Fundamentos

Bootcamp Desenvolvedor Mobile Apps

Marcelo Sampaio Brigolini Silva

2020

Bootcamp Desenvolvedor Mobile Apps

Marcelo Sampaio Brigolini Silva

Sumário

Capítulo 1. Informações Gerais 4

Importância do módulo e do nivelamento de informação 4

Capítulo 2. Conceitos e Tipos de Aplicação Mobile 5

O que é a Plataforma Android 5

Tipos de Aplicações Mobile mais conhecidas 7

Capítulo 3. Rest e JSon 9

Boas práticas no design de APIs Rest 15

Capítulo 4. Introdução ao JavaScript, Java e Dart 17

Java 17

JavaScript 21

Dart 24

Marcas registradas 29

Referências 30

Capítulo 1. Informações Gerais

Seja bem-vindo ao Módulo I do Bootcamp Desenvolvedor Mobile. Nosso objetivo com este módulo é que você aprenda conceitos básicos importantes para o desenvolvimento de uma app mobile. Ainda não será criada nenhuma aplicação neste módulo, mas pedimos ao aluno que preste bastante atenção aos conceitos dele. Eles serão a base para todo o curso.

Esta apostila está organizada de forma a permitir ao aluno um guia de consulta rápida durante e após o Bootcamp. Entendemos que a melhor forma de aproveitar todo o conteúdo é ter bastante atenção às aulas gravadas e interativas e utilizar a apostila para consulta na hora de realizar o Trabalho Prático e Desafio.

Importância do módulo e do nivelamento de informação

O desenvolvimento de apps mobile para Android™, independentemente da linguagem ou framework utilizados, depende de conceitos comuns como a utilização de APIs REST, conhecimento do modelo Json.

Além disso, conheceremos as estruturas básicas de cada uma das linguagens utilizadas durante o curso.

Capítulo 2. Conceitos e Tipos de Aplicação Mobile

Um app mobile é um programa feito especificamente para ser executado em um dispositivo móvel. Os dispositivos móveis que mais conhecemos são celulares e tablets. No entanto, não podemos nos esquecer de outros dispositivos tais como os wareables (vestíveis) como relógios. Além disso, temos aplicações menos comuns móveis. São vários os carros hoje que contam com Media Players para Android. Também temos outros dispositivos como sistemas de entretenimento de aeronaves e sistemas de Home Entertainment como, por exemplo, dispositivos para streamming com Android.

O que é a Plataforma Android

Apesar do que diz o senso comum, o Android não é um sistema operacional, na verdade a documentação apresenta o Android como uma “pilha de software em Linux™ de código aberto”.

Dizer que o Android é uma pilha de software significa que ele é construído em várias abstrações interdependentes, cada uma responsável por uma parte do trabalho de entregar ao usuário final o serviço necessário.

A primeira das camadas da pilha é o Sistema Operacional Linux. Ele é responsável pela iteração direta com o Hardware como acesso à memória, acesso à câmera. Outra função importante do Kernel é o gerenciamento de segurança no acesso ao hardware. Na verdade, essa foi uma das grandes decisões da equipe que criou o Android. A utilização do Linux trouxe ao sistema operacional a estabilidade de um SO com longa história de mercado.

Mas acessar diretamente o Kernel do Linux não seria uma ideia inteligente, pois tornaria o desenvolvimento extremamente complexo. Além disso, como o Android é uma plataforma criada com foco nos dispositivos móveis, o acesso eficiente aos dispositivos é muito importante.

