



Corso di Laurea in Informatica  
Tesi di Laurea

# Insight

Una piattaforma per l'erogazione di contenuti  
contestualizzati a luoghi e oggetti

**Relatore**  
Ch. Prof. Andrea Albarelli

**Laureando**  
Franco Scarpa  
Matricola 842833

**Anno Accademico**  
2017/2018

*Al relatore, per l'opportunità che mi ha dato  
di sviluppare nuove conoscenze*

*Ai miei genitori, per l'immancabile e costante aiuto*

*Alla Dott.ssa Doina Bizgan, la mia ragazza,  
per l'incessante sostegno*

*Ai miei compagni di corso, per la bellissima  
avventura vissuta insieme*

*“Lo studio è come la luce che illumina la tenebra dell’ignoranza, e la conoscenza che ne risulta è il supremo possesso, perché non potrà esserci tolto neanche dal più abile dei ladri. Lo studio è l’arma che elimina quel nemico che è l’ignoranza. È anche il miglior amico che ci guida attraverso tutti i nostri momenti difficili.”*

- Dalai Lama -

## Sommario

Il progetto del tirocinio, su cui si è basata la realizzazione di questa tesi, è consistito nella creazione di un'applicazione per dispositivi *mobile* che fosse compatibile con i due principali sistemi operativi *mobile* presenti sul mercato, ossia **iOS** e **Android**. Il compito è stato portato a termine optando per uno sviluppo ibrido del *software* sfruttando il *framework* **Ionic**, il quale utilizza **TypeScript** (un *super-set* di JavaScript) come suo linguaggio nativo. Al termine della stesura del codice sono state realizzate due versioni differenti dell'applicazione, installate, rispettivamente, su un dispositivo Android e su un *device* iOS.

# Indice

<b>1 Le applicazioni</b>	<b>5</b>
1.1 Cos'è un'applicazione . . . . .	5
1.2 Le applicazioni native . . . . .	6
1.3 Le applicazioni <i>web</i> . . . . .	7
1.4 Le applicazioni ibride . . . . .	8
<b>2 Il contesto di realizzazione</b>	<b>10</b>
2.1 La modalità di realizzazione . . . . .	10
2.2 I vantaggi e gli svantaggi . . . . .	11
2.3 Gli interessi applicativi . . . . .	11
2.4 Lo stato dell'arte . . . . .	14
2.5 Le sperimentazioni in contesti simili . . . . .	17
<b>3 Panoramica del progetto</b>	<b>19</b>
3.1 Premesse . . . . .	19
3.2 Il funzionamento dell'applicazione . . . . .	22
3.3 La gestione dell'applicazione . . . . .	25
3.4 L'ideazione del prototipo . . . . .	26
3.5 La preparazione del progetto . . . . .	32
3.5.1 La versione per Android . . . . .	33
3.5.2 La versione per iOS . . . . .	34
3.6 Il supporto dei browser . . . . .	35
<b>4 Una panoramica dei concetti e dei meccanismi fondamentali</b>	<b>36</b>
4.1 I componenti e le pagine . . . . .	37
4.2 Come funziona Angular . . . . .	38
4.3 La navigazione . . . . .	39
4.4 I <i>plugin</i> . . . . .	42
4.4.1 <i>Promise</i> e <i>Observable</i> . . . . .	43
4.4.2 Installazione e utilizzo . . . . .	44
4.5 Il formato dei dati nei codici QR . . . . .	45
4.6 I dati e i <i>provider</i> . . . . .	46

<b>5 Il codice</b>	<b>48</b>
5.1 La struttura della cartella di progetto . . . . .	48
5.2 La directory <i>./src/</i> . . . . .	50
5.3 La creazione e la gestione delle pagine . . . . .	51
5.4 La gestione della navigazione . . . . .	54
5.5 L'installazione e l'importazione dei <i>plugin</i> . . . . .	55
5.5.1 Lo <i>scanner</i> dei codici QR . . . . .	56
5.6 La pagina principale . . . . .	57
5.7 La pagina della lista dei luoghi . . . . .	62
5.8 La pagina che descrive un singolo luogo . . . . .	65
5.9 La pagina che elenca gli artefatti . . . . .	68
5.10 La pagina che descrive un artefatto . . . . .	70
5.11 Il <i>provider</i> . . . . .	73
<b>6 Conclusioni</b>	<b>77</b>

# Introduzione

L’idea alla base di questo progetto era quella di fornire un servizio di visita guidata ai luoghi dell’**Università Ca’ Foscari** e alle bellezze che sono custodite al loro interno. L’obiettivo che si è voluto perseguire consisteva nel rendere l’utente in grado di ricevere, direttamente sul suo *smartphone* e in maniera automatica, informazioni sul luogo in cui egli si trova e sui vari artefatti che è possibile ammirare. A tale scopo, si è richiesto l’utilizzo di due strumenti specifici: i **codici QR** e i **beacon BLE** (*Bluetooth Low Energy*). I primi sono elementi molto simili ai ben noti codici a barre, che permettono di “nascondere” informazioni, le quali possono essere portate alla luce eseguendo una scannerizzazione del codice stesso. I secondi, invece, sono elementi che fungono da antenne e che ne seguono il principio di funzionamento: quando un dispositivo entra nel raggio d’azione di un *beacon*, il *device*, mediante *software* appositamente programmato, reagisce. Quest’azione può essere finalizzata, ad esempio, per lanciare in maniera automatica un video o navigare a un URL specifico.



Figura 1: Un codice QR



Figura 2: Un *beacon* BLE

Se si parlasse in termini di programmazione orientata agli oggetti, le entità che hanno determinato il funzionamento dell’applicazione sono state due: il concetto di **luogo** e il concetto di **artefatto**. In parole poche, l’utente riceve informazioni che possono essere relative sia a un preciso luogo (posizionando un apposito *beacon* e un codice QR in prossimità dell’ingresso), sia a uno specifico artefatto (posizionando, in questo caso, soltanto un codice QR nelle

vicinanze dello stesso). Una volta eseguita la scannerizzazione di un codice, il *software* installato sul *device* sotto forma di applicazione permette di fornire informazioni aggiuntive e dettagliate sull'elemento in esame. I *beacon*, a differenza dei codici QR, sono presenti, come detto, solo in prossimità degli ingressi.

Per quanto concerne la struttura della tesi, essa è così articolata:

- nel **capitolo 1** viene presentato il concetto di applicazione, che cos'è, come può essere realizzata e quali sono i vantaggi e gli svantaggi di ognuno dei diversi percorsi di sviluppo che possono essere intrapresi;
- nel **capitolo 2** si discute riguardo al contesto in cui si posiziona l'applicazione che è stata concretamente progettata, gli interessi applicativi delle tecnologie utilizzate e quali strumenti siano stati usati per portare a termine la progettazione;
- nel **capitolo 3** si presenta una panoramica del progetto, affrontando temi importanti quali la gestione dell'applicazione, l'ideazione del prototipo e la preparazione della cartella che ha contenuto tutto il progetto;
- nel **capitolo 4** si spiegano tutti i meccanismi fondamentali che regolano il funzionamento dell'applicazione, come la gestione della navigazione attraverso l'interfaccia utente, l'utilizzo dei *plugin* e la gestione dei dati;
- nel **capitolo 5**, infine, viene presentato tutto il codice che è stato scritto per l'effettiva realizzazione del *software*, da quello delle singole pagine a quello del *provider* dei dati.

# Capitolo 1

## Le applicazioni

### Introduzione

In questo capitolo si affronta la spiegazione di che cosa sia un'applicazione e delle sue caratteristiche generali. Nel paragrafo 1.1 vengono date alcune nozioni generali sul concetto di applicazione. I tre paragrafi successivi presentano le principali modalità con cui è possibile realizzare un tale tipo di *software*: nel paragrafo 1.2 viene presentata la categoria delle applicazioni native, nel paragrafo 1.3 quella delle applicazioni *web* e nel paragrafo 1.4, infine, quello delle applicazioni ibride. In ognuno di questi tre paragrafi si delineano le caratteristiche, i vantaggi e gli svantaggi di ciascuna categoria.

