TEMA 20

Aplicações Móveis Web

Objetivos:

- Conceitos sobre aplicações móveis
- Conceitos sobre HTML para ambientes móveis
- Interacção com câmara e GPS
- Comunicação com serviços externos

20.1 Introdução

Os ambientes móveis, nomeadamente os focados em *smartphones* e *tablets*, possuem métodos próprios para o desenvolvimento de aplicações, sobre a forma de um Software Development Kits (SDKs). Esses ambientes possibilitam aceder às capacidades do dispositivo, nomeadamente às Application Programming Interfaces (APIs) para acesso à câmara fotográfica, som, localização, acelerómetros ou outros sensores. As aplicações desenvolvidas podem igualmente trocar informação com servidores, permitindo a implementação de aplicações ricas de troca de mensagens, troca de imagens, ou jogos, entre muitas outras.

Uma desvantagem desta abordagem é que as três principais plataformas actualmente em utilização (*Windows Phone*, *Android*, *iOS*) apresentam APIs completamente diferentes. Além disso, todas estas plataformas exigem que as aplicações sejam desenvolvidas em linguagens diferentes. A plataforma *Android* utiliza *Java*, a plataforma *Windows Phone* utiliza *C#* e a plataforma *iOS* utiliza *ObjectiveC* ou *SWIFT*. Desenvolver uma aplicação para as três plataformas obriga a triplicar parte do trabalho, o que levanta vários problemas para o planeamento e manutenção do código produzido, assim como para o planeamento de um modelo de interacção consistente.

Felizmente, os navegadores *Web* presentes nas várias plataformas possuem capacidades bastante avançadas de processamento. Assim, em vez de três aplicações nativas, podemos

muitas vezes criar uma aplicação *Web*, que corre igualmente em qualquer plataforma. A grande vantagem é que a aplicação pode ser desenvolvida uma única vez, usando tecnologias standard como *JavaScript* e HTML5. Ainda para mais, bibliotecas como o *Twitter Bootstrap* foram desenhadas para permitirem uma visualização correta das páginas *Web* tanto em dispositivos móveis como em computadores com ecrãs maiores. A página do serviço *Facebook*, que se ajusta automaticamente ao dispositivo cliente, é um bom exemplo da capacidade de escalabilidade das páginas atuais.

É claro que as aplicações web também têm algumas desvantagens: a velocidade de execução pode ser inferior, o aspeto da aplicação poderá não ser exatamente igual ao das aplicações nativas, a execução depende da existência de uma ligação de dados e pode não ser possível aceder a todos os recursos disponíveis nativamente.

Para o desenvolvimento deste tipo aplicações é comum utilizarem-se sistemas como o *PhoneJS* ou *PhoneGap*. No entanto, a utilização destes sistemas requer conhecimentos mais avançados. Por isso, este guião foca-se na utilização das funcionalidades através de uma variante da biblioteca *Twitter Bootstrap* denominada de *Ratchet* que pode ser obtida em `http://goratchet.com/`. A documentação desta biblioteca pode ser obtida em `http://goratchet.com/components/`. Deve consultar esta documentação antes da execução deste guião.

20.2 Uma aplicação simples

As aplicações não são mais do que páginas *Web* em que a lógica é implementada em *JavaScript* e um conjunto de páginas de estilos criam a ilusão de se tratar de uma aplicação real. Antes de iniciar este ponto, aceda à documentação da biblioteca *Ratchet* e verifique que componentes estão disponíveis.

O desenvolvimento inicia-se obtendo a biblioteca *Ratchet* e colocando os ficheiros necessários nos directórios corretos (directório **`dist`**). Tem também de existir um ficheiro **`index.html`** que representará a aplicação.

