LÓGICA DE PROGRAMAÇÃO PARA

FRONT END

Lógica de programação para Front-End

Dorian Torres

Por que você deve ler este livro

Ao longo de minha carreira profissional como desenvolvedor de sistemas, trabalhando com tecnologias de *back* e *front-end,* nos mais diversos nichos (desde aplicações financeiras, ERP, bases de dados, *e-commerce,* EAD), me deparei com os mais diversos tipos de situações, problemas e cenários.

Entre a evolução das diversas tecnologias envolvidas nesses cenários, acompanhei o andamento de linguagens e tecnologias diversas, como:

- PHP

- Javascript

- HTML5 e CSS3

- JAVA

- C#

- Design Patterns,

- Processos de desenvolvimento

- Surgimento de frameworks como React, Vue.JS, Angular, Node JS

- Ferramentas de desenvolvimento 3D

- Fluxos de plataformas de pagamento

- Módulos de gerenciadores de conteúdo

E como é natural, toda essa evolução costuma gerar confusão e dificuldade no aprendizado, principalmente em quem está iniciando na área.

Afinal, hoje em dia nos deparamos com uma característica importante, que é, ao mesmo tempo, a solução e problema: A quantidade de informações que temos disponível.

Chamo de **solução** pois nos dá um mar de possibilidades para pesquisa e aprendizado, nos mais diversos canais: livros, *audiobooks*, vídeos no *Youtube*, artigos de blog, exemplos em vários idiomas, códigos abertos para serem baixados e testados, editores de texto diversos.

E também vejo como um grande problema: Se você enfrenta alguma dificuldade em organizar suas ideias e pensamentos, e construir uma boa estratégia de aprendizado.

A overdose de informação é um problema complicado…

Por causa disso tudo, eu resolvi reunir toda minha experiência, conhecimento, técnicas, estratégias de aprendizado, e então, elaborar um material rico e cheio de conteúdo, com exemplos e formas de elaboração distintas.

A intenção é **facilitar** seu aprendizado.

Eu quero fazer com que este livro seja um manual que abra as portas e janelas em sua mente, que ajude seu cérebro a fazer as conexões neurais da forma mais eficiente possível.

E isso inclui utilizar de forma produtiva e positiva os demais materiais ricos que já existem por aí.

Somos todos uma grande comunidade, reunida para somar, e multiplicar projetos e soluções.

Leia livros, assista aos vídeos, absorva conhecimento de outros profissionais. Filtre o que for necessário e aplique o que fizer sentido para você!

Afinal, nossa mente nunca se farta de conhecimento.

Introdução

Estou escrevendo este livro em 2020.

O ano em que a programação é uma das atividades mais valorizadas do mundo.

Lado a lado com a venda, que é a atividade responsável por movimentar os poderes monetários no planeta.

Hoje, quem entende de programação, fala a língua dos grandes.

Fala a língua do homem e da máquina.

Entende criatura e criador.

Se você refletir por alguns segundos sobre isso, perceberá todo o poder que tem em mãos.

Diante disso, você está com uma **oportunidade de ouro** nas mãos.

Este material vai te dar os passos necessários para você aprender a dominar a lógica de programação quando o assunto é desenvolvimento front-end.

Existem alguns conceitos mais genéricos (que podem ser utilizados em back ou em front-end) e outros, são específicos de cada área de atuação.

Se você é ou almeja ser um desenvolvedor front-end, esse livro é pra você!

Boa leitura e bons estudos!

Índice

Estrutura da computação

Se você é ou pretende ser um bom programador, é importante que entenda como os computadores funcionam, em sua estrutura básica.

A intenção dessa parte do livro é ir direto ao ponto, de forma objetiva e muito clara. Afinal, o que buscamos são resultados práticos.

É claro que não temos a intenção de perder anos da nossa vida entendendo como o computador funciona.

O mundo em que vivemos é dinâmico, e nós também somos.

As informações contidas nessa seção do livro podem ser bem básicas e repetitivas para pessoas que já trabalham na área ou já tem algum conhecimento em desenvolvimento de software.

Mas, se você é iniciante, ou atua mais com *front-end,* pode ser que essas informações sejam muito úteis para o seu dia a dia.

Então, chega de enrolação! Bora pro que interessa:

Como o computador funciona

O que conhecemos como computador, é tecnicamente chamado de CPU – *Central Process Unit* e significa *Unidade Central de Processamento.*

É na CPU que todo acontece. Todos os cálculos, somas, atribuições, são feitas ali.

Esses cálculos são transformados em impulsos elétricos positivos ou negativos, que são **entradas** de circuitos lógicos diversos.

Esses circuitos lógicos transformam essas **entradas** em **saídas,** e geram os resultados que queremos (desde abrir um software até acionar uma impressora).

É uma “dança” digital muito bem orquestrada.

Para que nós, humanos, fôssemos capazes de manipular os impulsos elétricos (positivos e negativos), criamos um sistema numérico que permite apenas duas possibilidades: 0 e 1. Ligado ou desligado.

Isso se chama **sistema binário**.

Entenda que o sistema binário pode ser interpretado como o meio de campo entre homem e máquina.

É ele que permite que a máquina entenda o que queremos, e que a gente entenda o que ela produz.

Linguagem de montagem

O sistema binário é um grande avanço na forma que nos relacionamos com a máquina. Porém, a solução trouxe consigo outro problema: a evolução do sistema.

Era fácil manipular dados binários enquanto tínhamos apenas 5, 8, 20, ou 100 possibilidades.

Mas… como poderíamos trabalhar com dados grandes, como é o caso de filmes, que tem milhões, ou trilhões de possibilidades de “zeros” e “uns”?

E mais: como um homem ou uma mulher seria capaz de conversar com a máquina (programar) a esse nível?

Esse problema fez com que fosse desenvolvida a **linguagem de montagem.**

Imagine uma linguagem de montagem como uma grande tabela que relaciona nomes com o seu equivalente em bits (dados binários).

Vamos trabalhar com um exemplo bem simples. Imagine dois valores: SOMA e SUBTRAÇÃO.