### 1.1 Cos'è un'applicazione

Un'applicazione (termine derivante dall'espressione inglese *mobile application software*, o semplicemente *mobile app*, spesso abbreviata in *app*) è un *software* applicativo progettato per essere eseguito all'interno di dispositivi *mobile*, come *smartphone* o *tablet*, sebbene ormai il concetto venga esteso anche alle piattaforme *desktop*. Il processo che porta dall'ideazione all'effettiva realizzazione di un'applicazione può avvenire secondo metodologie differenti. Nello specifico, esistono tre grandi categorie a cui un *software* del genere può appartenere. Esso, infatti, può essere nativo, una *web app* oppure, ancora, un *software* ibrido. Ognuna di queste tre macro aree presenta caratteristiche peculiari nonché punti di forza e di debolezza, che emergono sulla base di molte questioni differenti, tra cui:

- la **modalità di sviluppo**;

- i **requisiti** che il *software* deve soddisfare;
- l'**esperienza**, l'**abilità** e le **conoscenze** del programmatore;
- l'**utenza** che fruirà dell'applicativo;
- la **modalità di diffusione**.

La scelta di quale percorso di sviluppo intraprendere spetta alla decisione condivisa del programmatore e dell'eventuale committente, che devono ponderare i pro e i contro che le diverse categorie mettono di fronte. Si procede, ora, col delineare le caratteristiche salienti di ognuna di queste macro aree.

## 1.2 Le applicazioni native

Un'applicazione nativa è tale qualora il *software* viene sviluppato con codice nativo, ossia il medesimo utilizzato dal sistema operativo di destinazione. Esse sono scaricabili (gratuitamente o meno) e installabili su un dispositivo *mobile*, come uno *smartphone* o un *tablet*, tramite gli appositi *store* presenti sulle diverse piattaforme. A titolo esemplificativo, si potrebbe sviluppare un'applicazione nativa per la piattaforma iOS scrivendo il suo codice sorgente in **Objective-C** o nel più moderno **Swift**, linguaggi scelti da Apple per le sue applicazioni, e rilasciarla sfruttando l'apposito *store* digitale che l'azienda di Cupertino mette a disposizione dei suoi utenti, l'**App Store**. Il medesimo discorso vale per il sistema operativo Android; in questo caso, però, il linguaggio da utilizzare per la piattaforma di casa Google sarebbe **Java**, mentre il *software* verrebbe divulgato tramite **Google Play**, il *market* ufficiale di Android. Questa tipologia di sviluppo garantisce la realizzazione di *software* integrato profondamente con il *device*, in grado di accedervi a tutte le risorse: fotocamera, *file system*, dati sulla batteria, localizzazione, etc. In figura 1.1 sono presentate le icone di alcune applicazioni che rientrano in questa categoria:



**Figura 1.1:** Alcuni esempi di applicazioni native

L'elemento principale che denota queste applicazioni è, non a caso, la potenza: essendo sviluppate con il linguaggio della piattaforma su cui vengono installate, riescono a sfruttare le piena capacità del dispositivo. Rappresentano, quindi, la scelta ottimale quando si tratta di dover sviluppare *software* che richiede notevoli calcoli computazionali o che, ad esempio, sfrutta pesantemente la potenza grafica del dispositivo. Elemento da non trascurare è che, essendo diffuse tramite gli appositi *store* digitali, è possibile monetizzarne dalla loro commercializzazione.

A questi lati positivi si accompagnano, però, degli aspetti più critici. Il primo fra tutti è la necessità di dover scrivere un codice diverso per ogni piattaforma, il che porta il programmatore ad essere necessariamente abile in più di un linguaggio di programmazione, oppure ad affidarsi a individui terzi per poter fornire *software* che sia eseguibile su *device* che eseguono sistemi operativi differenti. Oltre a questo, il processo di distribuzione dell'applicativo sugli *store* non è così immediato, in quanto le applicazioni devono essere sottoposte a dei controlli prima di poter essere rese disponibili al *download*, controlli che possono impiegare anche delle settimane per un semplice aggiornamento del *software*.

### 1.3 Le applicazioni *web*

Molto diverse rispetto alle precedenti, le applicazioni *web* sono veri e proprio siti *web*, altamente *responsive*, progettati in maniera tale da simulare l'aspetto delle interfacce native (da qui il nome di applicazioni *web based* o, semplicemente, *web app*). In quanto tali, esse non sono in grado di accedere alle piene funzionalità *hardware* e *software* del dispositivo su cui vengono eseguite e non vengono diffuse tramite gli *store*, ostacolando, pertanto, un'ipotetica monetizzazione dalla loro commercializzazione. Da non dimen-

ticare, inoltre, la necessaria connessione a Internet per coloro che ne fanno uso, anche se quella dell'utilizzo *offline* è una delle sfide del presente e del futuro. In figura 1.2 è presentata la schermata di un'applicazione che rientra in questa categoria:



**Figura 1.2:** Un esempio di applicazione *web*

Anche questa categoria gode, però, dei suoi vantaggi. Il principale riguarda la progettazione, che risulta essere decisamente più semplice rispetto a quella di un'applicazione nativa, dato che viene portata a termine mediante l'utilizzo dei linguaggi **HTML**, **CSS**, **JavaScript** e altri, sia *front-end* che *back-end*. In qualità di sito *web responsive*, inoltre, questo tipo di applicazioni non è sottoposto al processo di approvazione necessario per pubblicare *software* negli *store* digitali.

## 1.4 Le applicazioni ibride

La terza categoria, infine, funge da una sorta di “anello di giunzione” tra le altre due poc’anzi presentate, ed è quella delle applicazioni ibride. Esse combinano gli aspetti fondamentali delle due precedenti filosofie di sviluppo. Nella sostanza, un'applicazione ibrida si basa sul principio (preso in prestito dal mondo Java) del “*write once, compile anywhere*”. Viene infatti progettata partendo da un unico codice sorgente, che prevede l'utilizzo di linguaggi come **HTML**, **CSS** e **JavaScript** (come le *web app*), per dar vita ad applicazioni che vengono eseguite, però, all'interno di un **container** nativo che sfrutta una **WebView**, il tutto installato sotto forma di applicazione all'interno di un dispositivo. Tutto ciò permette all'applicazione di accedere alle funzionalità del *device*. In figura 1.3 sono presentate le icone di alcune applicazioni che rientrano in questa categoria:



**Figura 1.3:** Alcuni esempi di applicazioni ibride

L'evidente vantaggio che emerge consiste nella possibilità di progettare *software* con un'unica stesura del codice sorgente, in grado di sfruttare le risorse del dispositivo. Tali applicazioni, dal momento che sono, in tutto e per tutto, simili a quelle native, almeno da un punto di vista teorico, combinano i vantaggi della pubblicazione negli *store* digitali, e quindi della monetizzazione, con l'utilizzo di un “unico” linguaggio (sebbene ramificato), caratteristica tipica delle *web app*.

Lo svantaggio delle applicazioni ibride si snoda attorno all'interazione non perfettamente ottimale con il *device* su cui vengono eseguite, dato che essa non potrà mai essere comparata con quella di un *software* nativo: se si intende realizzare un'applicazione che richiede molta potenza di calcolo, è meglio optare per uno sviluppo nativo del codice. Inoltre, le funzionalità richieste vengono aggiunte, in fase di sviluppo, sfruttando *plugin* di terze parti, sviluppati da programmatore più o meno capaci, accompagnati da una documentazione che può essere ottimale, così come terribilmente scarsa.

## Capitolo 2

# Il contesto di realizzazione

### Introduzione

In questo capitolo si presenta una panoramica del contesto in cui l'applicazione realizzata si colloca. Nel paragrafo 2.1 si indicano la modalità e i linguaggi che sono stati utilizzati durante tutto il progetto. Nel paragrafo 2.2 si elencano tutti i vantaggi e gli svantaggi che comporta la scelta di realizzare *software* ibrido. Nel paragrafo 2.3 si evidenziano quali sono gli interessi applicativi che scaturiscono dall'utilizzo delle tecnologie scelte (ossia i codici QR e i *beacon* BLE). Nel paragrafo 2.4 viene presentato il vasto insieme degli strumenti e *framework* che permettono di realizzare applicazioni ibride.