O exemplo seguinte resulta numa aplicação apenas com uma barra de título com o texto **`LABI`**.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>LABI</title>
<meta name="viewport"
 content="initial-scale=1, maximum-scale=1, user-scalable=no, minimal-ui">
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
<link rel="stylesheet" href="css/ratchet.css">
```

```
<link rel="stylesheet" href="css/ratchet-theme-ios.css">
<link rel="stylesheet" href="css/app.css">
<script src="js/jquery-2.2.3.min.js"></script>
<script src="js/ratchet.js"></script>
</head>
<body>
<header class="bar bar-nav">
<h1 class="title">LABI</h1>
</header>
<div class="bar bar-footer">
</div>
<div class="content">
<ul class="table-view">
<li class="table-view-cell">Hello World</li>
</ul>
</div>
</body>
</html>
```

Exercício 20.1

Crie um directório **`hello/`** e coloque nele um ficheiro **`index.html`** com o conteúdo acima. Obtenha a biblioteca *Ratchet* e copie o conteúdo do directório **`dist/`** para dentro do directório **`hello/`**. Verifique o resultado com o seu browser.

Para verificar a aplicação num telemóvel, pode enviar a pasta completa para o servidor **`xcoa.av.it.pt`** e depois aceder à página *Web* a partir do telemóvel. Ou então verifique a versão dos professores em `http://xcoa.av.it.pt/labi/labi2016/ratchet/hello/`.

Em alternativa ao **`xcoa`**, se estiver numa rede local sem firewalls, poderá activar um servidor HyperText Transfer Protocol (HTTP)[1] no seu computador para servir o conteúdo do diretório atual e poder aceder noutro dispositivo. Pode fazer isto executando **`python -m SimpleHTTPServer`**. (Na rede da UA, devido às firewalls instaladas, isto não resultará.)

Exercício 20.2

Modifique a referência do tema para **`ratchet-theme-android.css`** e verifique que a aplicação muda para um aspeto semelhante ao sistema android.

Exercício 20.3

Adicione um componente adicional à aplicação desenvolvida.

20.3 Localização

Frequentemente os dispositivos móveis possuem um equipamento de Global Positioning System (GPS) que permite determinar a localização com bastante exactidão. No entanto, é possível determinar a localização de outras formas, por exemplo através das redes sem fios disponíveis. Os diversos meios de determinação de localização foram assim englobados numa mesma API que permite obter a localização independente do método utilizado. Para salvaguardar a privacidade, o acesso da aplicação à localização requer uma autorização explícita pelo utilizador.

O código JavaScript abaixo demonstra como obter a localização.

```
function showPosition(position){
    $("#location").html(position.coords.latitude+"  "+position.coords.longitude);
}

$(document).ready(function() {
    navigator.geolocation.getCurrentPosition(showPosition);
});
```

O método **navigator.geolocation.getCurrentPosition** pede acesso à localização e regista a função **showPosition** como *callback* para ser chamada quando o resultado estiver disponível.

Na aplicação cliente será necessário incluir este código *JavaScript* e um elemento HyperText Markup Language (HTML)[2] com o identificador certo para apresentar o resultado.

```
<div class="content">
    <div id="location"></div>
</div>
```

Recomenda-se a utilização da biblioteca *jQuery* que tem de ser incluída no directório **js** e importada para a página (no **<head>**).

Exercício 20.4

Altere a sua aplicação de forma a mostrar a localização do utilizador. Pode testar localmente ou enviar a página para o servidor **xcoa.av.it.pt**.

Exercício 20.5

Utilizando *LeafletJS* adicione um mapa centrado na posição atual do dispositivo.

20.4 Comunicação com serviços externos

A comunicação com serviços externos é importante pois permite que uma aplicação possa trocar informação com serviços, ou mesmo com outros utilizadores. Aplicações de *chat*, de visualização do estado do tempo, ou de partilha de outra informação (ex, imagens), podem ser implementadas rapidamente. Acima de tudo, este tipo de aplicações é útil para demonstrar que as aplicações *Web*, mesmo em ambientes móveis, podem ser dinâmicas e interagir com outros serviços externos.

Um exemplo simples é o de obter a data e hora de um servidor remoto. Para isto serão necessários os seguintes componentes:

- Um botão para iniciar o processo (pode ser substituído por um *timer* caso se pretenda actualizar a informação de forma periódica.)
- Código *JavaScript* que obtenha a informação do servidor remoto.
- Uma aplicação que implemente o serviço pedido.
- Um elemento HTML para armazenar o resultado.

Os dois elementos HTML são adicionados no elemento de classe **`content`**.