SOMA representa o bit “1”.

SUBTRAÇÃO representa o bit “0”. Da seguinte forma:

SOMA – 0000 0001

SUBTRAÇÃO – 0000 0000

Lembra da CPU? Lá na CPU existe um sistema chamado **compilador de linguagem de montagem.**

Esse sistema é como um dicionário. Ele verifica a palavra que entrou, e converte a saída para o seu equivalente em binário.

Então, quando o programador inserisse a palavra “SOMA”, o compilador tentava encontrar nesse dicionário. Se encontrasse, fazia a conversão e enviava os dados já convertidos.

Agora, imagine essa tabela com milhões, ou bilhões de instruções diferentes?

Seria impossível para um ser humano programar tudo isso em números binários.

Mas, é perfeitamente possível quando podemos utilizar uma linguagem que se parece mais com a que utilizamos para nos comunicarmos.

Linguagens de alto nível

Mesmo depois que a linguagem de montagem tornou-se popular, o problema continuou crescendo. Afinal, as instruções eram confusas e complexas.

Os problemas sociais e gerais da humanidade demandavam uma maior produtividade (mais resultados com menor tempo de execução).

Então, novamente, foi necessário pensar em soluções ainda mais avançadas.

Para isso, a linguagem tinha que ser cada vez menos parecida com a linguagem de máquina, e cada vez mais parecida com a linguagem humana.

E mais do que isso: era necessário que as máquinas fizessem cada vez mais o trabalho automático, para que nós, cada vez mais, pudéssemos dar atenção à criações de soluções inteligentes (e não nos prendermos a resolver problemas de comunicação homem-máquina).

Então, criou-se uma nova “camada” de conversação.

O compilador

Agora, imagine um compilador semelhante ao da linguagem de máquina que falamos antes.

Só que muito mais inteligente, eficiente e com mais recursos.

Agora, esse compilador transforma uma linguagem “superior” na linguagem de montagem (que falamos antes).

Isso possibilitou que criássemos as **linguagens de alto nível**.

Aquelas com condições e estruturas de fácil entendimento, e com uma linguagem muito semelhante ao nosso idioma natural.

Linguagem C foi uma das primeiras linguagens de alto nível construídas.

(E sim, sabemos que hoje, linguagem C não parece tão alto nível assim… (risos)).

Mas, é graças a essa linguagem que temos o mundo de possibilidades que temos hoje, seja em desenvolvimento web, desktop, aplicações mobile, e por aí vai.

E isso não parou mais.

Dia após dia, novas linguagens, novas tecnologias e novas formas de programar foram surgindo.

As linguagens de programação nos permitem, hoje, escrever verdadeiros contos.

É isso que você vai aprender neste livro: Você vai escrever histórias com as linguagens de programação.

E isso é só o começo.

A base da programação

Se você nunca programou, essa parte é extremamente importante.

É a base de todo o funcionamento de linguagem de programação que você venha a ter contato.

Se você já programa em alguma linguagem, vai ajudar você a estruturar ainda mais os conceitos que você já conhece, e ampliar seus horizontes.

Eu tenho como premissa básica que a base de qualquer aprendizado, é dominar a estrutura do assunto. Isto é, como funciona, e por que funciona.

Sendo assim, arrisco dizer: se você entende essa base que será mostrada a seguir, será capaz de programar em qualquer linguagem front-end que você desejar.

O que é um programa?

Ao utilizar as linguagens de programação, nós escrevemos diversos códigos e salvamos em um arquivo de determinado tipo.

Pode ser .c, .java, .php, .js, .asp, .aspx, .cshtml, .py…

Dependendo da linguagem utilizada, você salva o arquivo em um formato diferente.

Esse conjunto de códigos foi escrito seguindo uma lógica de passos. Este passo a passo é chamado de **algoritmo.**

Sabendo disso, um programa é um conjunto completo de um ou vários algoritmos.

Quando colocamos um programa em execução, seja em um navegador, um servidor ou em um computador pessoal, ele é compilado e/ou interpretado.

E a partir do momento que entra em execução, ele é chamado de **processo**.

Processos utilizam recursos de **disco**, **processamento** e memória **RAM**.

O processamento é a capacidade (e velocidade) que a *CPU* tem de calcular e gerar o resultado das instruções.

Já ouviu falar que alguns programas são chamados “pesados” ?

Então, quando um software qualquer exige muito processamento (como uma renderização 3D). Este tipo de conteúdo precisa de um número muito grande de instruções sendo executadas e avaliadas em conjunto.

Se o computador não der conta, costuma ocorrer os travamentos e aquecimentos.

Naturalmente, os fatores envolvidos não são apenas processamento. O disco e a memória RAM também fazem parte disso.

São eventos em conjunto.

Interpretação, Compilação e Transpilação

Agora que já sabemos o que é um programa, é preciso saber como eles são transformados em processos e por fim, gerar resultados,

como exibir um gráfico, escrever um texto ou carregar uma imagem.

Isso se dá através da etapa que transforma o programa escrito em linguagem de alto nível em uma linguagem de baixo nível que a máquina possa entender (os zeros e uns, ou bits, ou impulsos elétricos).

Programas que são compilados precisam passar por um compilador antes de serem transformados em pacotes prontos para utilização.

Esse pacote costuma ser chamado de **build** pelos desenvolvedores.

Usamos esse processo no Javascript, quando trabalhamos com Node JS, por exemplo, que cria um servidor local, e compila o código e o transforma em um código para ser interpretado.

Na outra esfera, temos os programas interpretados. PHP por exemplo, é uma linguagem de servidor interpretada.

O javascript puro também é interpretado pelo navegador.

A principal diferença na interpretação, é o que o código vai sendo executado passo a passo, na ordem de execução que citamos antes: De cima para baixo, e da esquerda para a direita.

Na prática, vamos supor que temos um código com 5 blocos. Os 3 primeiros estão corretos e o bloco 4 tem um erro.

O navegador executará os 3 primeiros e mostrará o resultado na tela.