### 2.1 La modalità di realizzazione

Il progetto del tirocinio, come detto, ha richiesto la creazione di un'applicazione *mobile* ibrida, sviluppata utilizzando i linguaggi che si incontrano durante il classico sviluppo *web*: **HTML**, **CSS** e **JavaScript** (o meglio, TypeScript). Il *software* così realizzato è stato successivamente inglobato all'interno di un contenitore nativo, dandogli così le “sembianze” di una comune applicazione, installabile sui *device*. Il requisito fondamentale era la possibilità di realizzare una versione per le due principali piattaforme *mobile* presenti sul mercato, iOS e Android. È stato così deciso perché questi sistemi operativi, proprietà rispettivamente di Apple e Google, dominano interamente il mercato dei sistemi operativi *mobile*.

## 2.2 I vantaggi e gli svantaggi

Si è già avuto modo, nell’introduzione al presente elaborato, di indicare quali sono i vantaggi e gli svantaggi nell’intraprendere tale percorso di sviluppo *software*. Il lato positivo predominante è la comodità di poter effettuare un’unica stesura del codice, il quale viene poi compilato in base alle varie piattaforme desiderate. Altro punto di forza, in qualità di applicazione, è rappresentato dagli *store* e dalla possibilità di monetizzazione in seguito alla commercializzazione del programma. Così come ci sono lati positivi, tuttavia, ci sono anche aspetti più critici, che abbiamo già avuto modo di vedere, in primis la ridotta potenza del *software*. Oltre a questo, le varie funzionalità rese disponibili all’interno dell’applicazione vengono implementate mediante l’utilizzo di strumenti di terze parti, i *plugin*. Si tratta di codice realizzato da altri sviluppatori e la cui documentazione, talvolta, non risulta essere ottimale o, ancor peggio, il codice può contenere *bug* o si dimostra essere poco ottimizzato. Ancora, questi strumenti possono offrire un livello di personalizzazione decisamente limitante. Avendo scelto uno sviluppo ibrido, non è stato possibile esimersi dall’affidarsi a questi componenti aggiuntivi. Si capisce come parte fondamentale della realizzazione del *software* non sia stata solo e puramente la stesura del codice “di base”, bensì anche l’attività di ricerca dei *plugin* migliori, ben documentati, che soddisfassero tutti i requisiti necessari e che, naturalmente, supportassero le piattaforme scelte. Tutto ciò non vuole gettare ombra sul fatto che esistono, comunque, ottimi *plugin* sviluppati dalle *community*, ma soltanto che la loro potenza e possibilità di personalizzazione non può, di fatto, essere equiparata ad una controparte nativa.

## 2.3 Gli interessi applicativi

Sebbene l’utilizzo dei codici QR possa essere ritenuto una funzionalità interessante, seppur, ormai, decisamente consolidata, la caratteristica distintiva dell’applicazione è, senza dubbio, l’invisibile integrazione con i *beacon* Bluetooth, che permettono di fornire un’esperienza d’uso coinvolgente e interattiva. Senza aver bisogno di alcuna azione manuale da parte dell’utente, il *software* equipaggiato nel dispositivo è in grado di rilevare, in maniera automatica, il luogo in cui egli si trova, fornendo dati e informazioni. Eseguendo una ricerca sul *web*, è facile imbattersi in articoli, *post*, interviste di specialisti e sviluppatori che pongono l’accento su come la tecnologia dei *beacon* abbia spiragli decisamente interessanti, finendo per sfociare in quel mare di possibilità che è rappresentato dall’**Internet of Things**.

**IoT** (Internet of Things) è un concetto che rappresenta l'estensione di Internet a un oggetto materiale qualsiasi rendendolo “intelligente”. In altre parole, un elemento di uso quotidiano può essere reso interattivo, in grado ad esempio di fornire dati e informazioni utili o di comportarsi in un determinato modo in reazione a un evento. Un banale esempio può essere quello di un semaforo che, a un incrocio fra due strade, diventa verde quando vede arrivare un veicolo, dopo essersi accertato che, nella strada trasversale, non stia transitando nessuno. Si riesce subito a capire come i campi applicativi a cui potrebbe essere estesa questa tecnologia siano innumerevoli: dai monumenti che forniscono informazioni su sé stessi, ai sensori che monitorano l’attività cardiaca e ne inviano i dati a un’applicazione installata su un *device*.

Questa interazione tra oggetti avviene mediante l'utilizzo di piccoli *chip* che raccolgono informazioni. Queste vengono trasmesse, tramite frequenze radio o Bluetooth, a un dispositivo, come uno *smartphone*, e vengono elaborate da un programma installato sul *device* che dà loro un senso grafico, strutturale, logico e comprensibile.



**Figura 2.1:** Una rappresentazione grafica dell’ecosistema IoT

La possibilità, da parte di un oggetto, di fornire dati non è una cosa nuova: gli storici **codici a barre**, in effetti, rappresentano un forma primordiale di questa tecnologia, la quale, allo stato attuale, trova la sua maggiore espressione, diffusione e potenza nei *beacon* BLE. Non bisogna quindi considerare queste due tecnologie come qualcosa di differente, di opposto: esse rappresentano soltanto *step* successivi (in ordine cronologico) di un unico percorso evolutivo, in cui ognuna delle varie fasi differisce dalla precedente per la ricerca di soluzioni ai problemi che la caratterizzavano.

I codici a barre hanno costituito (e costituiscono tuttora) uno strumento molto utile ma sono purtroppo caratterizzati, per loro natura, da diversi punti di debolezza, primi fra tutti il tempo necessario per effettuare una scannerizzazione del codice e l'impossibilità di fornire dati in tempo reale.

Non a caso, proprio i codici a barre costituiscono il segmento principale del percorso evolutivo che stiamo considerando. In figura 2.2 è rappresentato un codice a barre d'esempio:



**Figura 2.2:** Un esempio di codice a barre

I loro successori sono stati i **tag RFID** (*Radio-Frequency Identification*), che si presentano come un semplice *chip* di silicio dotato di antenna, disponibile in varie forme ma sempre di ridotte dimensioni. Essi sono in grado di memorizzare un numero di serie o altre informazioni limitate che vengono trasmesse poi tramite onde radio. Per attivarli è necessario un lettore RFID che invia dei segnali al *tag* stesso, il quale “risponde alla chiamata” emettendo le sue informazioni. Purtroppo, anche i tag RFID non sono in grado di fornire informazioni in tempo reale e hanno una copertura davvero minima. In figura 2.3 è rappresentato un esempio di *tag* RFID:



**Figura 2.3:** Un esempio di *tag* RFID

Ci sono poi i **tag NFC** che hanno un aspetto, di solito, simile a quello di una moneta, sia per formato che per dimensioni e sono facilmente acquistabili nei negozi sotto forma, ad esempio, di adesivi. Essi si differenziano dai *tag* RFID perché mentre i primi lavorano, come detto poc'anzi, trasmettendo informazioni in modo **univoco** (da *chip* a lettore), i *tag* NFC operano in modalità **biunivoca**. Si tratta di una tecnologia attiva attraverso cui è possibile connettere, in modalità *wireless*, due dispositivi che si trovano a una distanza che va da dieci a venti centimetri. La connessione è **bidirezionale** perché entrambi i dispositivi NFC possono inviare e ricevere informazioni. Non esiste, pertanto, un “fornitore” e un “ricevitore”, bensì due elementi

che si scambiano dati tra di loro. In figura 2.4 è rappresentato un esempio di *tag* NFC:



**Figura 2.4:** Un esempio di *tag* NFC

L'ultima fase evolutiva, quella attuale, è costituita dai *beacon* BLE, i *tag* di più recente sviluppo, uno sviluppo che è stato promosso, addirittura, da aziende come Apple e Google, le quali hanno prodotto le loro versioni di *beacon* proprietari: **iBeacon** nel caso di Apple ed **EddyStone** nel caso di Google. La personalizzazione del prodotto da parte dei vari *brand* ha fatto sì che questi dispositivi non abbiano una forma comune. I *beacon* sfruttano una comunicazione Bluetooth a **basso consumo energetico** (sono detti, per l'appunto, BLE, acronimo di *Bluetooth Low Energy*) che opera trasmettendo un codice identificativo in una specifica area di circa cento metri. Rispetto ai *tag* RFID e NFC, essi hanno un raggio d'azione molto più vasto e, invece che trasmettere i dati tramite onde radio, utilizzano la tecnologia Bluetooth. I *beacon* sono i *tag* che più si prestano a essere utilizzati nell'ecosistema IoT, per la distanza da cui riescono a operare e perché sono gli unici a cui possa essere incorporato un sensore. In figura 2.5 è rappresentato un esempio di *beacon* BLE:



**Figura 2.5:** Un esempio di *beacon* BLE

## 2.4 Lo stato dell'arte

Come abbiamo detto, l'applicazione che è stata sviluppata è ibrida, ossia realizzata con un'unica stesura di codice che è stato successivamente compilato e inglobato all'interno di contenitori nativi per generare due versioni del *software*, una compatibile con la piattaforma iOS e una compatibile con il

sistema Android. Di strumenti in grado di aiutare lo sviluppatore a portare a compimento un tale tipo di progettazione *software* ce ne sono molti. In figura 2.6 sono rappresentate le icone di alcuni *framework* per lo sviluppo di applicazioni ibride:

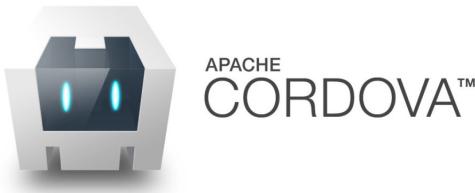


**Figura 2.6:** Alcuni esempi di *framework* per lo sviluppo di applicazioni ibride

È facile confondersi tra le molte piattaforme a disposizione e capire dove ognuna di esse si colloca, pertanto è doveroso e necessario fare chiarezza, soprattutto per quanto riguarda il passaggio da semplice applicazione *web* ad applicazione ibrida. Esistono molti strumenti che permettono di realizzare applicazioni *web* (che si presentano, come detto, sotto forma di siti altamente *responsive*, che cercano di simulare un'esperienza d'uso nativa). Queste applicazioni non sono in grado di accedere alle funzionalità del dispositivo su cui vengono eseguite, o perlomeno hanno accesso a un loro sottoinsieme davvero ristretto. Una volta realizzato il *software*, quindi, ciò che manca è l'inserimento del programma, della sua logica e di tutti i suoi *file* e risorse all'interno di un contenitore nativo, che permetta di rendere questa *web app* un'applicazione a tutti gli effetti, dandogli la facoltà di usufruire delle risorse del dispositivo stesso.

Lo strumento per eccellenza, usato per questo specifico scopo, è stato **Apache Cordova**, un *framework open-source* per lo sviluppo *mobile*. Anch'esso utilizza le tecnologie *web standard* come HTML5, CCS3 e JavaScript. Mediante Cordova, le applicazioni vengono eseguite all'interno di contenitori specifici per ogni piattaforma. Inoltre, questi programmi si basano su API *standard*, che permettono loro di avere accesso alla capacità di un *device* come, ad esempio, ai sensori, ai dati, allo stato della rete, etc. Si chiarifica da solo il fatto che l'utilizzo di Cordova per questo progetto è stato da subito indubbio, ciò che davvero è stato oggetto di analisi è stata la scelta di

quale *framework* utilizzare per creare la logica del programma. In figura 2.7 è rappresentato il logo del *framework* Apache Cordova:



**Figura 2.7:** Il logo del *framework* Apache Cordova

In questo processo di scelta, optare per una piattaforma piuttosto che un'altra non ha dovuto solo riflettere l'attrazione per uno strumento o l'altro, bensì la decisione doveva essere ponderata in base alle possibilità che ognuno di essi metteva a disposizione, in base a quali fossero i suoi punti di forza e aspetti negativi. Dopo un'analisi generale, la scelta è ricaduta su **Ionic Framework**. Ionic è un SDK (*Software Development Kit*) che permette agli sviluppatori di progettare applicazioni *mobile* sfruttando i linguaggi tipici dello sviluppo *web*. Come si può leggere dal sito ufficiale, questo *framework* si focalizza principalmente sul *look and feel* e la *user interface* dell'applicazione.

Invece di utilizzare direttamente il linguaggio JavaScript per la logica dell'applicativo, però, Ionic predilige la scrittura di codice in **TypeScript**, un *super-set* di JavaScript che aggiunge benefici al linguaggio originale quali:

1. la **tipizzazione statica opzionale**;
2. la ***type inference***;
3. l'accesso a **funzionalità ES6 ed ES7**, prima che esse siano supportate dai maggiori *browser*;
4. l'abilità di compilare a una versione di JavaScript che può essere **eseguita su tutti i browser**.

Una caratteristica peculiare di Ionic, oltre al fatto di usare, come abbiamo visto, TypeScript al posto di JavaScript, è quella di basarsi su **Angular**, la piattaforma che costituisce il vero motore di Ionic. È proprio essa, infatti, che gestisce i componenti e le API che costituiscono i mattoni di base dell'intero *framework*. Angular (anche noto come **Angular 2+**, successore del precedente **AngularJS**), è una piattaforma *open source* sviluppata

da Google per lo sviluppo di applicazioni *web*. Si capisce, ora, il motivo del perché Ionic utilizzi il linguaggio TypeScript: esso è proprio il linguaggio di programmazione di Angular e, dato che Ionic si basa su quest’ultimo *framework*, ha preferito seguire la sua strada. È possibile pertanto rappresentare graficamente la struttura di Ionic in questo modo:



**Figura 2.8:** La composizione del *framework* Ionic

In parole poche, Ionic si concentra sul lato *front-end* del *software*, sfruttando la potenza e maturità di Angular per la sua logica e delegando a Cordova il compito di “impacchettare” il programma in un contenitore *mobile* nativo, il quale dà vita a una vera e propria applicazione, pronta per essere caricata sugli *store* digitali, scaricata e installata sui singoli *device*. Alla creazione di un nuovo progetto, Ionic si occupa di caricare automaticamente Cordova e di generare una *directory* in perfetto stile Angular. Al programmatore non resta che scrivere il codice del programma e importare i vari *plugin* necessari.

## 2.5 Le sperimentazioni in contesti simili

In letteratura sono state affrontate diverse sperimentazioni inerenti all’ambito in cui si inserisce l’applicazione che è stata realizzata, ambito che si focalizza su due tecnologie principali, ossia quella dei *beacon* BLE e dei codici QR.

Un primo esempio di ciò lo possiamo trovare nell’articolo “*User centred development of a smartphone application for wayfinding in a complex hospital environment*” [1]. Questo progetto ha previsto la realizzazione di un’applicazione per *smartphone*, chiamata **PowerNav**, che utilizza un sistema di tracciamento e di localizzazione abilitato da *beacon* BLE posizionati sui soffitti dei corridoi di un ospedale. Il progetto doveva rispondere all’esigenza di aiutare le persone, come pazienti e visitatori, a trovare facilmente la loro destinazione all’interno di un complesso ospedaliero, operazione che tende a rivelarsi spesso piuttosto ostica e frustrante e che porta, molte volte, a lamentele, mancati appuntamenti e inefficienze.

Un secondo esempio è reperibile in “*Smart office energy management system using bluetooth low energy based beacons and a mobile app*” [2], in cui viene introdotto un sistema per la gestione energetica di uno *smart office*. Tale sistema è stato pensato per ridurre il consumo di energia dei PC, dei *monitor* e delle luci all’interno di un ufficio attraverso l’utilizzo di un’applicazione *mobile* e di *beacon* BLE. Essi vengono posizionati in diversi posti all’interno di un ufficio, mediante cui l’applicazione riesce a determinare se l’utente entra o esce dalla stanza, cambiando di conseguenza la modalità di risparmio energetico dei dispositivi, nonché delle luci.