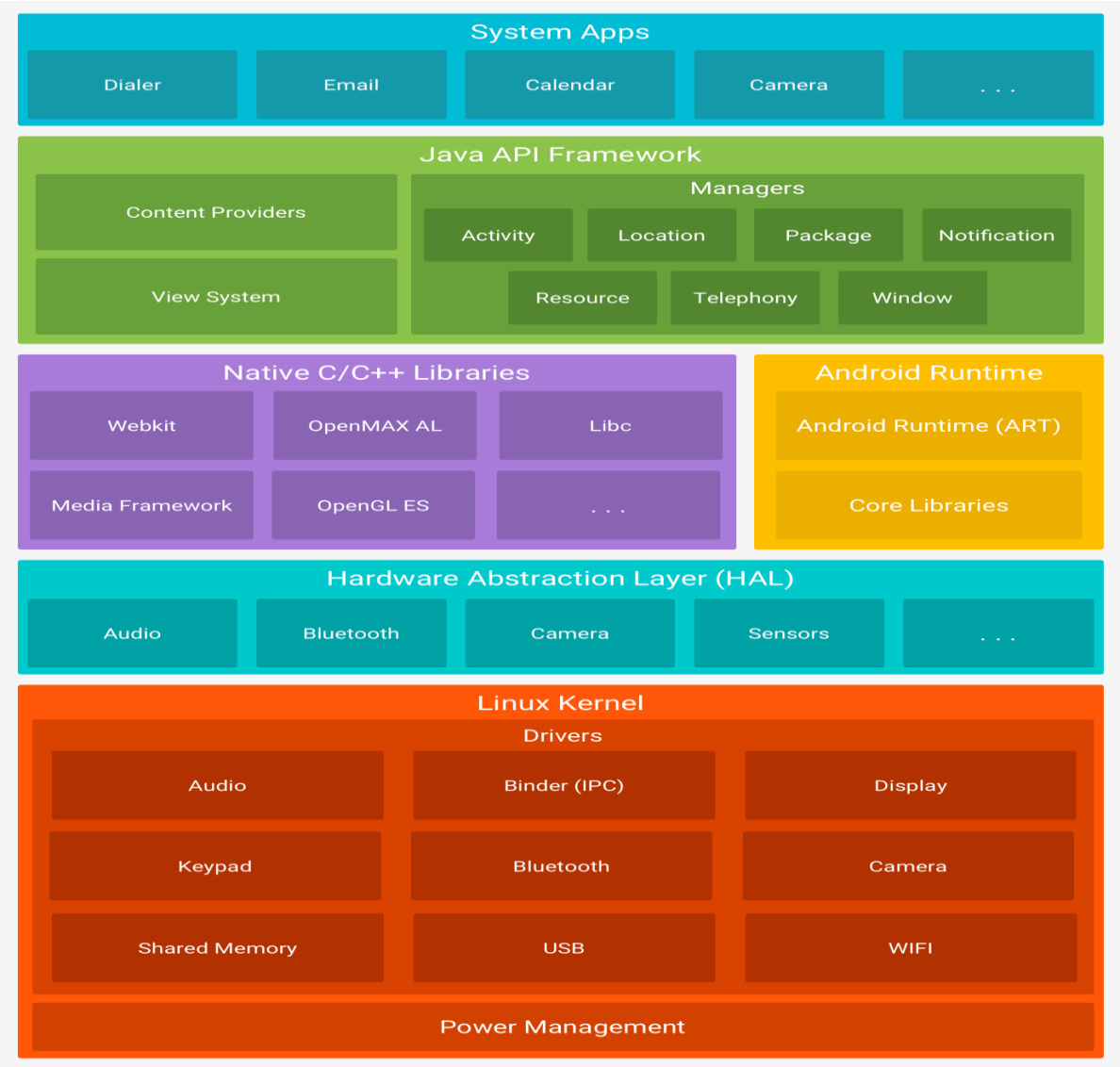
Dessa forma, foi criada uma segunda camada de abstração conhecida com HAL (camada de abstração ao hardware) para uma API de alto nível. Ela é responsável por saber quais capacidades o hardware provê e entregá-las às APIS Java. Assim, se um telefone tem uma câmera a HAL, é responsável por informar às APIS Java quais são as suas características como zoom ótico, amplitude de captação, velocidades de abertura etc.

A próxima parte da pilha é o Android Runtime (ART). Cada aplicação roda dentro do seu próprio Runtime. Isso garante que uma aplicação não consegue acessar informações de outra diretamente, utilizando o conceito de caixa de areia (SandBox). O Android Runtime também é responsável por recursos como Garbage Collector e JIT (Just in Time Compilation) e AOT (Ahead of Time Compilation).

O ART e o HAL fazem uso constante de bibliotecas C e C++. Apesar de não serem uma camada, essas bibliotecas são fundamentais para o que a plataforma tenha eficiência necessária.

A última camada são as APIS Java. São nelas que o programador age diretamente. No seu dia a dia o trabalho estará diretamente ligado a chamar APIs dessa camada. Não é possível para as aplicações acessar diretamente o ART e o HAL e muito menos o Linux e o Hardware. Isso garante, além do uso eficiente do hardware, a segurança de aplicações. Abaixo veja a foto com a stack da plataforma:

Figura 1 - A pilha da plataforma Android.



Fonte: Documentação Android on-line.

Tipos de Aplicações Mobile mais conhecidas

Existem várias diferentes categorizações para tipos de aplicação mobile. Cada uma leva em consideração alguns aspectos, tais como tipo de framework utilizado. Preferimos aqui ser um pouco mais práticos e vamos apresentar alguns

exemplos de frameworks de desenvolvimento para Android e explicar suas características:

SDK Android + Java ou Kotlin: É o desenvolvimento **100% nativo** para a plataforma Android. Gera apenas código que roda em dispositivos com Android. Como pontos positivos, têm a velocidade que a aplicação roda, além de receber as novidades da plataforma em primeira mão. De modo geral, faz mais sentido utilizar esse desenvolvimento para criação de jogos ou softwares, que precisam de uso intensivo ou muito específico do hardware. Normalmente os programadores são mais especializados e mais caros. Outro ponto é a consistência com a Interface Gráfica. Este ponto é cada vez menos relevante, pois as aplicações híbridas estão muito consistentes entre plataformas.

Xamarin/Flutter/React Native: Esses frameworks se encaixam no que conhecemos como **plataformas híbridas** ou **cross-plataform**. Eles geram código nativo para ambos Android e IOS. O código é relativamente rápido, pois o código é nativo, mas como há a necessidade de uma camada de abstração para cada plataforma, seu desempenho não é igual ao código nativo. É a escolha correta para a maior parte das aplicações criadas, pois boa parte dos softwares do mercado não tem necessidade de performance maior do que eles apresentam. Você conhecerá um pouco mais sobre Flutter e React Native neste Bootcamp.