```
...
<div class="content">
    <button id="refresh" class="btn btn-block btn-primary">Refresh</button>
    <div id="clock" style="width:100%; text-align:center;"></div>
</div>
```

Enquanto que o código *JavaScript* é adicionado num ficheiro que tipicamente se denomina **`app.js`**. Neste caso, o código espera que seja enviado um elemento JavaScript Object Notation (JSON)[3] com dois atributos: **`date`** e **`time`**. O pedido é activado quando o elemento com identificador **`refresh`** receber um evento de **`click`**.

```
function refresh(){
    $.get("/time",function(response){
        var text="<h2>"+response.date+"</h2><br /><h2>"+response.time+"</h2>";
        $("#clock").html(text);
```

```
    });
}

$( document ).ready(function() {
    $("#refresh").on("click",refresh);
});
```

Do lado do serviço, é necessário implementar um método que devolva a data e hora, como o seguinte.

```
import cherrypy
import time

class HelloWorld(object):
    @cherrypy.expose
    def index(self):
        # Método serve_file tb poderia ser utilizado
        f = open("index.html")
        data = f.read()
        f.close()
        return data

    @cherrypy.expose
    def time(self):
        cherrypy.response.headers["Content-Type"] = "application/json"
        return time.strftime('{"date":"%d-%m-%Y", "time":"%H:%M:%S"}').encode('utf-8')

cherrypy.server.socket_port = 8080
cherrypy.server.socket_host = "0.0.0.0"
cherrypy.tree.mount(HelloWorld(),"/","app.config")
cherrypy.engine.start()
cherrypy.engine.block()
```

Note que alguns ficheiros são estáticos, enquanto outros são gerados dinamicamente. Os ficheiros estáticos encontram-se nos directórios **`css`** e **`js`**. Desta forma, é necessário criar um ficheiro chamado **`app.config`** com o seguinte conteúdo:

```
[/]
tools.staticdir.root ="/home/utilizador/directorio-do-servico"

[/css]
tools.staticdir.on: True
tools.staticdir.dir: "css"

[/js]
tools.staticdir.on: True
tools.staticdir.dir: "js"
```

Exercício 20.6

Construa um exemplo que demonstre a comunicação entre uma aplicação Web e um serviço, através de JSON.

Exercício 20.7

O exemplo anterior pode ser modificado de forma a devolver algo mais útil, tal como a distância para o estádio do SL Benfica (38.752667, -9.184711).

Usando a função seguinte componha um exemplo que envie a localização actual do cliente para o servidor, que responderá devolvendo a distância para o estádio.

```
from math import radians, cos, sin, asin, sqrt
...
def distance(lat, lon):
    lat1 = 38.752667
    lon1 = -9.184711

    lon, lat, lon1, lat1 = map(radians, [lon, lat, lon1, lat1])  # Graus -> rads

    dlon = lon - lon1
    dlat = lat - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat) * sin(dlon/2)**2  # Haversine
    c = 2 * asin(sqrt(a))

    km = 6367 * c  # 6367=raio da terra

    cherrypy.response.headers["Content-Type"] = "application/json"
    return json.dumps({"distance": km})
...
```

20.5 Acesso a imagens

Através de APIs específicas é igualmente possível aceder às imagens armazenadas num dispositivo móvel ou mesmo activar a câmara e obter uma imagem. Estas podem depois ser processadas localmente ou enviadas para servidores. Este método recorre a um elemento **<input>**, especificando que se pretende aceder a fotografias.