Come ultimo esempio per quanto concerne la tecnologia dei *beacon*, si cita “*Exploring location-based augmented reality experience in museums*” [3], un testo in cui si pone l’accento su come la realtà aumentata e i *beacon* BLE siano tecnologie che hanno destato particolare interesse. Lo studio utilizza questi due strumenti per sviluppare un’applicazione che funga da guida ai musei, progettando il contenuto e le funzioni del *software* in base alla teoria della ricchezza multimediale.

Per quanto riguarda, invece, la tecnologia dei codici QR, è utile menzionare l’articolo “*Interactive tourist guide: Connecting web 2.0, augmented reality and QR codes*” [4]. Esso si avvale di tre tecnologie al fine di realizzare un progetto di visita guidata a due dei percorsi principali in una città patrimonio dell’umanità. Una di queste tre tecnologie consiste nell’utilizzo dei codici QR che, posizionati in prossimità di un’immagine, danno accesso a informazioni sia testuali che multimediali attraverso un sito *web* creato apposta per tale progetto.

Si considera anche il lavoro svolto in “*The design of a mobile navigation system based on QR codes for historic buildings*” [5], in cui si discutono le caratteristiche, i vantaggi e gli svantaggi dell’attuale tecnologia di navigazione *mobile*, soprattutto nell’ottica di implementare un sistema di navigazione basato su codici QR. Tale studio si pone come obiettivo quello di risolvere i problemi affrontati e discussi nel testo, al fine di costruire un ambiente di navigazione digitale accessibile che consente agli utenti di comprendere chiaramente ogni oggetto esposto e la sua posizione.

## Capitolo 3

# Panoramica del progetto

## Introduzione

In questo capitolo si presenta la gestione generale del progetto. Nel paragrafo 3.1 si spiega come l'applicazione utilizzi solo una delle due tecnologie presentate, ossia i codici QR. Nel paragrafo 3.2 si fornisce la spiegazione di come funziona l'applicazione. Nel paragrafo 3.3 si danno indicazioni su come si è scelto di gestire l'applicazione. Nel paragrafo 3.4 vengono presentati gli *screenshot* del prototipo realizzato prima della stesura del codice. Nel capitolo 3.5 si mostrano i comandi e i passaggi mediante cui è stata creata la *directory* di progetto. Nel capitolo 3.6 si danno alcune indicazioni riguardo al supporto dei *browser*.

### 3.1 Premesse

È doveroso, a questo punto, fare un paio di premesse introduttive. Sebbene finora sia stato detto che le funzionalità peculiari dell'applicazione sono la possibilità di effettuare una scannerizzazione dei codici QR e la capacità di reagire alla presenza di un *beacon* BLE nelle vicinanze, lo sviluppo vero e proprio si è concentrato solo ed esclusivamente sulla prima tecnologia, i codici QR. Pertanto si assume che, nel *software*, sia già stata scritta la parte di codice relativa ai *beacon* e non riportata nel presente elaborato.

Altro elemento fondamentale che ha caratterizzato lo sviluppo di questa applicazione è stata la totale mancanza di un qualsiasi testo, anche fittizio, che potesse fungere da *placeholder* per “popolare” l'interfaccia utente con dei dati d'esempio. Al solo scopo di riempire tutti quegli spazi vuoti che avrebbero dato l'idea di un qualcosa di incompleto, incompiuto, sono stati

utilizzati come esempio due luoghi con i relativi artefatti in essi custoditi. Il primo luogo scelto, rappresentato in figura 3.1, è l'**Aula Baratto**:



**Figura 3.1:** L'Aula Baratto

al cui interno sono custoditi due artefatti, entrambi affreschi: *Venezia, l'Italia e gli Studi*, rappresentato in figura 3.2, e *La Scuola*, rappresentato in figura 3.3:



**Figura 3.2:** *Venezia, l'Italia e gli Studi*



**Figura 3.3:** *La Scuola*

Il secondo luogo considerato è il **Cortile della Niobe**, rappresentato in figura 3.4:



**Figura 3.4:** Il Cortile della Niobe

e, come artefatto, l'omonima *Scultura della Niobe*, rappresentata in figura 3.5:



**Figura 3.5:** La Scultura della Niobe

## 3.2 Il funzionamento dell'applicazione

La prima fase del progetto si è svolta nel modo in cui è stato spiegato nelle pagine precedenti, ossia cercando di orientarsi e di prendere confidenza con i vari *framework* disponibili, valutando i pro e i contro di ognuno. Dopo aver scelto la piattaforma con cui procedere e prima di aver cominciato a scrivere una sola riga di codice, è stato necessario, naturalmente, riflettere sulla logica generale dell'applicazione. Innanzitutto si è pensato a come il programma dovesse funzionare e che cosa avrebbe dovuto fare. Gli elementi di base per questo progetto sono stati, come detto, i concetti di “luogo” e di “artefatto”: essi rappresentano le entità che governano tutta la logica del *software*. Allo stesso modo, è stato necessario tessere una serie di collegamenti che legassero tali entità alla funzionalità di scansione dei codici QR.

In generale, la struttura di un'applicazione si focalizza attorno al concetto di *pagina*, che può essere tranquillamente paragonata a quella di un comune sito *web*, la quale si apre quando, ad esempio, si effettua un *click* su uno dei pulsanti presenti nel menù di navigazione. Se si considera un semplice sito *web* statico, in cui il menù principale riporta le voci *home*, *blog*, *about* e *contatti*, ognuna di queste sezioni rappresenta una pagina del sito stesso. Cliccando su *home* si apre la relativa pagina. Allo stesso modo, questa applicazione è dotata di **cinque pagine**:

1. una **pagina iniziale**, o *homepage*;

2. una pagina che presenta una **lista dei vari luoghi** e che permette di navigare manualmente tra di essi;
3. una pagina che descrive ed elenca le **caratteristiche di un determinato luogo**, compresa una lista degli artefatti ivi disponibili;
4. una pagina che presenta una **lista dei vari artefatti** e che permette di navigare manualmente tra di essi;
5. una pagina che descrive ed elenca le **caratteristiche di uno specifico artefatto**.

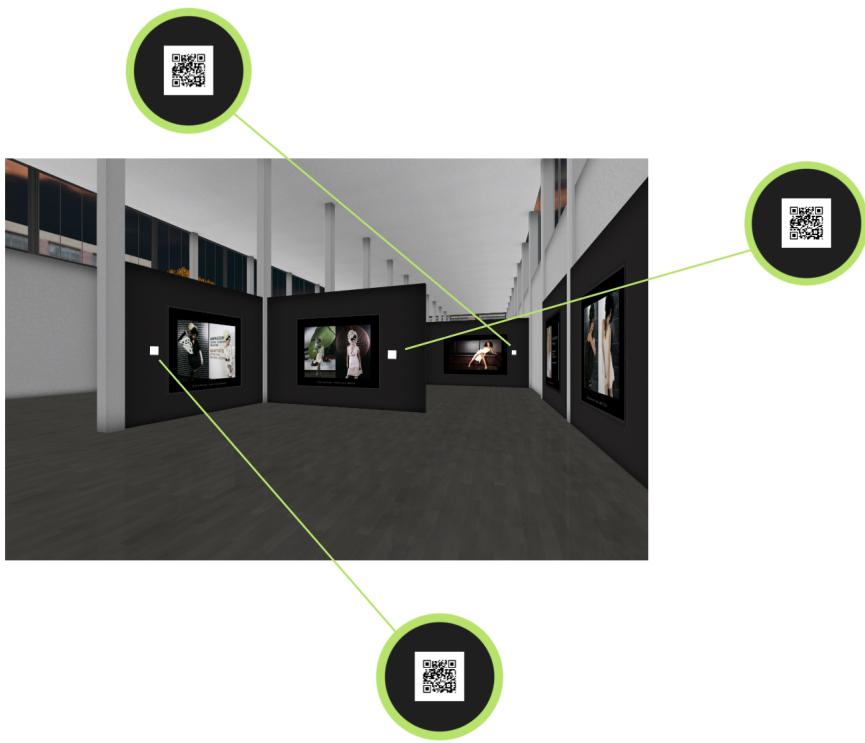
All'apertura del programma, l'utente viene naturalmente accolto dalla pagina iniziale. Per quanto concerne la *user interface*, essa presenta fondamentalmente **tre pulsanti**:

1. *Scannerizza codice QR*;
2. *Sfoglia i luoghi*;
3. *Sfoglia gli artefatti*.