Ele encontrará o erro no bloco 4, e vai parar a execução.

O bloco 5 não será executado. Veja no exemplo a seguir, utilizando a interpretação de código de um navegador, a tentativa de acessar uma propriedade com nome errado:

html

```
<div id="saida1"></div>

<div id="saida2"></div>

<div id="saida3"></div>

<div id="saida4"></div>

<div id="saida5"></div>
```

Javascript

```
// Recuperando variáveis do HTML, de acordo com atributo id

var saida1 = document.getElementById('saida1');

var saida2 = document.getElementById('saida2');

var saida3 = document.getElementById('saida3');

var saida4 = document.getElementById('saida4');

var saida5 = document.getElementById('saida5');

saida1.innerText = "Texto exibido na saída 1";
```

saida2.innerText = "Texto exibido na saída 2";

saida3.innerText = "Texto exibido na saída 3";

saidaerrado4.innerTexteRRADO = "Texto exibido na saída 4";

saida5.innerText = "Texto exibido na saída 5";

Este mesmo exemplo, em um código compilado, como em uma aplicação feita em *NodeJS*, o console de erros exibiria o erro, a compilação não seria feita.

O primeiro caso, é um **erro de execução.**

O segundo, é um **erro de compilação.**

Na linguagem interpretada, é considerado um erro de execução, pois ela SEMPRE está em execução. A linguagem interpretada sempre é um processo em andamento.

Ela não passa por um compilador ou um comparador léxico (que determina se a linguagem está escrita de forma correta).

Erros de execução não acontecem apenas na interpretação. Um erro de interpretação clássico é a divisão por zero.

É comum que linguagens interpretadas permitam a divisão por zero, e elas retornam algum valor do tipo *NaN* (Not a Number – Número não número). Por isso, não geram erro de execução.

Linguagens compiladas, por sua vez, costumam disparar erro na tentativa da divisão por zero.

A *transpilação* é o nome dado a conversão do Javascript, e acontece em alguns cenários específicos, onde se utiliza um motor de renderização. Muito comum em projetos que utilizam Typescript e Angular por exemplo.

O Typescript, é o Javascript mas com a adição de tipos de dados (veja mais adiante), tornando-o uma linguagem fortemente tipada.

Mas, o navegador não conhece o Typescript. Ele só entende o Javascript. Então, nós desenvolvemos em um ambiente de desenvolvimento, como o **Node JS**. Então, screvemos todo o código em Typescript.

E nesse momento o ambiente (Node JS) “compila” o código e te mostra erros em tempo de compilação - ou seja - você não consegue liberar o código para ser utilizado, pois o motor de renderização não permite que um código com erros (que não passou nas verificações da compilação) seja executado corretamente.

Depois que ele foi devidamente compilado, esse código é convertido em um equivalente em Javascript, para que seja interpretado pelo navegador. Esse processo é chamado de *transpilação*.

Tipos de dados

De forma direta: Tipos de dados em programação é o que define a alocação de recursos, especificamente: a memória RAM.

De forma básica, os processos envolvidos nas aplicações utilizam recursos de processamento e memória. Esses, são calculados em bits.

Os tipos de dados falam ao computador de quanto espaço será necessário para utilizar aquele valor.

E existem padrões para isso. Cada tipo de dados tem sua própria fatia de espaço.

Quem já trabalhou com Linguagem C, deve se lembrar como era complexo o processo de alocação de memória.

Graças a evolução das tecnologias, hoje é tudo muito simples.

O que precisamos saber, é que cada informação tem um tipo de dado diferente. Em geral, temos as seguintes nomenclaturas para os tipos básicos:

CHAR

Caracteres individuais, como ‘a’, ‘e’, ‘b’)

STRING

Cadeia de caracteres, como “essa é a uma string com 36 caracteres”)

NUMBER

Números inteiros ou decima, positivo ou negativo.)

BOOL

Verdadeiro ou falso

ARRAY

O array em si não é um “tipo de dado”, é uma lista de dados que pode conter um ou vários tipos

Alguns tipos de dados podem ter comportamentos diferentes em linguagens diferentes.

Já outras podem ter tipos de dados que não existem em determinadas linguagens. Isso depende muito da tecnologia que está sendo utilizada.

Na sessão de Orientação a Objetos, você entenderá formas mais eficientes de estruturar tipos. Já que os tipos de dados falados anteriormente são os “tipos primitivos”, diferente das classes, que podem ser tanto criadas pelo programador, como ser preexistentes em algumas linguagens.

Variáveis e Constantes

As variáveis existem em quase todas as linguagens existentes.

Imagine as variáveis como caixinhas para guardar informação.

Então, quando eu quero armazenar um nome, eu crio uma caixinha chamada Nome, e coloco o nome dentro.

Quando quero armazenar um número, eu crio uma caixinha chamada Número, e coloco dentro.

É claro que utilizamos termos que sejam de fácil entendimento, para não gerar confusão.

Imagine um sistema de cadastro de alunos de até 16 anos. Este sistema tem dois campos:

- Nome do aluno

- Nome do responsável

Neste caso, eu posso criar as variáveis assim:

var nomeAluno;

var nomeResponsavel;

Assim, fica fácil para qualquer pessoa ler e entender do que se trata.

Evite criar variáveis com nomes genéricos, tipo:

Nome1, nome2, nome3 ou *x, y, z, a,b,c…*

Em sistemas grandes, isso pode causar problemas sérios.

É comum também que as variáveis acompanhem o tipo de dados. No javascript puro, você pode atribuir uma *string* e depois um número em uma mesma variável.

Porém, não é o recomendado.

Em linguagens fortemente tipadas (e isso acontece também com o Typescript), cada variável deve ter seu tipo. Geralmente, os tipos de variáveis são definidos com o nome do tipo antes, ou após dois-pontos (:).

Nome de variável com o tipo antes

String nomeAluno;

Nome de variável utilizando dois-pontos (utilizado no Typescript)

var nomeAluno: string;

Essa nomenclatura depende da linguagem utilizada.