Apache Cordova: O Apache Cordova se encaixa no que chamamos de Web Mobile Application. O que ocorre nesse tipo de aplicação é que ele não gera código nativo. Apenas é criada uma “casca” nativa que chama código HTML e CSS. Esse tipo de abordagem está cada vez caindo mais em desuso com a entrada de frameworks como o React Native. Quando você estudar um pouco mais de React Native perceberá que os conceitos de CSS conseguem ser utilizados em boa parte nesse framework. As aplicações Web Mobile são mais lentas que as aplicações nativas, mas tem como única vantagem a utilização de um mesmo ambiente de desenvolvimento web.

Capítulo 3. Rest e JSon

O termo REST foi criado por Roy Fielding na sua tese de doutorado. Ele define uma arquitetura para troca de informações entre cliente e servidor.

Todas aplicações desse modelo contêm um back-end, responsável por acessar dados, realizar cálculos e outras operações comuns a todas as plataformas e enviá-las para os clientes (chamado de front-end).

Existe, hoje, a necessidade de que uma mesma aplicação seja visualizada em vários tipos de dispositivos e mídias diferentes. O home banking é um ótimo exemplo disso. Ele pode ser acessado por um programa instalado no seu computador, pela web (através da página do banco), de um celular, tablet etc.

Provavelmente, todas essas mídias têm necessidades de negócio parecidas, como buscar um saldo, extrato, fazer transferências etc. Não faria sentido que cada uma implementasse sua forma de acesso a dados. Além disso, uma estratégia única de acesso ajuda na segurança, pois diminui o que conhecemos como “superfície de contato”, ou seja, as formas de iteração com os dados do banco.

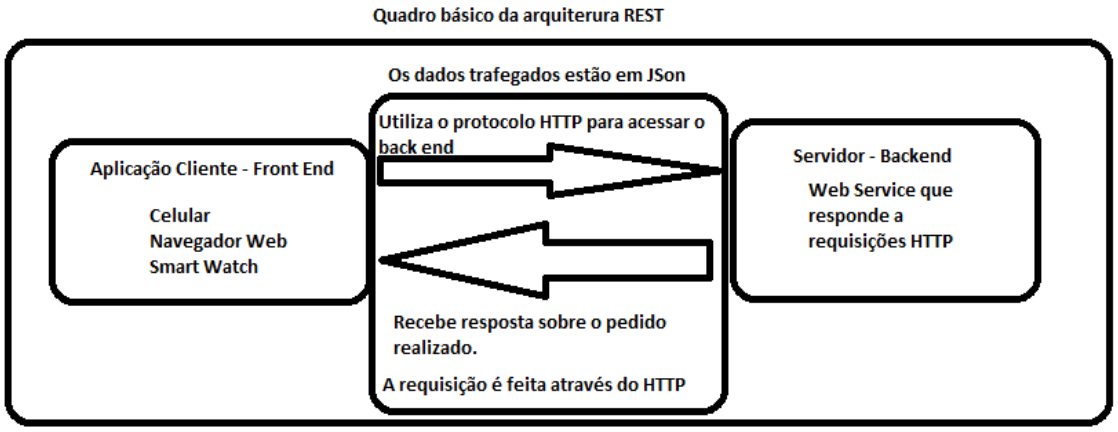
Dessa forma, todas as aplicações modernas estão baseadas no modelo em que um back-end pode ser acessado por diversos front-ends diferentes, em dispositivos e mídias diferentes.

Para que isso seja possível três coisas são necessárias:

- Um protocolo de comunicação padrão (HTTP).
- Uma arquitetura que permitisse padronizar a forma de apresentação do dado (REST).
- Uma forma padrão de representar o dado (JSon).

Apresentamos abaixo uma figura ilustrativa dessa arquitetura. Cada um dos itens será explicado melhor a seguir:

Figura 2 - Arquitetura REST.



A arquitetura Rest define que os dados são trafegados através do protocolo HTTP. Isso gera várias vantagens como veremos adiante.

O protocolo HTTP é Stateless. Isso significa que cada chamada ao servidor não guarda o estado da chamada anterior. Isso tem implicações diretas, especialmente na autenticação. Cada vez que o seu celular, por exemplo, precisa de pedir o saldo, a aplicação precisa enviar dados sobre a autenticação.

Uma requisição ao servidor sempre contém um verbo. Cada verbo tem uma aplicação específica. Os mais importantes verbos são:

- GET: Informa ao servidor que estamos requisitando que ele nos devolva informações quaisquer. Por exemplo, quando você requisita o seu saldo no home banking, uma requisição GET é feita, pedindo um recurso especial (o seu saldo).

- POST: Envia informações que devem incluir um recurso. Por exemplo, quando você inclui um cliente no sistema, o front-end envia uma requisição com o verbo POST para incluir o cliente no banco de dados.

- PUT: Envia informações que devem modificar um recurso. O mesmo cliente do exemplo anterior pode ser modificado por uma requisição com o verbo PUT.