Il funzionamento è semplice. Cliccando sul primo pulsante, si apre la fotocamera dello *smartphone*, pronta ad eseguire la scannerizzazione di un codice. Cliccando sul secondo pulsante, invece, l'utente viene condotto verso un'altra pagina, quella che presenta la lista di tutti i luoghi tra cui è possibile navigare. In questo caso, senza la necessità di alcun codice QR, egli è libero di consultare le informazioni del luogo che preferisce. Cliccando sulla terza voce, infine, l'utente viene indirizzato ad un'altra pagina ancora, ossia quella che presenta la lista di tutti gli artefatti. Anche in questo caso è libero di consultare le informazioni dell'artefatto che preferisce, senza la necessità di alcun codice.

Come spiegato nell'introduzione, i codici QR utilizzati per questa applicazione sono di due tipi differenti: essi possono, infatti, identificare un luogo o un singolo artefatto. Naturalmente, la scansione di un codice ha, come effetto, l'apertura di pagine diverse, a seconda che l'utente ne abbia acquisito uno relativo a un luogo oppure a un oggetto. Nel primo caso, viene aperta la pagina che contiene tutte le informazioni relative al luogo stesso, in cui trovano spazio una descrizione dell'ambiente, alcune fotografie, magari un video di presentazione e l'elenco di tutti gli artefatti che sono custoditi in quello specifico luogo. Nel caso in cui, invece, l'utente scansioni un codice inerente a un artefatto, si apre un altro tipo di pagina, che elenca tutte le caratteristiche dell'oggetto in esame. Logicamente, questa navigazione tra luoghi ed elementi non è possibile solo tramite i codici. Come abbiamo detto, infatti, nella pagina principale sono presenti due pulsanti che permettono di navigare tra i vari luoghi e artefatti. Cliccando sulla voce che corrisponde a un luogo presente nell'elenco, si apre la stessa pagina di descrizione che si

aprirebbe con la scansione del codice relativo al medesimo luogo. Allo stesso modo, cliccando su un *record* della lista degli artefatti (raggiungibile o dalla pagina iniziale, o dall'elenco nella pagina di un luogo), viene aperta la stessa pagina di descrizione che si aprirebbe nel caso in cui l'utente scansioni il codice QR posto nelle vicinanze dell'artefatto stesso. In figura 3.6 viene presentato il *render* 3D di una stanza, al cui interno gli artefatti presentano un codice QR identificativo:



**Figura 3.6:** Codici QR posti in prossimità di artefatti

Lo stesso, identico procedimento avviene con i *beacon*, con la sola differenza che, come detto, essi non sono presenti in prossimità dei singoli artefatti, ma sono disponibili solo per i luoghi, posti in prossimità dei loro ingressi. Una volta che l'utente, con il suo *device*, si trova nel raggio d'azione di un *beacon*, l'applicazione ne rivela la presenza e apre automaticamente la pagina di informazioni di quello specifico luogo, comprensiva della lista degli artefatti lì dentro presenti. In figura 3.7 viene mostrato come si potrebbero posizionare un *beacon* e un codice nelle vicinanze dell'ingresso di un luogo:

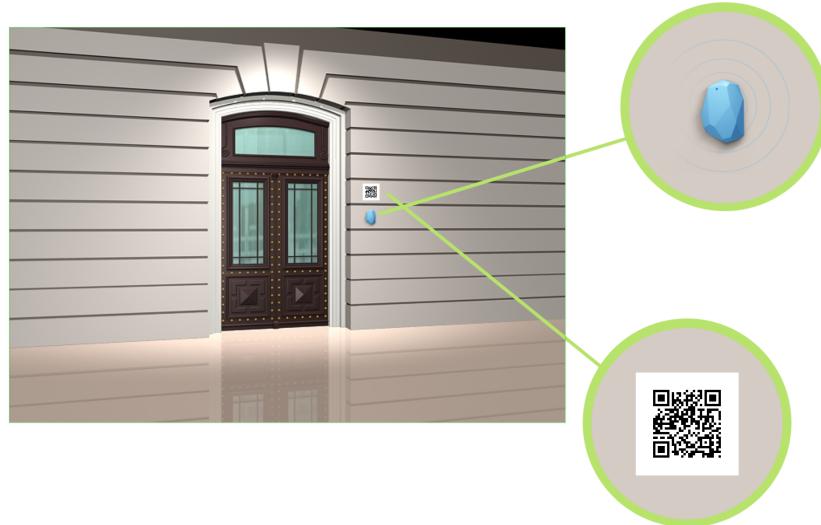


Figura 3.7: Un *beacon* e un codice QR in prossimità di un ingresso

### 3.3 La gestione dell'applicazione

Fondamentalmente, l'applicazione opera in modalità **offline**, il che significa che non necessita di una costante connessione a Internet per poter funzionare. Questo rappresenta un indubbio vantaggio, dal momento che le informazioni possono essere sempre disponibili per la consultazione. L'abilità di operare senza connessione è una conseguenza del fatto che i dati vengono salvati **in locale**, proprio all'interno dell'applicazione (ergo, fisicamente), nella memoria del *device*. Se da una parte ciò costituisce un vantaggio per l'utente, ci sono però anche dei risvolti negativi che scaturiscono da questa scelta e che sono, principalmente, due. Il primo riguarda l'**aggiornamento** dei dati: dal momento che tutte le informazioni a cui il *software* attinge, organizzate in una struttura di cartelle e *file*, si trovano all'interno dello stesso, per poter aggiornare questi dati è necessario aggiornare l'intera applicazione. Questo procedimento è naturalmente molto più laborioso e *time-consuming* rispetto al mero scaricare i dati aggiornati, magari da un **server web**. Infatti, se sfruttando un *server* basterebbe effettuare l'*upload* delle nuove informazioni (procedimento che richiede un lasso di tempo relativamente breve, in rapporto, chiaro, alla velocità di connessione di cui si dispone), aggiornare l'intero programma implicherebbe dover caricare sugli *store* digitali la nuova versione del *software*, operazione che non è mai immediata. Un semplice aggiornamento di un *file* comporta l'aggiornamento di tutto il programma.

Il secondo problema si lega inevitabilmente al primo e concerne le **dimensioni** effettive dell'applicazione, ossia la quantità di spazio di archiviazione che essa occupa nella memoria del *device*. Dovendo racchiudere, all'interno del programma, tutti i file necessari per una piena consultazione dei dati relativi a luoghi e artefatti, la quantità di spazio richiesta aumenta man mano che questi *file* divengono più numerosi e pesanti. Basti pensare a degli ipotetici video di presentazione di un luogo e di descrizione e storia di un artefatto. Essi rappresenterebbero, di sicuro, gli elementi più pesanti all'interno del *software*. Incorporarne uno per ogni stanza e/o, addirittura, uno per ogni artefatto, porterebbe presto l'applicazione ad occupare una quantità notevole di MB, se non perfino di GB. Funzionando mediante connessione a Internet tutto ciò sarebbe facilmente evitabile; ad esempio, sarebbe possibile caricare video sulla piattaforma gratuita **YouTube**, utilizzando le API messe a disposizione dal servizio stesso, mediante cui è possibile utilizzare un *player* integrato per riprodurre video direttamente all'interno dell'applicazione, personalizzando anche l'esperienza di riproduzione.