Variáveis também podem ser declaradas e atribuídas.

A declaração de uma variável define apenas seu nome, ou nome e tipo.

A atribuição coloca o valor dentro da variável. No javascript, utiliza-se o sinal de igual (=) para isso.

É possível fazer a declaração e atribuição ao mesmo tempo. Aliás, às vezes é até necessário, depende das necessidades do bloco de código sendo implementado.

Uma **constante** tem uma mecânica muito parecida com as variáveis, no sentido de como utilizar. A grande diferença, é que não é possível mudar o valor. Por isso ela é chamada de constante.

Então, se você tentar atribuir um novo valor a uma constante já atribuída anteriormente, será gerado um erro.

É importante ressaltar, que a constante pode ser manipulada. Ou seja: você não pode atribuir, mas pode manipular, caso ela seja um objeto por exemplo. Veja:

```
const person = {
    name: 'Joao',
    idade: 35
};
```

Você **não pode** fazer isso:

```
person = { name: 'Luana', idade: 18 };
```

Mas pode fazer isso:

```
person.name = 'Luana';
person.idade = 18;
```

Isso é permitido, pois, você está mantendo a estrutura da constante, mas pode manipular propriedades dentro dela. Funciona de igual modo para listas e arrays. Um array pode ser uma constante e ainda assim, você conseguir adicionar e remover itens.

Em geral, falando de EcmaScript, é comum utilizarmos constante para grande parte dos valores que fazemos para manipular dados. Por exemplo: retorno de funções assíncronas, instâncias de classes, serviços, etc.

Mas, caso você precise alterar seu valor, lembre-se sempre de utilizar uma variável (com VAR ou LET).

Estruturas de dados

As estruturas de dados mais comuns em projetos front-end são:

- Enum
    - Utilizado para enumerar uma lista de itens em sequência ou tipo. Por exemplo:
    - *const Status = {*
        Ativo = 1,
        Inativo = 2
      }
- Arrays e listas
    - Para armazenar um conjunto de itens ou objetos
- Classes

- Em alguns casos, quando se utiliza uma linguagem tipada como o Typescript, é comum a utilização de classes para organizar e abstrair o código.

Operadores

Na programação, são os operadores que realizam os cálculos, fazem atribuições, condições e outras possibilidades. Os operadores são divididos em grupos.

Naturalmente, a lista de operadores aqui é para que você tenha uma ideia de como funciona. Consulte a documentação específica de cada linguagem para entender as particularidades e possíveis operadores adicionais.

Operadores aritméticos

+ = soma dois números. Em algumas linguagens, concatena (junta) strings.

- = subtrai dois números.

* = multiplicação

/ = divisão

Operadores lógicos

&& = AND (Verifica se duas condições são verdadeiras)

|| = OR (verifica se uma ou outra condição é verdadeira)

Operadores de atribuição

Você pode atribuir qualquer valor com o sinal de igual (=).

E você também pode calcular e atribuir qualquer operação aritmética.

Por exemplo:

+= (soma e atribui)

-= (subtrai e atribui)

++ (neste caso, soma e atribui 1. O 1 é implícito)

– (subtrai e atribui 1. O 1 é implícito)

Estruturas de código

As diversas estruturas de código (existentes em todas as linguagens de programação) é o que nos permite trabalhar com os dados, manipular valores, considerar condições, tomar decisões, automatizar tarefas.

As principais estruturas de código conhecidas e utilizadas são:

for (início; condição de parada; alteração)

O famoso laço de repetição “para”. É a estrutura básica para repetir dados.

Ela é baseada em 3 partes diferentes controladas por um número inteiro, geralmente: Início, condição de parada e regra de alteração.

Início = É o DE – De onde começar

Parada = É o PARA – PARA onde vai.

Alteração = Incremento ou decremento. Pode ser +1 -1, ou qualquer outra fórmula. Vale a observação: Dependendo da condição, o FOR nunca para. Fique atento!

Exemplo de uso:

```
var inicio = 0;
var parada = 10;

//Este código significa que vai começar em "inicio" (0). Vai executar enquanto "início" for menor que "parada" (inicio < parada). A cada vez que executar o código de dentro do for, será incrementado 1 em "inicio" (inicio++)
for(inicio, inicio < parada, inicio++) {
// Aqui vai sua lógica
}
```

while (condição)

O while também é uma estrutura de repetição, que se baseia apenas na condição de parada. É importante controlar a condição dentro do while, para que ele não se torne um laço infinito.

Exemplo:

```
var parada = 0;

while(parada < 10) {

   //sua lógica aqui. Depois, incrementa 1 na parada.

   parada ++;

}
```

if (condição)

O if é a estrutura que verifica se o resultado (condição) é TRUE ou FALSE.

Exemplo:

```
//Exemplo de retorno FALSE. Não entra na condição

var valor1 = 2;
var valor2 = 3;
var resultado = 10;
if (valor1 + valor2 == resultado) {
}

//Exemplo de retorno TRUE. Entra na condição
var valor1 = 2;
var valor2 = 3;
var resultado = 5;
if (valor1 + valor2 == resultado) {
}
```

switch (opção)

O switch existe como uma estrutura que define opções diversas. Atualmente, é uma estrutura de código pouco utilizada, devido aos padrões de código e *design patterns* (padrões de projeto) utilizados.

Dentro do **switch** usamos a estrutura **case** que define algo como “quando a opção for esta”.

A condição de parada do **switch** é o **break** ou **return.** Se a estrutura de parada não for utilizada, o **switch** executará todas as próximas opções, à partir da encontrada.

O **switch** também permite a estrutura **default,** que será executada caso nenhum **case** seja encontrado.

Vários **case´s** podem ser colocados juntos, buscando um mesmo retorno para dois casos diferentes. Essa regra também vale para o **default**.

```
//Exemplo com break
var nome = "";
switch(opcao) {
        default:
        case 1:
                nome = "Nome 1"
        break;
        case 2:
                nome = "Nome 2"
        break;
        case 3:
        case 4:
                nome = "Nome 3 ou 4"
        break;
}

//Exemplo com return
var nome = "";
switch(opcao) {
        default:
        case 1:
                nome = "Nome 1"
        return;
        case 2:
                nome = "Nome 2"
        return;
        case 3:
        case 4:
```

```
            nome = “Nome 3 ou 4”
        return;
}
```

Funções

As funções são estruturas de códigos independentes que podem ser chamadas em outros lugares do código.