- PATCH: Também é a requisição para modificar um recurso. Com a diferença de que o PUT deverá enviar todas as informações do cliente, a modificação no banco de dados do recurso deverá ser feita por completo (todos os campos) na requisição PUT, enquanto que o PATCH deverá modificar apenas um campo.

- DELETE: Exclui um recurso no servidor. Por exemplo a exclusão de um cliente.

Como o protocolo HTTP não nos obriga a realizar as operações conforme o REST, é muito comum ver APIs utilizando verbos errados para realizar as operações. Isso, apesar de não gerar grande diferença no desenvolvimento, pode ser desastroso na produção. Por exemplo, o verbo POST é conhecido por **não** ser idempotente enquanto os verbos DELETE, PUT e PATCH são idempotentes.

Mas o que isso gera? Bom, entre o cliente e o servidor, muitos servidores recebem e repassam a aplicação, cada um é conhecido por nó da rede. Se a comunicação entre dois desses servidores falhar, é realizada a tentativa de repetir a comunicação. Os verbos idempotentes podem ser repetidos quantas vezes for necessário sem gerar problemas. Imagine, se eu enviei a requisição de deletar um cliente da minha base de dados, ela pode ser repetida, pois não há risco de “dupla exclusão”. Já se eu pedir a inclusão de um cliente e a comunicação entre um dos nós precisar ser repetida por algum motivo, ela poderá gerar duplicidade de dados.

O protocolo HTTP nos garante que uma requisição POST não gerará repetição entre dois nós. No caso de falha de comunicação será retornado um erro. Já uma requisição DELETE poderá ser tentada novamente.

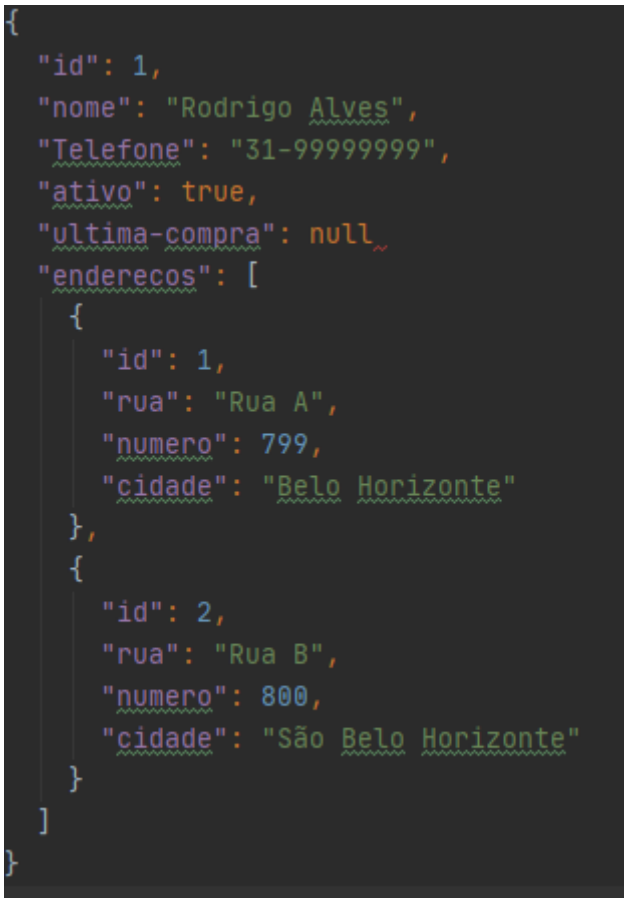
Atenção, tudo isso ocorre internamente. Do ponto de vista da sua aplicação apenas uma requisição está ocorrendo.

Já sabemos como a requisição ocorre. Mas como os dados são trafegados?

Aí entra o padrão JSON (JavaScript Object Notation). Apesar do nome, esta notação não serve apenas para programas feitos em JavaScript. Ela é utilizada de uma forma geral para tráfego de dados no REST.

A sintaxe de um JSON é a seguinte:

Figura 3 - Exemplo de JSON.



O que é importante saber sobre o formato JSON:

- Cada objeto JSON é delimitado por chaves. Um Array é delimitado por colchetes.
- Cada informação deve ser separada por vírgula.
- Os dados são sempre representados em pares nome:valor. Isso permite que as informações sejam acessadas diretamente pelo seu nome. Assim, se eu quiser saber o telefone do cliente posso usar algo do tipo “cliente.telefone”
- Um item pode ser dos tipos: string (sempre entre aspas), número, objeto (um outro objeto JSON), um array de objetos JSON, boolean ou nulo.

Apesar de bastante simples, O JSON é apenas isso. É importante mencionar que dados considerados binários como imagens e áudio também podem ser enviados dentro de objetos JSON através do formato BASE-64

Se você quiser se aprofundar sobre o JSON vá até a página https://www.json.org/json-en.html.

O JSON pode ser usado tanto na chamada da requisição como no retorno dela. Na inclusão de um cliente, por exemplo, o objeto JSON pode conter as informações do cliente. Se, por outro lado, precisamos de buscar uma lista de clientes, ela virá também no formato JSON.