### 3.4 L'ideazione del prototipo

La fase successiva si è articolata nella realizzazione di un prototipo che rappresentasse il modo in cui è possibile interagire con l'interfaccia grafica del *software* e come le pagine stesse si susseguono. Così come molteplici sono le piattaforme che permettono di sviluppare applicazioni *web*, così numerosi sono i programmi e i servizi *online* che aiutano nella creazione di prototipi per applicativi e siti *web*. È stato utilizzato, a questo scopo, il programma **Adobe XD**, un software molto intuitivo e di facile utilizzo, che ha permesso di realizzare un ottimo *wireframe* dell'applicazione. In figura 3.8 è rappresentata l'icona di Adobe XD:

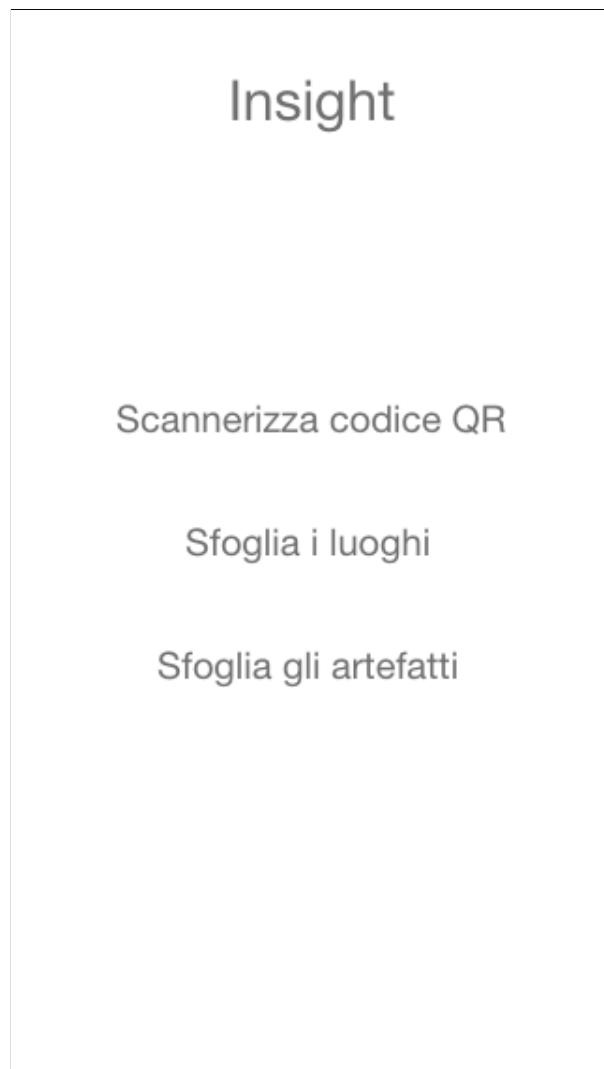


**Figura 3.8:** L'icona di Adobe XD

Le pagine che costituiscono l'applicazione sono, come detto, cinque:

1. la pagina principale;
2. la pagina della lista dei luoghi;
3. la pagina dei dettagli di un luogo;
4. la pagina della lista degli artefatti;
5. la pagina dei dettagli di un artefatto.

Naturalmente, gli abbozzi realizzati per ciascuna di esse sono stati molto approssimativi, senza colore (soltanto in scala di grigi), icone o un *layout* definito. In figura 3.9 è riportato il *mockup* della pagina principale:



**Figura 3.9:** Il *mockup* della pagina principale

Come si può vedere, sono presenti i tre pulsati principali citati nel paragrafo 3.2. Il primo permette di aprire la fotocamera per acquisire un codice QR, il secondo permette di sfogliare manualmente l'elenco completo con tutti i luoghi, il terzo consente di sfogliare la lista degli artefatti. Cliccando su questi ultimi due pulsanti, l'utente viene condotto, rispettivamente, alle seguenti pagine, il cui *mockup* è riportato nelle figure 3.10 e 3.11:



**Figura 3.10:** Il *mockup* della pagina con la lista dei luoghi



**Figura 3.11:** Il *mockup* della pagina con la lista degli artefatti

Vale la pena ricordare che questi luoghi, così come gli artefatti che custodiscono, sono stati inseriti solo per dare un senso logico e grafico all'applicazione e al suo funzionamento. Cliccando su un qualsiasi *record* presente nella lista della prima pagina, si apre una seconda pagina che viene popolata all'istante con tutti i dettagli di quello specifico luogo (descrizione, video, URL utili, etc.), così come una lista degli artefatti presenti. In figura 3.12 è rappresentato il *mockup* della pagina relativa a un luogo:



**Figura 3.12:** Il *mockup* della pagina relativa a un luogo

Come si nota in fondo alla pagina, è presente la sopracitata lista di artefatti. Eseguendo un *tap* sulla voce relativa a un artefatto si apre la pagina che viene, anch'essa, popolata automaticamente con tutti i dettagli di quel preciso elemento. In figura 3.13 è rappresentato il *mockup* della pagina inerente a un artefatto:



**Figura 3.13:** Il *mockup* della pagina relativa a un artefatto

Quest'ultima pagina è la stessa che si ottiene se l'utente seleziona un *record* dalla lista degli artefatti presenti nella relativa pagina. Come si vede, in tutte le pagine, tranne quella principale, è presente in alto a sinistra l'apposito pulsante per ritornare alla pagina precedente, secondo la logica dello **stack di navigazione**, concetto che viene affrontato nel paragrafo 4.3.

### 3.5 La preparazione del progetto

Scelte le modalità di sviluppo dell'applicazione, della gestione delle sue funzionalità e dopo aver preparato un prototipo iniziale, si è passati alla fase di preparazione della cartella di progetto, contenente tutti i *file* necessari. Come detto nel paragrafo 2.4, è stato utilizzato il *framework* Ionic, basato su Angular, per sfruttare infine Apache Cordova al fine di inglobare il *software* all'interno di un contenitore nativo.

Innanzitutto è stato necessario installare, naturalmente, Ionic Framework. Esso predilige la creazione e lo sviluppo di applicazioni tramite l'*utility* a riga di comando, spesso abbreviata in **CLI** (*Command Line Utility*). Dal momento, poi, che l'intero progetto è basato su Cordova, è stato necessario procedere anche alla sua installazione. La maggior parte degli strumenti di sviluppo mediante CLI sono costruiti su **Node.js** e vengono gestiti tramite il suo apposito gestore di pacchetti, **npm** (*Node Package Manager*). Dall'omonimo sito ufficiale è stato quindi scaricato l'eseguibile di Node.js, che ha portato all'installazione sia dell'ambiente *runtime* JavaScript che del gestore dei pacchetti. Installati questi due strumenti fondamentali, non restava altro che installare Ionic e la **CLI** di Cordova, operazione portata a termine mediante il seguente comando da terminale:

```
>_ npm install -g ionic cordova
```

Fatto questo, si è passati alla creazione della cartella di progetto. A tale scopo, il comando utilizzato, sempre mediante terminale, è stato il seguente:

```
>_ ionic start Insight blank
```

In questo comando, **Insight** rappresenta il nome del *folder* in cui sono stati scaricati tutti i *file* necessari per il progetto, mentre **blank** rappresenta uno dei *template* che Ionic mette a disposizione degli sviluppatori. Utilizzando questo particolare *template*, viene generato un *codebase* essenziale, ridotto al minimo e privo di qualsiasi funzionalità integrata, permettendo di cominciare con un codice pulito e senza caratteristiche e funzionalità non pertinenti o inutili.