As funções geralmente tem nome, tipo de retorno e seus argumentos. Exemplo:

```
// Exemplo de função em javascript
function somar(valor1, valor2) {
   return valor1 + valor2;
}

// Exemplo de função em Typescript
somar(valor1: number, valor2: number): number {
   return valor1 + valor2;
}
```

Quando chamamos uma função, passamos os **parâmetros**, caso tenha

// Chamando a função “somar”, que retornará 13, neste caso.

somar(5, 8)

Funções assíncronas

Uma função comum é executada em sequência - ou seja - como se fosse uma pilha: 1 primeira é executada, depois a segunda, depois a terceira, e assim vai.

Uma função assíncrona acontece fora desse tempo. Ela tem seu próprio tempo de execução. Então, se chamarmos uma função assíncrona, e continuar com nosso código, ela será chamada, passaremos ao restante do código, e ela ainda vai estar executando.

Para ter controle sobre isso, nós precisamos **aguardar** que função finalize. Isso é muito útil quando precisamos, por exemplo, usar o retorno de uma função.

No exemplo abaixo, nós estamos consultando dados de uma API, e na sequência, utilizando o retorno da API em um laço. Veja:

//Seu código…

*//Usamos **await** para aguardar o retorno da função.*

```
const dados = await service.getDados();

//Utilizamos os dados após preenchidos.
dados.forEach(item => {  console.log(item) })
```

Se não fizermos desse jeito, será gerado um erro, pois “dados” não estará pronto para ser utilizado.

No Javascript, uma outra forma de fazer isso, é através do then. Toda função assíncrona retorna um tipo de dado chamado Promise.

Usando o THEN, nós passamos uma função de callback como parâmetro. Veja como ficaria o exemplo acima:

```
service.getDados().then(dados => {
    dados.forEach(item => console.log(item));
} )
```

Assim como **await**, o THEN também aguarda a função, e só executa a função de callback (que está no trecho dados => {}) após a chamada assíncrona terminar.

Introdução à POO

A **Programação orientada a objetos** foi um dos grandes passos que a tecnologia deu, e possibilitou que os sistemas tomassem uma escala muito maior.

A **programação orientada a objetos** é a definição para transformar o que escrevemos de código em dados como se fossem objetos, facilitando o entendimento e a organização dos códigos.

Sendo assim, a orientação a objetos é baseada em modelos, que chamamos de **classes.**

Explicando de forma leiga: Uma **classe** é um modelo que define regras de comportamento e características.

As características chamamos de **propriedades.**

Os comportamentos chamamos de **métodos.**

Para usar um exemplo da vida real, vamos criar uma empresa e os colaboradores dentro dela. Veja:

Classe Programador

- Propriedades: *Nome, altura, idade, lista de linguagens de programação, formação.*

- **Métodos:** Andar, programar, tomar café.

Agora, vamos transformar isso em uma classe na linguagem Typescript:

```
class Programador {
    nome: string;
    altura: number;
    idade: number;
    linguagens: string[];
    formacao: string;

    andar(): void {
        console.log("Programador está andando");
    }

    programar(): void {
        console.log("Programador está programando");
    }

    tomarCafe(): void {
        console.log("Programador está tomando café");
    }
}
```

E como utilizamos uma classe? Através das **instâncias.**

Uma classe por si só não tem utilidade. Como falamos antes, ela é apenas um modelo. E se não for utilizada, isto é, **instanciada**, ela não terá utilidade.

Nós criamos uma instância de uma classe e então temos um **objeto**.

Veja:

```
var Pedro = new Programador();
Pedro.nome = “Pedro”;
Pedro.idade = 27;
Pedro.altura = 178;
Pedro.linguagens = [“C#”, “Cobol”, “Javascript” ];
```

No exemplo acima, nós criamos uma **instância** da classe Programador, e colocamos em uma variável chamada “Pedro”.

E então, definimos algumas das **propriedades**.

Agora sim, temos um objeto, que pode ser manipulado e utilizado pelo sistema.

Herança

Herança em orientação a objetos é um recurso muito útil. Imagina o mesmo exemplo do programador acima.

Se eu precisasse definir programador Front-End e programador Back-End, de modo que eu consiga manipular essas classes de forma separada, e reconhecê-las em meu sistema.

Eu poderia criar duas novas classes. Veja como ficaria em C#:

```
class ProgramadorFrontEnd extends Programador {
        linguagensDeMarcacao: string[];
}

class ProgramadorBackEnd extends Programador {
        bancosDeDados: string[];
}
```

Apenas para exemplificar: a classe ProgramadorFrontEnd tem uma propriedade “LinguagensDeMarcacao” e a classe ProgramadorBackEnd tem uma propriedade “BancosDeDados”.

Essas classes **herdam** de Programador (em c#, a herança é repreentada pelos dois-pontos logo após o nome da classe). E

essas classes possuem todas as propriedades e métodos que a classe “Programador” possui.

Sendo assim, eu não preciso criar as duas propriedades específicas que criei nessas classes, na classe “Programador”. E eu consigo ganhar algum nível de performance e organização de código.

Exemplos

```
var joao = new ProgramadorFrontEnd();
Joao.nome = “João”;
Joao.idade = 22;
Joao.altura = 168;
Joao.linguagens= [“Javascript” ];
Joao.linguagensDeMarcacao = [“XML”, “HTML5” ];

var laura = new ProgramadorBackEnd();
Laura.nome = “Laura”;
Laura.idade = 23;
Laura.altura = 170;
Laura.linguagens= [“Java”, “C#”, “PHP” ];
Laura.bancosDeDados = [“Sql Server”, “MySql”, “Oracle DB” ];
```

Estrutura da web

Nesta sessão, vamos entender como a internet funciona.