Toda requisição HTTP retorna um código de status. Esse código é extremamente importante para a arquitetura REST e, infelizmente, por vezes esquecida pelos desenvolvedores. Os status HTTP são divididos da seguinte forma:

1. Informational Responses (100 – 199).
2. Sucessfull Responses (200 – 299).
3. Redirects (300 – 399).
4. Client Errors (400 – 499).
5. Server Errors (500 – 599).

Não vamos nos estender em explicar cada um dos status. No seu dia a dia, para cada tipo de operação, você deve pesquisar qual o status HTTP mais se assemelha com o que você quer dizer.

Alguns exemplos:

- 200 – OK – A requisição foi realizada com sucesso. Por exemplo, quando pedimos um cliente ao servidor e ele retorna corretamente.
- 201 – Created – Um objeto foi criado no servidor. Usado, por exemplo, nos verbos POST.

- 400 – Bad Request – Algo da sua requisição está errado, por exemplo, você enviou alguma informação no JSON não reconhecida pelo servidor ou faltou alguma informação.
- 403 – Forbidden – Você não tem acesso ao recurso.
- 404 – Not Found - Recurso não encontrado.

A última coisa que devemos saber sobre o REST é como acessar um determinado recurso. A fórmula é sempre a mesma:

PROTOCOLO://ENDEREÇO_DO_SERVIDOR/RECURSO?QUERYSTRING

Para explicar melhor vamos usar alguns exemplos de requisições, todas com o verbo GET.

Exemplo 1:

- **http://meubanco.com.br/cliente/1**
  - http é o protocolo, essa parte não muda muito. Por vezes utilizaremos a versão segura do protocolo que é o https.
  - meubanco.com.br é o servidor.
  - Cliente significa que queremos informações de clientes.
  - 1 especifica exatamente qual cliente queremos.

Exemplo 2:

- **http://blog.com.br/posts?descricaoParcial=”pandemia”**
  - http e servidor são iguais.
  - A interrogação significa que iremos passar uma query (conhecida como querystring). Nesse caso estamos dizendo que queremos descrições que contenham a palavra pandemia.

Exemplo 3:

- **http://blog.com.br/posts/1**
  - Busca o post de id 1.

Exemplo 4:

- **http://blog.com.br/posts/1/comments**
  - Busca os comentários do post de id 1.
  -

Boas práticas no design de APIs Rest

Segue uma lista de boas práticas, ela serve como um checklist para quando for criar sua API. Boa parte delas já foram descritas no capítulo:

1. Use o verbo correto para a ação que você vai tomar.
2. Rest é orientado a recursos. O endpoint (rota) que for criado deve levar em conta isso.
3. Um endpoint é a combinação de um verbo (GET, POST etc.) e uma URI onde ficam o recurso, exemplo: GET /*comments*/1 busca (GET) o comentário de id 1.
4. Use o status de resposta adequadamente. Consulte https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.
5. Sempre retorne Json. Lembre-se de retornar o header de content correto.
6. Não use verbos nas URIs. Lembre-se, URIs no Rest devem ter recursos (substantivos).
7. Use plural para os recursos.
8. Quando gerar um erro, retorne a mensagem de erro no body em formato Json.

9. Essa é mais uma recomendação de design limpo. Não aninhe recursos. Se você quer todos os livros de um autor ao invés de usar */authors/5/books,* use */books/?author=5.*

10. Filtros e paginações devem ser feitas na querystring. Exemplo: */authors/?page=3;namePartial="Marcelo".*

11. Os status 401 (Unauthorized) e 403 (Forbidden) são diferentes! 401 significa que ele não está autorizado (não se autenticou no sistema) 403 significa que ele se autenticou, mas não tem acesso ao recurso (Acesso negado).

Capítulo 4. Introdução ao JavaScript, Java e Dart

Durante nosso curso serão aprendidos três frameworks diferentes para desenvolvimento de apps para Android. São eles:

- Desenvolvimento utilizando SDK Android utilizando a linguagem **Java.**
- Desenvolvimento utilizando o framework React Native que utiliza **JavaScript.**
- Desenvolvimento utilizando o framework Flutter que utiliza a linguagem **Dart**.

Neste capítulo apresentaremos as principais estruturas de cada uma das linguagens. Muito longe de querermos apresentar todos os recursos das linguagens, mostraremos de forma simplificada poucos recursos que são os mais importantes neles.

Optamos por deixar os trechos de código de forma a ser possível fazer a cópia deles diretamente do PDF, por outro lado, pedimos ao aluno que tente fazer cada um dos códigos sem copiar. Assim o aprendizado será muito maior.

Para executar cada um dos exemplos o aluno poderá utilizar emuladores on-line para facilitar o trabalho. São eles:

- Java: https://www.jdoodle.com/online-java-compiler/ (Atenção, coloque apenas as classes nesse site. A definição de pacotes não funciona nele).
- JavaScript: https://playcode.io/.
- Dart: https://dartpad.dev/.

