
username@ubuntu-512mb-nyc2-01: ~
username@ubuntu-512mb-nyc2-01:~$ ls
angular2-codes  blog
username@ubuntu-512mb-nyc2-01:~$ ls blog
app  model  package.json  public  server.js  tsconfig.json
username@ubuntu-512mb-nyc2-01:~$
```

11.6 Instalando Node

O node ainda não está instalado no servidor. Para a sua instalação, use o seguinte comando:

```
$ apt-get install curl
$ curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
$ sudo apt-get install --yes nodejs
```

Após a instalação do node, devemos executar o seguinte comando:

```
$ sudo ln -s /usr/bin/nodejs /usr/bin/node
```

Este comando cria um link simbólico de nodejs para node.

11.7 Instalando o nginx

Em sua definição mais simples, o nginx é um servidor web capaz de processar as requisições do servidor. Usaremos o nginx em conjunto com o node. Para instalar o nginx, faça:

```
$ sudo apt-get install nginx
```

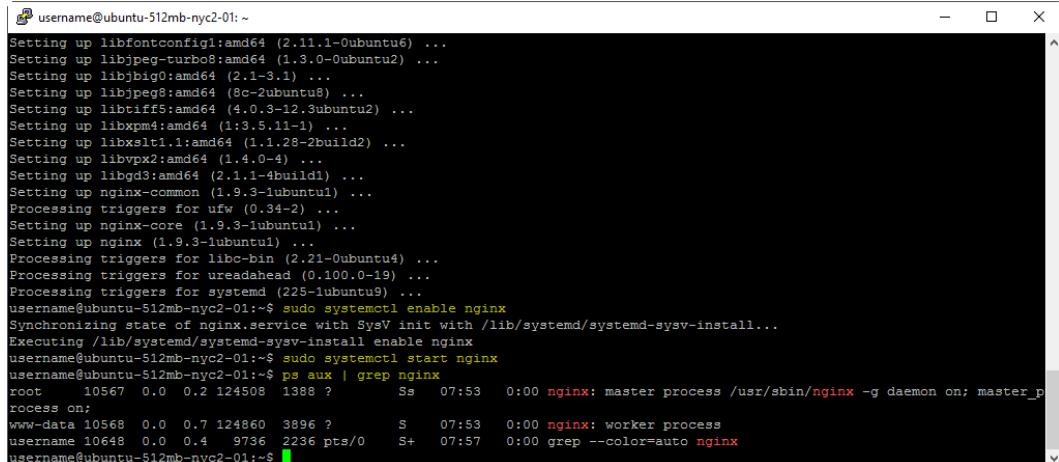
Após a instalação, precisamos configurar o nginx como um serviço do servidor. Ou seja, ele precisa estar ativo quando o servidor for reiniciado. O Ubuntu 15 (e a maioria das distribuições linux) usa o sistema systemctl para isso, e para ativá-lo, devemos executar os seguintes comandos:

```
$ sudo systemctl enable nginx  
$ sudo systemctl start nginx
```

Após essa configuração, pode-se verificar se o serviço nginx está ativo pelo comando:

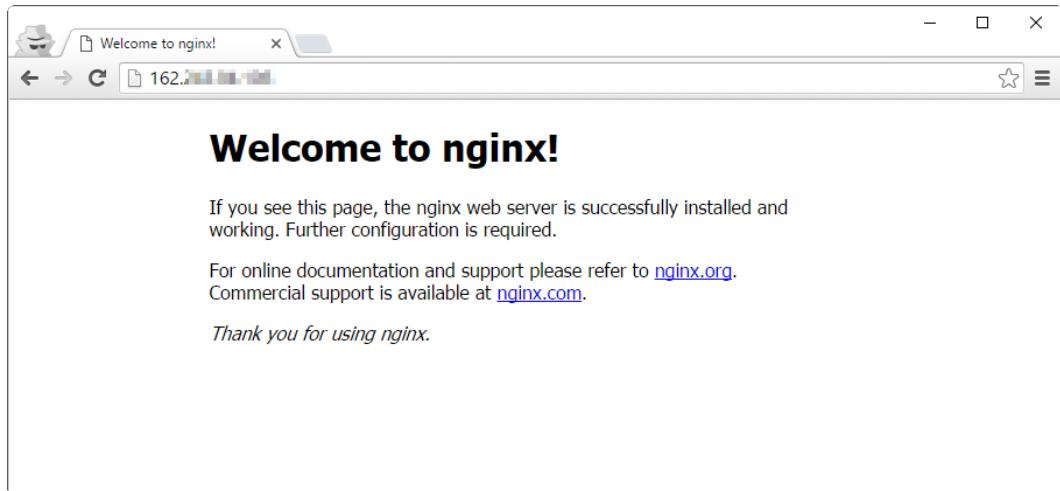
```
$ ps aux | grep nginx
```

Os processos serão exibidos, conforme a figura a seguir:



```
username@ubuntu-512mb-nyc2-01: ~  
Setting up libfontconfig1:amd64 (2.11.1-0ubuntu6) ...  
Setting up libjpeg-turbo0:amd64 (1.3.0-0ubuntu2) ...  
Setting up libjbig0:amd64 (2.1-3.1) ...  
Setting up libjpeg8:amd64 (8c-2ubuntu8) ...  
Setting up libtiff5:amd64 (4.0.3-12.3ubuntu2) ...  
Setting up libxpm4:amd64 (1:3.5.11-1) ...  
Setting up libxs1t1.1:amd64 (1.1.28-2build2) ...  
Setting up libvpx2:amd64 (1.4.0-4) ...  
Setting up libgd3:amd64 (2.1.1-4build1) ...  
Setting up nginx-common (1.9.3-1ubuntu1) ...  
Processing triggers for ufw (0.34-2) ...  
Setting up nginx-core (1.9.3-1ubuntu1) ...  
Setting up nginx (1.9.3-1ubuntu1) ...  
Processing triggers for libc-bin (2.21-0ubuntu4) ...  
Processing triggers for ureadahead (0.100.0-19) ...  
Processing triggers for systemd (225-1ubuntu9) ...  
username@ubuntu-512mb-nyc2-01:~$ sudo systemctl enable nginx  
Synchronizing state of nginx.service with SysV init with /lib/systemd/systemd-sysv-install...  
Executing /lib/systemd/systemd-sysv-install enable nginx  
username@ubuntu-512mb-nyc2-01:~$ sudo systemctl start nginx  
username@ubuntu-512mb-nyc2-01:~$ ps aux | grep nginx  
root      10567  0.0  0.2 124508  1388 ?        Ss   07:53  0:00 nginx: master process /usr/sbin/nginx -g daemon on; master_p  
rocess on;  
www-data  10568  0.0  0.7 124860  3896 ?        S    07:53  0:00 nginx: worker process  
username 10648  0.0  0.4   9736  2236 pts/0   S+   07:57  0:00 grep --color=auto nginx  
username@ubuntu-512mb-nyc2-01:~$
```

Neste ponto pode-se realizar um teste no navegador, bastando acessar o ip do servidor, como no exemplo a seguir:



11.8 Instalando os módulos do node

Como foi abordado nos capítulos anteriores, o arquivo package.json contém as informações sobre o projeto, dentre elas das suas dependências. Para instalar as dependências do projeto blog, devemos executar o seguinte comando:

```
$ cd /home/username/blog  
$ npm install
```

Após executar o npm install, verifique que o diretório node_modules foi criado, e as bibliotecas adicionadas.

11.9 Recompilando os arquivos TypeScript

Vamos recompilar os arquivos TypeScript para javascript. Primeiro, devemos adicionar o typeScript no servidor de forma global, com o seguinte comando:

```
$ sudo npm install typescript -g  
$ cd /home/username/blog  
$ tsc
```

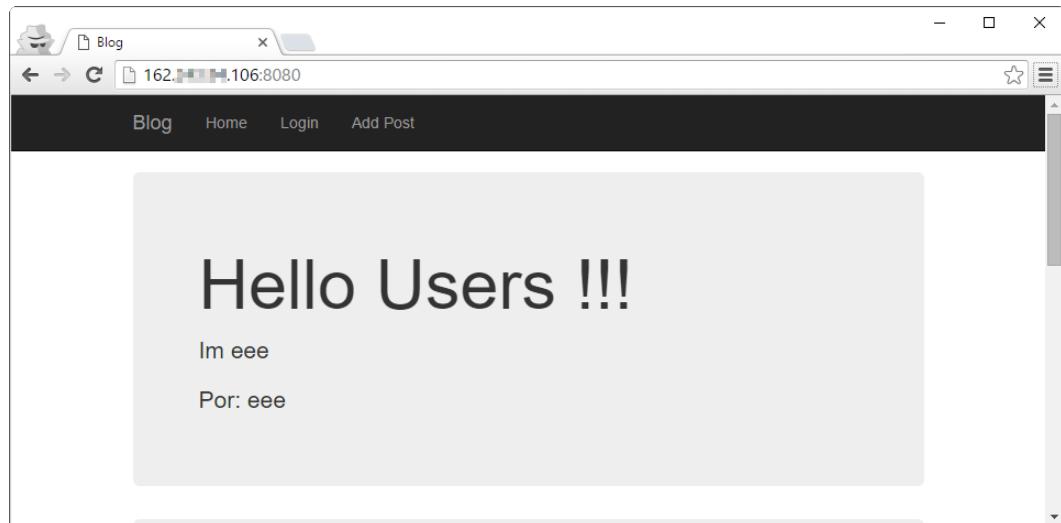
Os arquivos TypeScript são recompilados e a mensagem Compilation complete. Watching for file changes. surge caso o tsconfig.json esteja com a opção watch=true.

11.10 Teste inicial

Para testar a aplicação no navegador, execute o seguinte comando:

```
$ cd /home/username/blog  
$ node server.js
```

A mensagem “Listem 8080” surgirá, indicando que o servidor express está funcionando na porta 8080. Isso significa que podemos acessar a aplicação através do ip e sua porta, conforme a imagem a seguir:



11.11 Integração entre nginx e node

Já sabemos que o node pode ser executado via linha de comando, através do `node server.js` foi possível fazer com que a aplicação blog fosse acessada. Mas o nosso objetivo é realizar uma integração entre o nginx e o node, e fazer com que a porta 80, que é a porta padrão para acesso http, use o node.

Para isso, é preciso realizar dois procedimentos:

- 1- Adicionar o `node server.js` como um serviço do servidor
- 2- Configurar o nginx para executar este serviço

Primeiro, vamos adicionar o node como um serviço do servidor. Isso fará com que ele seja executado quando a máquina é religada, ou quando algum problema ocorre no node, fazendo com que o serviço seja reexecutado.

Como estamos usando o Ubuntu 15, usamos novamente o *systemd* para adicionar este serviço. Através do editor de textos nano, vamos adicionar o seguinte arquivo:

```
$ sudo nano /etc/systemd/system/node-app-1.service
```

Com o nano aberto, copie este código e cole:

```
[Service]
ExecStart=/usr/bin/node /home/username/blog/server.js
Restart=always
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=node-app-1
User=username
Group=username
Environment=NODE_ENV=production PORT=5000

[Install]
WantedBy=multi-user.target
```

Este arquivo `node-app-1.service` é uma configuração que irá, de acordo com o parâmetro `ExecStart` executar o comando `/usr/bin/node /home/username/blog/server.js`. Outros parâmetros são adicionados, como por exemplo o `Environment=NODE_ENV=production`, que pode ser usado na sua aplicação.

Outro parâmetro importante é `PORT=5000`, configurando a porta do serviço. Se observar no arquivo `server.js` a definição da porta é feita pelo código `var port = process.env.PORT || 8080`, neste momento a porta 5000 será utilizada.

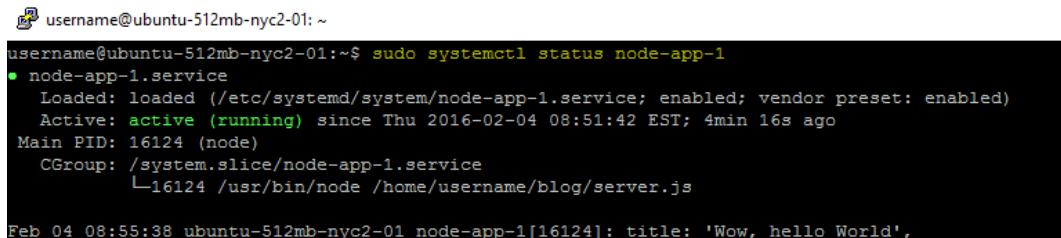
Com o arquivo criado, podemos iniciar e ativar o serviço, através do seguinte comando:

```
$ sudo systemctl start node-app-1
$ sudo systemctl enable node-app-1
```

Agora o serviço `node-app-1` está ativo. Pode-se verificar isso através do seguinte comando:

```
$ sudo systemctl status node-app-1
```

Através deste comando, tem-se a seguinte resposta:



```
username@ubuntu-512mb-nyc2-01:~$ sudo systemctl status node-app-1
● node-app-1.service
  Loaded: loaded (/etc/systemd/system/node-app-1.service; enabled; vendor preset: enabled)
  Active: active (running) since Thu 2016-02-04 08:51:42 EST; 4min 16s ago
    Main PID: 16124 (node)
   CGroup: /system.slice/node-app-1.service
           └─16124 /usr/bin/node /home/username/blog/server.js

Feb 04 08:55:38 ubuntu-512mb-nyc2-01 node-app-1[16124]: title: 'Wow, hello World!',
```

Pode-se testar também através do navegador, acessando o ip do servidor e a porta 5000.

Com a primeira parte pronta, temos o serviço node sendo executado na porta 5000. Precisamos agora ativar o nginx e fazer o que chamamos de “proxy”, isso é, fazer com que quando a porta 80 for chamada, o nginx direcione para a porta 5000.

Para isso vamos reescrever o comportamento do site padrão do nginx. Execute o seguinte comando:

```
$ sudo mv /etc/nginx/sites-available/default /etc/nginx/sites-available/default-original
$ sudo nano /etc/nginx/sites-available/default
```

No nano, adicione a seguinte configuração:

```
upstream node_server {
    server 127.0.0.1:5000 fail_timeout=0;
}

server {
    listen 80 default_server;
    listen [::]:80 default_server;

    index index.html index.htm;

    server_name _;
```

```
location / {  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
    proxy_redirect off;  
    proxy_buffering off;  
    proxy_pass http://node_server;  
}  
}
```

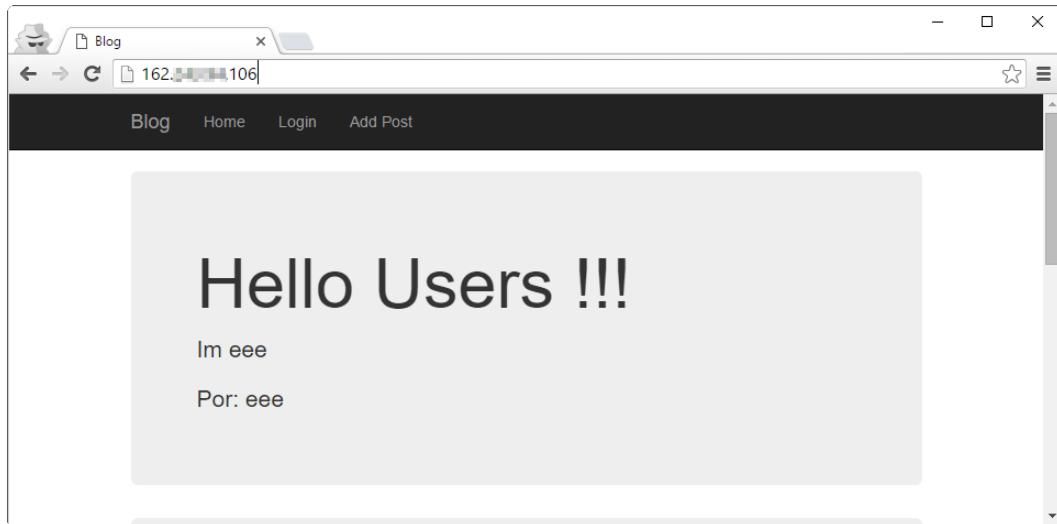
Nesta configuração, usamos o comando `upstream` para criar um servidor chamado `node_server`. Este servidor irá apontar para a própria máquina na porta 5000, a mesma que configuramos no serviço `node-app-1`.

O servidor “principal” está escutando a porta 80, e definimos através do `location /` o proxy para o servidor `node_server`. Através desta configuração, fazemos que com que o servidor nginx escute a porta 80 e faça um proxy a porta 5000, executando assim o node e consequentemente o express.

Após alterar a configuração, precisamos reiniciar o nginx com o seguinte comando:

```
$ sudo systemctl restart nginx
```

Com o nginx reiniciado, podemos finalmente acessar a aplicação somente com o ip do servidor, conforme a figura a seguir:



11.12 Algumas considerações sobre node+nginx

É importante frisar que esta configuração visa dar um passo inicial ao seu estudo sobre o gerenciamento de sistemas com node e nginx. Existem pontos extras que precisam ser reavaliados para tornar a sua aplicação mais robusta. Por exemplo, requisições estáticas neste momento estão sendo processadas pelo Node através do `express.static`, mas o melhor caminho é configurar o nginx para gerenciar estas requisições, ao invés do node. Conexões SSL também necessitam ser gerenciadas pelo nginx. Estas configurações envolvem passos extras que não serão abordados nesta obra. Nossa objetivo é mostrar que podemos criar aplicações em node e disponibilizá-las na web.

11.13 Domínio

Se você criou um domínio para a sua aplicação, chegou o momento de configurá-lo. Para isso, acesse o painel de administração do seu domínio, que varia de acordo com o seu provedor. Usamos aqui o `name.com`, que vc pode adquirir um cupom de \$5.00 [nesta url⁶](https://www.name.com/referral/2794d9).

Após criar o domínio, acesse o painel de DNS RECORDS, conforme a seguir:

⁶<https://www.name.com/referral/2794d9>

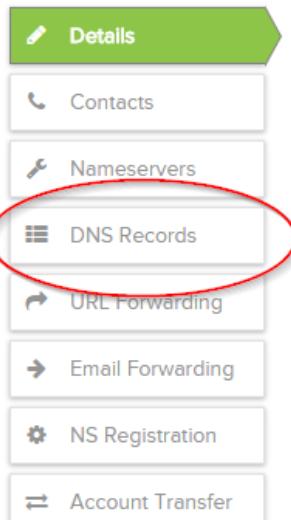
practicalangular2.com

Annual Renewal: \$10.99 - Renew Domain

Domain Expires: 04 Feb 2017

Auto Renew: Inactive 

Whois Privacy: Add to Cart



Domain Details

Domain name: practicalangular2.com

Domain lock:  Transfer Lock

Transfer Auth Code:  Show Code

Nameservers: Edit Nameservers

ns1.name.com, ns2.name.com, ns3.name.com, ns4.name.com

DNS hosted: Yes Update DNS records

Registrar: name.com

Website hosted: No

Auto renew: Inactive 

Whois Privacy: Add to Cart

Na tela Edit DNS Record, adicione duas entradas informando o ip do servidor que a Digital Ocean criou para você, da seguinte forma:

Edit DNS Records: practicalangular2.com

DNS Templates							
	Type	Host	Answer	TTL	Prio	Created	Actions
<input type="checkbox"/>	A	practicalangular2.com	162.24.106	300	N/A	2016-02-04	<button>Edit</button> <button>Delete</button>
<input type="checkbox"/>	CNAME	www.practicalangular2.com	practicalangular2.com	300	N/A	2016-02-04	<button>Edit</button> <button>Delete</button>
	A	.practicalangular2.com		300	N/A		<button>Add Record</button>

Com estas duas entradas, aguarde algumas horas para a publicação DNS ocorrer e acesse o domínio que você criou, não mais o IP. O blog que criamos em nossa obra será carregado:

Edit DNS Records: practicalangular2.com

DNS Templates ▾							
Type	Host	Answer	TTL	Prio	Created	Actions	
A	practicalangular2.com	162.255.106	300	N/A	2016-02-04	<button>Edit</button>	<button>Delete</button>
CNAME	www.practicalangular2.com	practicalangular2.com	300	N/A	2016-02-04	<button>Edit</button>	<button>Delete</button>
A	.practicalangular2.com		300	N/A		<button>Add Record</button>	

Você pode testar o blog neste momento, acessando <http://www.practicalangular2.com/>⁷

11.14 Conclusão

Após criarmos a aplicação em nosso ambiente de desenvolvimento, conseguimos executar todos os passos necessários para ter a mesma aplicação no servidor de produção, ligando um domínio ao servidor que criamos pela Digital Ocean, e usando nginx+node para hospedar a aplicação.

Como estamos publicando a obra pela Leanpub, é possível estender o livro com mais conteúdo, mas para isso precisamos do feedback da comunidade. Acesse a página desta obra em <https://leanpub.com/livro-angular2> e clique em “Discuss this Book”.

⁷<http://www.practicalangular2.com/>