Entender a estrutura de qualquer assunto é o princípio básico para dominar os procedimentos envolvidos.

Quando você vai começar a construir uma casa, por exemplo, você começa pela estrutura.

No mundo digital é a mesma coisa.

A partir do momento que você começar a entender a estrutura de como funciona, você será capaz de pensar de forma objetiva para desenvolver as soluções que precisa.

Considere isso vale para front-end, back-end e qualquer outra tecnologia para desenvolvimento de software.

Como os dados trafegam na internet

A estrutura de comunicação na internet toda, basicamente se dá em 3 passos – os mesmos passos da comunicação humana: Emissor, canal e receptor. Isto é: Alguém pede uma informação.

Essa solicitação passa por um canal, e chega até o receptor. E esse ciclo pode se repetir ou fazer o caminho inverso.

Na internet, acontece de modo semelhante:

1. Web request: Quem solicita a informação chama-se **CLIENTE**.

Esse processo é chamado de Web Request (traduzindo para o português: Requisição Web).

2. Envio da mensagem: Essa solicitação passa por um **canal**, que é informação digital trafegando pelo seu provedor de internet, por exemplo.

3. Web response: A solicitação chega até o receptor, que neste caso chama-se **SERVIDOR.**

O servidor lê, interpreta, verifica se tudo está correto, e tenta encontrar o que foi solicitado.

E então, devolve a informação para o canal que volta ao **CLIENTE.** Este processo é chamado de Web Response (resposta web).

É isso que acontece quando alguém clica em um link na internet.

Percebeu quem é o cliente nesse caso? Sim, o navegador.

Assim que o navegador recebe a informação do servidor (ali em cima no passo 3), esta é verificada, tratada, manipulada se preciso for, “maquiada”, e tudo mais o que for necessário para exibir da forma desejada, em **HTML.**

Deploy

Chamamos de *deploy* o processo de publicar desenvolvimento e/ou alterações de sites ou sistemas em um servidor de hospedagem.

Geralmente, os serviços de hospedagem tem um manual ou um descritivo explicando o processo de publicação, bem como meios utilizados (como dados FTP).

Basicamente, após finalizar seu desenvolvimento, você fará a publicação, que é feita em 2 etapas: a **publicação** e o **deploy.**

Costuma-se chamar de **publicação**, o processo de transformar o que foi desenvolvido em um pacote pronto para o *deploy*. Não significa necessariamente que o *deploy* será feito na sequência.

O *deploy* acontece quando decidimos pegar todo esse conteúdo e colocar no servidor, tornando-o pronto para uso.

A publicação pode ser diferente para cada *framework* ou tecnologia. Por exemplo, uma publicação de um projeto Angular é feito através do comando: *ng build*

Uma publicação de um projeto *.Net* pode ser feito diretamente pelo *Visual Studio*, ao clicar com o botão direito, e utilizar a opção *"Publicar…".*

E projetos como PHP ou HTML / CSS e Javascript "puros" (que são interpretados), basta colocar o conteúdo diretamente no servidor (pulamos a etapa da "publicação" e fazemos o *deploy* de forma direta).

Após estar pronto para subir seus arquivos (seja uma pasta com os arquivos compilados ou o projeto na sua máquina), é necessário saber a forma de deploy e dados de acessos.

O modo mais comum e antigo é via FTP. Onde, basicamente, é feita uma cópia de arquivos de um lugar para outro.

Isso é feito através de:

- *Endereço do servidor*
- *Login*
- *Senha*

Basta configurar tudo isso em um *client* de FTP, e pronto, você estará com acesso às pastas do servidor. Geralmente, a pasta que é acessada assim que o site é solicitado tem nomes comuns como "*wwwroot*", "*public*" ou "*www*".

Essas informações são descritas pelo serviço de hospedagem.

Naturalmente, os servidores possuem permissões e regras de escrita e leitura em determinadas pastas. Então, é comum que algumas configurações ou validações de dados desse tipo sejam verificados para que tudo funcione corretamente.

Além do FTP, é bastante comum utilizarmos processos automáticos de *deploy*, como as **integrações contínuas,** que são integrações feitas de um **repositório de versionamento** (como o GIT), diretamente para o ambiente em questão.

Isso é feito através de configurações e regras bem definidas.

Repositórios de Versionamento (GIT)

Versionamento é uma funcionalidade muito importante para garantir desenvolvimento fácil, organizado, simples e com fácil acesso a alterações passadas.

Todas as alterações feitas em desenvolvimento, que forem enviadas para um servidor de versionamento, tem um número próprio de versão, e ainda permite que você inclua sua própria mensagem de observação sobre o que foi desenvolvido / alterado em cada versão

Basicamente, esses servidores funcionam da seguinte forma:

- No servidor, existe a **versão original** do código.

- Essa versão original pode ser dividida em várias **ramificações** (branches)

- Essas **ramificações** podem ter desenvolvimentos específicos

- Então, para você fazer seu desenvolvimento, você baixa uma cópia da versão original na sua máquina. Chamamos esse processo de **Clone.**

- Você faz o que precisa fazer na sua própria cópia, e depois sobe de volta as alterações para o servidor com a versão original. Chamamos esse processo de **push.** Pronto, o servidor original foi atualizado com uma nova versão do código.

Introdução ao HTML

HTML é uma sigla para Hyper-Text Markup Language (Linguagem de marcação de hipertexto) – e é o “esqueleto” da internet.

O HTML é um conjunto de elementos escritos que informa ao navegador o que deve ou não ser exibido.

O HTML não tem programação, lógica ou cálculos. É apenas uma marcação. Uma marcação que instrui o navegador o que deve ser feito.

É como um mapa do tesouro, onde a linha tracejada é um caminho, um círculo é um a árvore, e o “X” é onde está a recompensa.

É mais ou menos dessa forma que os navegadores entendem o HTML.

Sem entrar em detalhes, o navegador entende duas grandes estruturas no HTML: A *head* e o *body*.