Java

```
package br.com.marcelo.sampaio.igti.exemplo;
public class Exemplo {
   public static void main(String[] args) {
```

```
        String hello = "Hello World";
        System.out.println(hello);
    }
}
```

Este é um código básico em Java. Os pontos que devemos ter atenção:

- Todo arquivo em Java começa com a descrição de qual pacote ele está. O Pacote é utilizado para facilitar a organização da aplicação. Um pacote define o que conhecemos como namespace da classe. Uma forma de pensar uma namespace é o como o nome completo da classe. Assim, a classe acima é conhecida como br.com.marcelo.sampaio.igti.exemplo.Exemplo.
- Dentro de cada arquivo Java existe uma classe pública, seu nome é o mesmo do arquivo.
- Nesse caso, a classe Exemplo tem um método main que é utilizado para iniciar a aplicação. Na verdade, quando você for desenvolver para Android, verá que é um pouco diferente essa classe com o método main.
- Vemos depois a forma padrão de criação de variáveis em Java. Criamos uma variável do tipo String.
- Por último imprimimos na saída padrão a variável criada.

```
package br.com.marcelo.sampaio.igti.exemplo;

class Cliente {
    private Integer id;
    private String nome;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
```

```
        this.id = id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
public class Exemplo {
    public static void main(String[] args) {
        Cliente cliente = new Cliente();
        System.out.println(cliente);
    }
}
```

O Código acima cria uma segunda classe dentro do mesmo arquivo. Repare que a segunda classe não é pública. Apenas uma classe pode ser pública em Java dentro de um arquivo. Criamos dois atributos id e nome com acesso privado. Para acessar esses atributos criamos o que chamamos de getters e setters. São métodos que permitem ler e modificar os valores de um atributo.

Repare que criamos, com o comando new, uma nova “Instância” de Cliente e associamos à variável “cliente”. Mas ao imprimir a classe veremos a seguinte informação:

```
br.com.marcelo.sampaio.igti.exemplo.Cliente@7c53a9eb
```

Isso ocorre, pois, cada classe em Java deve ter um método toString que “explica” como ela deve ser exibida como String. Como o método println espera uma String, precisamos dela. Colocamos o seguinte código dentro da classe Cliente:

```
@Override
public String toString() {
```

```
    return "Cliente{" +
            "id=" + id +
            ", nome='" + nome + '\'' +
            '}';
}
```

E receberemos a resposta:

Cliente{id=null, nome='null'}

Por fim, repare que os valores de id e nome estão “null”. Esse é o padrão ao criar algum objeto, se não damos valor a ele, será null.

Por fim vamos apresentar outro código com dois conceitos bastante importantes:

```
public class Exemplo {
    public static void main(String[] args) {
        ArrayList<Cliente> lista = new ArrayList<>();
        Cliente cliente = new Cliente();
        cliente.setId(1);
        cliente.setNome("Marcelo");
        lista.add(cliente);
        cliente = new Cliente();
        cliente.setId(2);
        cliente.setNome("Romário");
        lista.add(cliente);
        for (Cliente item : lista) {
            System.out.println(item);
        }
    }
}
```

Criamos uma lista de clientes. Para isso usamos um tipo chamado ArrayList. ArrayList é uma das várias Coleções possíveis em Java. Além dela temos Map, Set e várias outras. Cada uma com sua utilidade.

Repare que a declaração é ArrayList<Cliente>. Isso diz para o ArrayList que você quer que ele contenha elementos do tipo Cliente. Serve para garantirmos coerência do nosso código e evita muitos erros.

A sintaxe do for colocada também é apenas uma delas. Existem várias outras. Durante os vídeos mostraremos outras formas de iterar nas coleções.

Além disso, outras sintaxes mais modernas da linguagem existem, algumas serão apresentadas durante o módulo de Desenvolvimento Nativo, já outras não podem ser utilizadas com a plataforma Android.

JavaScript

Criado inicialmente para funcionar apenas no Browser, o JavaScript hoje tomou uma proporção gigante. Praticamente tudo pode ser feito em JavaScript, desde aplicações back-end até aplicações front-end e, inclusive, Inteligência Artificial!

Todos os criadores de soluções para JavaScript devem seguir uma padronização feita pela Ecma International, chamada EcmaScript. Todo ano temos uma nova versão de EcmaScript com novas funcionalidades, o que torna a linguagem cheia de funcionalidades. Então vamos ao código.

```
const cliente = {
   "id":1,
   "nome":"Marcelo"
}

console.log(cliente);
```

O primeiro ponto interessante sobre javascript, ele não é tipado! Isso significa que você não dirá que cliente é do tipo Cliente. Simplesmente define o objeto e liga à variável conforme feito acima. Isso pode ser muito bom ou muito ruim! Como não há tipagem em JavaScript fica muito mais fácil cometer erros. Existem algumas soluções que estão ganhando corpo no mercado como o Typescript para solucionar isso.

Outro ponto importante é que javascript utiliza a notação JSON para tudo. Não seria difícil de perceber isso: JavaScript Object Notation já explica muito.

```
let listaCliente = [cliente, cliente, cliente]
```

A linha acima permite a criação de um array de clientes. Para manipular um array (incluir ou excluir elementos) temos várias formas. No entanto a boa prática em JavaScript é tentar sempre usar o conceito de imutabilidade.