```
...
<div class="content">
    <input type="file" accept="image/*"></input>
</div>
...
```

O resultado será que o dispositivo irá pedir ao utilizador que selecione o método de entrada, podendo este ser a câmara ou os documentos já existentes. Se se especificar o atributo **capture="camera"**, a câmara será activada imediatamente para que possa capturar uma imagem.

Exercício 20.8

Integre o exemplo anterior numa aplicação e verifique o que acontece quando activa o input criado.

Exercício 20.9

Adicione as classes **btn**, **btn-primary**, **btn-block** e **button-input**. Considere depois o seguinte excerto de Cascading Style Sheets (CSS)[4].

```
.button{
  text-align: center;
  color: #007aff;
  width: 137px;
}
.button:before{
  color: white;
  content: 'Usar Imagem';
  padding-right:40px;
  margin-top: -20px;
  padding-left: 10px;
}
```

Componha a aplicação e teste.

Depois de estar selecionada a imagem, seria interessante poder visualizá-la. Isto pode ser feito se for activada uma função depois da fotografia ter sido seleccionada e existir um elemento onde a apresentar. O elemento neste caso será um **<canvas>**. Este elemento HTML é semelhante a um **<img>**, com a diferença que é possível desenhar para ele em tempo real, enquanto que um elemento **<img>** apresenta uma imagem estática. O novo HTML seria:

```
...
<input type="file" accept="image/*" onchange="updatePhoto(event);"></input>
<canvas id="photo" width="530" height="400">
...
```

O código *JavaScript* necessário corresponde à função **`updatePhoto()`** e irá aplicar a imagem capturada ao elemento **`<canvas>`**:

```
function updatePhoto(event){
    var reader = new FileReader();
    reader.onload = function(event){
        //Criar uma imagem
        var img = new Image();
        img.onload = function(){
            //Colocar a imagem no ecrã
            canvas = document.getElementById("photo");
            ctx = canvas.getContext("2d");
            ctx.drawImage(img,0,0,img.width,img.height,0,0,530, 400);
        }
        img.src = event.target.result;
    }

    //Obter o ficheiro
    reader.readAsDataURL(event.target.files[0]);
    sendFile(event.target.files[0]);
}
```

Exercício 20.10

Componha uma aplicação que replique o exemplo anterior.

Eventualmente a imagem pode ser enviada para um servidor remoto. Será necessário criar código *JavaScript* para construir um pedido de envio (**`POST`**):

```
function sendFile(file) {
    var data = new FormData();
    data.append("myFile", file);
    var xhr = new XMLHttpRequest();
    xhr.open("POST", "upload");
    xhr.upload.addEventListener("progress", updateProgress, false);
    xhr.send(data);
}
```

```
function updateProgress(evt){
    if(evt.loaded == evt.total)
        alert("OK!");
}

function updatePhoto(evt){
    ...

    sendFile(image[0]);

    //Libertar recursos da imagem seleccionada
    windowURL.revokeObjectURL(picURL);
}
```

Do lado do servidor será depois necessário receber o ficheiro. Os ficheiros enviados são fornecidos ao serviço e é necessário copiar a informação para um local mais permanente. Isto porque os dados serão apagados assim que o método **upload** terminar. O código de exemplo será o seguinte:

```
@cherrypy.expose
def upload(self, myFile):
    fo = open(os.getcwd()+ '/uploads/' + myFile.filename, 'wb')
    while True:
        data = myFile.file.read(8192)
        if not data:
            break
        fo.write(data)
    fo.close()
```

Exercício 20.11

Crie um conjunto de aplicações que permita o envio de imagens para o servidor.

Exercício 20.12

Altere o exemplo de forma a enviar outros ficheiros além de imagens.

Glossário

API Application Programming Interface

CSS Cascading Style Sheets

GPS	Global Positioning System
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
SDK	Software Development Kit

Referências

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616 (Draft Standard), Updated by RFCs 2817, 5785, 6266, Internet Engineering Task Force, jun. de 1999.

[2] W3C. (1999). HTML 4.01 Specification, URL: `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[3] E. T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 7159, Internet Engineering Task Force, mar. de 2014.

[4] W3C. (2001). Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, URL: `http://www.w3.org/TR/2011/REC-CSS2-20110607/`.