A *head* é onde fica todo o conteúdo lógico de estrutura do documento HTML.

O *body,* como diz o nome*,* é o corpo do documento. É a parte que o usuário tem acesso e interage.

Head

A *head* é onde ficam as informações que são processados no início do carregamento e quem utiliza o navegador não tem acesso por meios convencionais a este conteúdo.

Na *head* podemos utilizar a tag *script,* para criar âncoras (*links*) para scripts como o *javascript*.

Também podemos utilizar *link* para ancorar estilos CSS externos.

Na *head* podemos também declarar *scripts* e estilos com conteúdo direto (e não apenas linkando documentos externos).

É aí também que definimos elementos utilizados em SEO (*Seach Engine Optimization*), como título, autor, descrição e palavra-chave, muito utilizados no mundo digital, bem como dados para redes sociais (conhecidos como Open Graph*).

**Open Graph é um padrão de nomenclatura para exibição de conteúdo social, como foto, link, url exibida, informações de autor, e outras informações*

Body

O *body* é o corpo do documento: É a parte que o usuário manipula e visualiza de forma direta.

Um formulário que será preenchido, um botão pressionado, textos, animações, tudo isso fica dentro do *body*.

Document Object Model - DOM

O HTML completo recebe o nome de DOM (Document Object Model – Modelo de objeto do documento), é isso que nos permite manipular através de tecnologias como o Javascript e o CSS.

O DOM é um modelo de dados que transforma todas essas marcações em “objetos manipuláveis”.

Cada objeto do DOM pode ser acessado através de seu nome, bem como seus atributos.

O html define atributos nos objetos do DOM, por exemplo:

- Atributos dos links (tag: ‘a’)

href – Acrônimo para *Hyperlink Reference,* define a âncora

title – Define um título que será exibido quando o mouse estiver sobre o link

- Atributos das imagens (tag: ‘img’)

src – Acrônimo para *Source,* define a origem da imagem.

title – Texto exibido caso a imagem não seja encontrada

- Atributos gerais (utilizados em qualquer elementos

id – *Identity* e indica um identificador único do elementos

class – Define uma lista de classes CSS. Cada classe é passada utilizando seu nome e separadas por espaço.

onclick – Evento que dispara uma função em Javascript ao ser clicado

style – Atributo utilizado para passar estilo *inline*

Esses foram alguns exemplos de atributos de atributos dos objetos do DOM.

Para destacar os elementos que são mais comuns no DOM, temos:

- Seções (section)

- Estruturas de navegação (nav)

- Formulários (form)

- Parágrafos (p)

- Containers de divisão (div)

- Rodapé (footer)

- Âncoras de hyperlink (a)

- *Cabeçalho (header)* – não confundir com a *head* da raiz do *HTML.*

Como o navegador interpreta o HTML

O navegador interpreta todas as marcações escritas, na ordem de leitura *top-down-left-right.*

Ou seja, da esquerda para a direita, e de cima para baixo.

Em quase todos os *scripts* utilizados isso faz diferença.

No CSS, por exemplo, que depende da ordem de execução das regras, o impacto é considerável.

De igual modo no Javascript, o script é lido e interpretado pelo navegador na mesma ordem.

Porém, dependendo da estratégia de utilização e dos frameworks, não necessariamente as funcionalidades são executadas em ordem. Você vai entender as peculiaridades adiante.

O que é uma API?

API é o acrônimo para *Application Programming Interface.*

Na web, API é conhecida como uma forma de acessar informações estruturadas via HTTP, tal como é o acesso a um site a partir de um navegador.

Uma API permite acessar dados diversos de um sistema, de modo que possamos manipular os dados, de acordo com as **permissões** definidas.

Elas também são utilizadas em contextos diversos para descentralizar responsabilidades de um sistema, de modo que seja possível construir várias aplicações que acessem a uma mesma API, para suas próprias necessidades.

É o caso de aplicações SPA (*Single Page Application*), desenvolvidas em *Angular* ou *React*, ou até mesmo aplicativos desenvolvidos em plataformas e/ou linguagens como Kotlin, React Native, Flutter, entre outras.

Uma API geralmente é acessada através de partes da URL chamadas de r**otas** ou **endpoints.**

Um **endpoint** é acessado através de um **método** de acesso HTTP (GET, POST, DELETE, PUT).

E podem ou não receber parâmetros. No caso do **GET,** os parâmetros são permitidos na URL. No exemplo abaixo, “2” é um parâmetro do **endpoint “**alunos”.

http://dominio/alunos/2

No caso do **POST**, podemos utilizar o **BODY** da requisição para passar dados mais complexos. Continue lendo para um exemplo prático da construção de uma API.

Autenticação x Autorização

Uma API pode permitir acesso anônimo (ou seja, não é preciso nenhum tipo de autenticação para isso), ou exige alguma **autenticação.**

A **autenticação** de uma API consiste em fornecer dados (como usuário, e-mail, login e senha) de modo que o sistema reconheça este usuário e libere o acesso.

Além da autenticação, uma API também pode ter níveis de **autorização,** que podem definir:

- Quais informações você tem acesso

- Como você pode acessar as informações

- Que tarefas você pode fazer na API (como inclusão, exclusão, edição de dados).

Tudo depende das regras definidas pelo sistema.

Formato JSON

Atualmente, o formato de dados mais comum utilizado no retorno de consultas em API´s é o JSON.

JSON é um padrão de comunicação baseada em objetos do **Javascript** que possibilita fácil manipulação de informação.

Há algum tempo atrás, as API´s e serviços externos tinham como padrão de comunicação o **XML**.

Ler um **XML** em uma API era um trabalho de difícil leitura, que demandava a construção de recursos específicos, conversão de dados, e uma série de questões que tornava o trabalho complexo e muitas vezes demorado.

A construção de uma API que retorna o formato JSON é relativamente simples nos dias atuais.

Praticamente todas as tecnologias e ferramentas já possuem suporte para este formato. Basta que na configuração da API (os *headers* do HTTP) seja definido o formato entregue, e a entrega do servidor seja de acordo.