Um objeto é imutável quando ele não pode se modificar. A única coisa que podemos fazer com um objeto imutável é criá-lo novamente. Ser imutável torna o objeto menos propenso a erros como modificações de algum atributo de forma incorreta. Além disso, facilita o acesso concorrente ao objeto. Tudo em JavaScript deve, preferencialmente, ser imutável.

Por exemplo, para incluir um novo cliente ao array acima usamos:

```
listaCliente = [...listaCliente,clienteNovo]
```

Os três pontos são chamados em JS como “Spread Operator”. A forma que devemos ler a linha acima é: Crie um vetor e coloque nele todos os clientes de listaCliente e mais o clienteNovo.

Outro ponto importante de se observar: na declaração de listaCliente usamos “let” enquanto que em cliente usamos “const”. Uma variável const não pode ser modificada, enquanto que uma variável declarada com let pode ser modificada. Mas não estamos falando de imutabilidade? Como trocamos o valor de listaCliente. Observe! Não trocamos o valor de listaCliente. Apenas apontamos para um novo array que contém todos os valores anteriores mais no novo cliente. Assim, o que

acontece na prática é: criamos um array com todos os itens do array anterior, incluímos um novo item, apontamos listaCliente para esse novo array e o anterior é apagado.

Caso queiramos excluir um item do array podemos utilizar o método filter. Por exemplo, se quisermos excluir o cliente com id = 1 fazemos o seguinte:

```
listaCliente = listaCliente.filter((cliente) => !cliente.id == 1 )
```

Temos vários conceitos importantes aqui. O método filter devolve um novo array que é colocado em listaCliente. Esse método recebe como parâmetro uma função, o que essa função retornar é retornado pelo filter. É muito comum o envio de funções no JavaScript como parâmetro de outras funções ou métodos.

Vamos detalhar mais a função que filter recebe:

```
(cliente) => cliente.id !== 1
```

Essa notação é chamada de arrow function. Uma arrow function é apenas um "açúcar sintático", ou seja, uma forma mais elegante de escrever uma função.

Seria a mesma coisa de dizer:

```
function filtro(cliente) {

return cliente.id !== 1

}
```

Mas uma das belezas do JavaScript é não ser verboso (muito código para expressar pouca coisa). Repare mais uma coisa, enquanto na função normal existe um return na arrow function apenas temos uma expressão. Essa é outra característica do JavaScript, se uma função serve apenas para retornar um valor podemos simplesmente colocar esse valor ou a expressão que gera ele sem o return.

Concluindo, a arrow function acima faz o seguinte: retorno verdadeiro para todos os clientes em que o id for diferente (!==) de 1. Como a função filter retorna

todos os itens que retoma true na função, serão retornados todos os clientes menos o que tenha id = 1.

E se quisermos iterar em um array?

```
for (let item of listaCliente) {
   console.info(item);
}
```

Esse é um exemplo simples de laço for que envia para a tela (console) cada item existente no array.

Mas e se eu quiser, por exemplo, gerar um outro array apenas com os nomes dos clientes? Para isso o Javascript tem outra funcionalidade chamada Map. Map é outro método de array que mapeia um array em outro. Veja abaixo:

```
listaCliente = listaCliente.map((cliente) => cliente.nome )
```

O método map devolverá um novo array com o resultado da função. O retorno da função acima para o array que criamos é:

[“Marcelo”,“Marcelo”,“Marcelo”]

Dart

Dart é uma linguagem criada pelo Google que tem seu foco em desenvolvimento de User Interfaces. Um código em Dart pode rodar basicamente de duas formas: Nativa ou Web.

A forma nativa do Dart gera código para processadores ARM e Intel (X_86 e X_64). Os processadores ARM são bastante utilizados nos aparelhos móveis de hoje em dia, e isso foi pensado pelo Google para criar um Framework que permitisse gerar código Nativo para aparelhos com Android ou IOS embarcados. Esse framework é o Flutter. Conheceremos mais sobre o Flutter nos próximos módulos.

Diferente do JavaScript, Dart é uma linguagem fortemente tipada. Vamos aos conceitos mais importantes da linguagem:

```
void main() {
   var msg = "Marcelo";
   print("Hello ${msg}");
}
```

Assim como o Java, um código Dart inicia-se dentro do main. Mas aí já começam as diferenças. Repare que ao declarar a variável msg, usamos a palavra reservada var. Ela serve para fazer o que chamamos de inferência de tipos. O que é isso? Se você olhar para a atribuição da variável, verá que estamos atribuindo a uma string. O Dart já reconhece isso automaticamente e diz que msg é uma string. Atenção, isso não significa não ter tipo! Se na linha abaixo eu tentar atribuir msg a um objeto, não será possível. Na segunda linha temos um exemplo de interpolação. Interpolar uma string significa permitir que variáveis e expressões possam ser colocadas dentro da string, formando uma só informação. Podemos dizer que:

“Hello ${msg}”

É o mesmo que:

“Hello “+${msg}

Vamos agora fazer um acesso à API utilizando Dart:

```
import 'dart:html';
void main() {
   HttpRequest.getString('https://jsonplaceholder.typicode.com/todos/1')
      .then(print);
}
```

A primeira novidade aqui é o import no início. Como no Java e no JavaScript, ele serve para carregar um determinado pacote para nosso programa. Nesse caso, ele está carregando um pacote especial diretamente do Dart. Os pacotes da library

dart são exclusivos da linguagem. Repare que falamos em library e não namespace. O conceito de namespace não existe no Dart. Já o conceito de library é mais simples. Library é um conjunto de classes que estão agrupadas.

Após isso usamos a classe HTTPRequest para fazer uma requisição com o verbo GET. Há outras várias formas de fazer isso e, inclusive, pacotes de terceiros bem mais complexos e completos. Mas vamos ficar com o simples, por enquanto.

Importante: No Flutter normalmente se usa outra biblioteca para acessos à APIs rest. Optamos por mostrar essa forma por ser possível de se executar no DartPad. De qualquer forma os conceitos são os mesmos e a sintaxe muito parecida.

Repare que a chamada é assíncrona. Lembre-se, quando você chama uma API ela pode demorar um milissegundo para responder um 1 segundo, ou nem responder.

Portanto, ao responder, queremos fazer algum tratamento. Dessa forma fazemos a requisição e **então (then)** imprimimos o retorno.

```
import 'dart:html';
void main() async {
  print (await
HttpRequest.getString('https://jsonplaceholder.typicode.com/todos/2'));
}
```

O Código acima traz apenas uma novidade (um açúcar sintático). Ele permite imprimir diretamente a saída do getString. Para isso usamos o await.

Entenda da seguinte forma: Estamos mandando imprimir algo. Mas não temos o que vamos imprimir imediatamente pois precisamos chamar uma API (tempo de resposta variável). Então dizemos para o método print “esperar por” (await) uma resposta da API. O async serve para avisar ao dart que aquele método deverá rodar em modo assíncrono. Basicamente o async e o await são um par.

Recomendamos o uso dessa estratégia quando o aninhamento de “then” for muito grande. Por exemplo: a chamada de uma API gera um resultado que chama uma segunda e assim sucessivamente.

```
import 'dart:html';
void main() async {
   var data = {'title': 'Marcelo', 'body': 'Professor Marcelo Explica DART',
'userId': '999'};
   HttpRequest.postFormData('https://jsonplaceholder.typicode.com/posts',
data).then((HttpRequest resp) {
      print(resp.status);
      print (resp.responseUrl);
      print (resp.responseText);
   });
}
```

O código acima serve apenas para mostrar que a resposta (variável resp) é um objeto com várias informações sobre a requisição como status, url de resposta e responseText devolve o que está no body em formato texto.

```
import 'dart:html';

import 'dart:convert';

void main() async {
   var result = await
HttpRequest.getString('https://jsonplaceholder.typicode.com/todos');
   var toJSon = jsonDecode(result);
   print(toJSon);
}
```

Neste exemplo, mostramos como transformar um resultado em um objeto Json. Utilizaremos esse objeto nos próximos exemplos:

```
import 'dart:html';
import 'dart:convert';

void main() async {
   var result = await
HttpRequest.getString('https://jsonplaceholder.typicode.com/todos');
   var resultJSON = jsonDecode(result);
   for (int i=0;i<resultJSON.length;i++){
      print("Posição ${i} ${resultJSON[i]}");
   }
}
```

Neste exemplo, iteramos no resultado do array, mostrando cada um dos itens do Json em separado.

Marcas registradas

- IOS é uma marca registrada de Apple.
- Java é uma marca registrada de Oracle.
- Android, Dart e Flutter é uma Marca Registrada de Google LLC.
- Linux é uma marca registrada de Linux Foundation.

Para saber como utilizar as marcas registradas entre em:

- https://www.linuxfoundation.org/trademark-usage/.
- https://developer.android.com/distribute/marketing-tools/brand-guidelines.
- https://www.apple.com/legal/intellectual-property/guidelinesfor3rdparties.html.
- https://flutter.dev/brand.
- https://dart.dev/terms.
- https://www.oracle.com/legal/trademarks.html.

Referências

ANDROID. *Documentation for app developers*. Disponível em: <https://developer.android.com/docs>. Acesso em: 01 jul. 2020.

ANDROID. *Platform Architecture*. Disponível em: <https://developer.android.com/guide/platform>. Acesso em: 01 jul. 2020.

ECMASCRIPT. In: *WIKIPÉDIA, a enciclopédia livre*. Flórida: Wikimedia Foundation, 2020. Disponível em: <https://en.wikipedia.org/wiki/ECMAScript>. Acesso em: 01 jul. 2020.

JSON. *Introducing JSON*. Disponível em:<https://www.json.org/json-en.html>. Acesso em: 01 jul. 2020.

REST. In: *WIKIPÉDIA, a enciclopédia livre*. Flórida: Wikimedia Foundation, 2020. Disponível em: <https://en.wikipedia.org/wiki/Representational_state_transfer#References>. Acesso em: 01 jul. 2020.