Veja abaixo exemplos de um **endpoint** chamado **"alunos"** sendo consultado por Javascript:

```
async function consultar() {
        var url = 'https://dominio/alunos';
        var options = {
        method: 'POST',
        body: JSON.stringfy({
            alunoId: 2,
            status: 'ativo'
        })
    };
    var response = await fetch(url, options);
    if (response.ok) {
        var jsonData = await response.json():
```

```
            //Neste exemplo, nosso request retorna um objeto JSON
com os      dados id, nome e idade.
            const {id, nome, idade} = jsonData;
        }
}
```

//Exemplo chamada GET em Javascript:

```
async function consultar() {
            var url = 'https://dominio/alunos';
            var options = {
            method: 'GET'
        };
        var response = await fetch(url, options);
        if (response.ok) {
            var jsonData = await response.json():
            const {id, nome, idade} = jsonData;
        }
}


//Exemplo de retorno:
{
        id: 2,
        nome: 'Pedro',
        idade: 23
}
```

Glossário de tecnologias

Em sua grande maioria, projetos front-end utilizam as seguintes tecnologias como base para qualquer desenvolvimento:

HTML (HyperText Markup Language)

Linguagem de marcação usada para criar a estrutura e o conteúdo da página web.

CSS (Cascading Style Sheets)

Linguagem de folhas de estilo usada para estilizar a página web, definindo cores, fontes, layouts e animações.

JavaScript

Linguagem de programação usada para adicionar interatividade e dinamicidade às páginas web, permitindo a criação de funcionalidades avançadas, como animações, validação de formulários e manipulação do DOM (Document Object Model).

Frameworks

São conjuntos de bibliotecas, ferramentas e padrões de projeto que facilitam o desenvolvimento de aplicações web. Alguns dos mais populares são React, Angular, Vue.js e Bootstrap.

Pré-processadores CSS

São ferramentas que permitem escrever CSS de forma mais eficiente e organizada, adicionando recursos como variáveis, mixins e nesting. Alguns exemplos são Sass, Less e Stylus.

Task runners e bundlers

São ferramentas que automatizam tarefas de desenvolvimento, como minificação de arquivos, compilação de pré-processadores, criação de bundles e execução de testes. Exemplos incluem Grunt, Gulp e Webpack.

Gerenciadores de pacotes

São ferramentas que permitem instalar, atualizar e gerenciar bibliotecas e dependências em um projeto. Os mais populares são o npm (Node Package Manager) e o Yarn.

Versionamento

Ferramentas de versionamento são utilizadas para guardar versões de código e facilitar o desenvolvimento e manutenção. Uma das ferramentas mais conhecidas utilizadas mundialmente, é o Git, com seus respectivos serviços (Github, Azure Devops, Gitlab, Bitbucket e outros).

Continue estudando com o Combo

Front-End

O aprendizado não para aqui. Quantas vezes você já se deparou com situações complicadas no dia a dia, que não sabia a quem recorrer ou como resolver?

Se você já é programador, deve saber que existem diversos desafios no dia a dia, que você não encontra facilmente na internet.

São situações que vem com anos de experiência e prática. Por isso, nós, da Kyrios Cursos Online, reunimos em média 15 anos de experiência em desenvolvimento de software, back e front-end, em um conjunto de 10 livros, concentrando mais de 1000 páginas de conteúdo, além dos bônus.

E nós estamos disponibilizando somente para quem tem este e-book em mãos, um desconto especial de 40% na compra do nosso Combo Front-End.

Lembrando que essa oferta é limitada, e corre o sério risco de você abrir agora o link para acessar, e já ter esgotado!

Então, se você quer continuar estudando, dar uma guinada em sua carreira e aproveitar nosso desconto especial, aqui está o link: **https://combofrontend.com.br/oferta-especial/**

O que é o Combo Front-End?

É um kit com 10 e-books distribuídos em 1000 páginas de conteúdo sobre desenvolvimento front-end. Nesse kit de livros, você vai encontrará os seguintes e-books:

- Manual do Javascript
- React Básico
- Manual do SASS
- CSS: Layout Responsivo
- CSS: Cores e Geometria
- Manual do Grid-Layout
- Manual do Flexbox
- Dicionário CSS
- Manual do Typescript
- Aplicações web com React e Node JS
- Manual do Programador

Além disso, você também terá acesso a alguns bônus, voltados para soft skills, como marketing digital, desenvolvimento pessoal e aprendizado acelerado.

Formas de pagamento e garantia

A compra é totalmente segura, feita através da plataforma Hotmart, empresa líder no mercado de produtos digitais. Você pode efetuar o seu pagamento através de:

- Cartão de crédito parcelado ou à vista
- Boleto
- PIX
- E outras opções podem estar disponíveis na página de pagamento.

Agora é com você!

Se você deseja continuar adquirindo conhecimento nessa jornada e se transformar em um desenvolvedor respeitado, que não tem medo de problemas e que recebe o valor monetário de acordo com seus resultados, este é o link:

https://combofrontend.com.br/oferta-especial/

Te espero do outro lado!

Sobre

Esse e-book foi desenvolvido pela Kyrios Cursos Online, e escrito por Dorian Torres que é especialista em desenvolvimento de sistemas, desenvolvedor, consultor de negócios digitais e tecnologia, e entusiasta do comportamento humano e aprendizado.

Temos o objetivo de levar o conhecimento da programação de forma simples e fácil. Defendemos muito que a leitura é um dos meios mais importantes para se tornar um bom programador. Afinal, um programador ESCREVE códigos!

Seja bem vindo ou bem vinda!

Disclaimer

Todo o conteúdo encontrado neste livro está protegido sob as leis do direito autoral. Você não pode reproduzir esse material sem autorização expressa.

As entidades da marca Kyrios Cursos Online, bem como o Combo Front-End não garantem nenhum tipo de resultado financeiro ou garantia de sucesso. A utilização do conteúdo e aplicação dos conhecimentos obtidos nos projetos em que você vier a trabalhar é de sua total responsabilidade.