Una caratteristica molto utile che il *framework* mette a disposizione degli sviluppatori è la possibilità di testare al volo l'applicazione in fase di sviluppo, mediante un *web server* integrato che, di *default*, apre automaticamente una pagina del *browser* all'indirizzo specificato e abilitando il **live-reload**. Questo significa che, al salvataggio di un *file* del progetto, viene eseguito il *refresh* dell'intero programma, così come quello del *browser*, in modo tale da poter verificare subito il corretto funzionamento dell'applicazione e poter disporre della sua versione più aggiornata. Questo permette di evitare il

tedioso procedimento di riavviare, manualmente, il *server*. Per questa funzionalità, è bastato entrare nella cartella del progetto ed eseguire l'opportuno comando:

```
>_ cd Insight  
>_ ionic serve
```

Testare l'applicazione all'interno del *browser* è un modo facile, veloce e conveniente quando essa si trova ancora in una fase embrionale, di sviluppo. Naturalmente, alla fine, è stato necessario eseguire dei *test* accurati su un dispositivo reale. Questo non solo è l'unico modo per verificare, in maniera precisa, il corretto funzionamento e comportamento del programma all'interno di un vera piattaforma *mobile*, ma molti *plugin* nativi di Ionic funzionano soltanto quando vengono eseguiti su un *device* fisico. Questo è il motivo per cui è stato necessario “produrre” due apposite versioni dell'applicazione per due piattaforme differenti. Come detto, le piattaforme *target* per questo progetto sono state iOS e Android.

### 3.5.1 La versione per Android

Produrre una versione del programma compatibile con il sistema di casa Google è stato un processo piuttosto semplice, che ha richiesto la presenza dei seguenti elementi:

1. il **Java SDK**;
2. **Android Studio**;
3. gli strumenti, la piattaforma e le dipendenze dei componenti aggiornati di Android SDK, disponibili tramite l'SDK Manager di Android Studio stesso.

In figura 3.14 è rappresentata l'icona del programma Android Studio:



**Figura 3.14:** L'icona di Android Studio

Per eseguire l'applicazione, è bastato abilitare l'**USB debugging** e la **Developer mode** dalle impostazioni del *device*, lanciando poi il comando:

```
>_ ionic code run android --device
```

Esso produce una *build debug* dell'applicazione, sia per quanto riguarda il codice di Ionic che quello di Android. Per eseguire o per compilare il *software* per la produzione, il comando da utilizzare era il seguente:

```
>_ ionic cordova run android --prod --release
```

oppure, in alternativa:

```
>_ ionic cordova build android --prod --release
```

Questi ultimi due hanno, come effetto, quello di minimizzare il codice del programma e di rimuovere qualsiasi funzionalità di *debugging* dall'APK. In generale, questi procedimenti vengono avviati quando si effettua il *deploy* dell'applicazione nel Google Play. Nel momento in cui si decide di rilasciare il *software* nello *store* digitale, è necessario **firmare** il *file APK*, creando per esso un nuovo certificato. Anche per questo procedimento, sul sito di Ionic sono riportati tutti i passi da seguire.

### 3.5.2 La versione per iOS

A differenza di Android, gli sviluppatori iOS hanno bisogno di generare, per il *testing*, il cosiddetto *provisioning profile*, una collezione di entità digitali che lega univocamente gli sviluppatori e i dispositivi a un *iPhone development team* autorizzato e che permette che un *device* venga utilizzato per eseguire *test* al suo interno. A partire da iOS 9, è possibile sviluppare e testare applicazioni sui dispositivi iOS senza un **Apple developer account**, che si ricorda essere a pagamento. Di sicuro, questo rappresenta un enorme aiuto e passo avanti, in quanto rende gli sviluppatori in grado di provare “con mano” lo sviluppo *mobile* mediante Ionic, dato che permette di risparmiare sul costo di creazione dell'*account* fornendo, al tempo stesso, un gran numero delle funzionalità che si avevano, originariamente, solo con un *Apple developer account* completo. I requisiti, in questo caso, sono:

1. **Xcode 7** o superiore;
2. **iOS 9**;
3. un **Apple ID** gratuito o un *Apple developer account* a pagamento.

L'IDE necessario per poter creare applicazioni iOS è **Xcode**, l'editor ufficiale di Apple. In figura 3.15 è rappresentata l'icona del programma Xcode:



**Figura 3.15:** L'icona di Xcode

Per prima cosa, è stato necessario creare il *provisioning profile*, procedimento che, sul sito di Ionic, è riportato sia nel caso dell'Apple ID che dell'*Apple developer account*. Per eseguire l'applicazione si lanciava il comando:

```
>_ ionic cordova build ios --prod
```

aprendo, in Xcode, il *file* con estensione `.xcodeproj` presente nella cartella `./platforms/ios/`, connettendo il *device* con l'apposito cavo USB e selezionando il dispositivo come *target*. Il tutto si concludeva cliccando sul pulsante *play*, posto in alto a sinistra nell'*editor*. Per far ciò è stato necessario firmare l'applicazione, scegliendo l'opzione *trusting the certificate*. Infine, il comando:

```
>_ ionic cordova run ios --device
```

permetteva di installare e lanciare l'applicazione direttamente dal terminale.

## 3.6 Il supporto dei browser

È utile ricordare che Ionic Framework si focalizza sia nella progettazione di applicazioni native e ibride tramite Cordova, sia nello sviluppo delle cosiddette PWA (*Progressive Web App*). Detto ciò, è stato fondamentale capire quali erano i requisiti, in termini di sistemi operativi e *browser*, che dovevano essere soddisfatti per garantire il corretto funzionamento del *framework*. Per quanti riguarda gli OS, Ionic supporta la piattaforma **iOS 8 e successive**, mentre per quanto concerne il sistema di casa Google, **Android 4.4 e successive**. In ambito prettamente *browser*, infine, Ionic supporta **Safari, Chrome ed Edge**.

## Capitolo 4

# Una panoramica dei concetti e dei meccanismi fondamentali

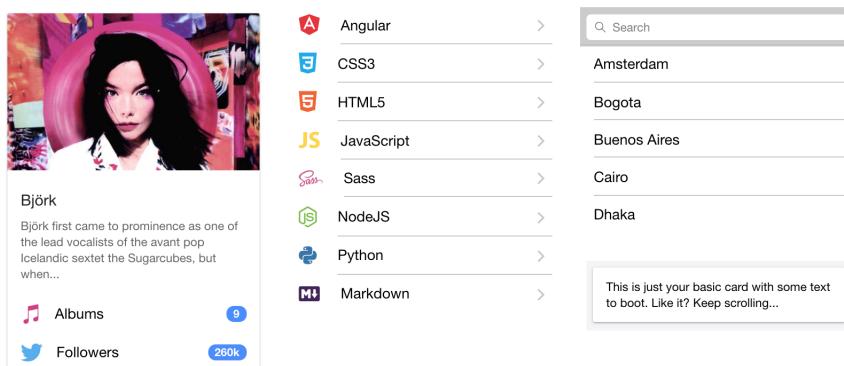
### Introduzione

In questo capitolo si affrontano le spiegazioni di alcuni concetti che sono stati alla base dell'interno percorso di sviluppo e la cui comprensione è necessaria per capire il modo in cui l'applicazione è stata realizzata. Nel paragrafo 4.1 viene affrontata la spiegazione di cosa siano i componenti e le pagine e di quale approccio sia stato preferito per la progettazione del *software*. Nel paragrafo 4.2 si introduce il *framework* Angular, la piattaforma progettata da Google sulla cui logica di funzionamento è stato realizzato Ionic. Nel paragrafo 4.3 si tratta di come venga gestito il meccanismo della navigazione all'interno di una qualsiasi applicazione Ionic. Nel paragrafo 4.4 viene approfondita la questione riguardante i componenti aggiuntivi, o *plugin*, che possono essere usati per aggiungere funzionalità, nonché si indica quali tra di essi sono stati utilizzati per questo progetto. Nel paragrafo 4.5 si spiega come siano strutturati i dati nascosti all'interno dei codici QR, nel caso di un codice relativo sia a un luogo, che a un artefatto. Nel paragrafo 4.6, infine, si introduce e si spiega il concetto di *provider*, uno strumento fondamentale di cui il *framework* Ionic è provvisto e che permette di affrontare la gestione dei servizi, dei dati e delle informazioni che vengono utilizzate dal *software*. La maggior parte dei concetti affrontati in questo capitolo verranno ripresi in maniera molto più dettagliata (assieme al relativo codice) nel capitolo 5.

## 4.1 I componenti e le pagine

All'interno dell'ecosistema Ionic il concetto di **component**, in italiano “componente”, indica un elemento basilare della GUI (*Graphical User Interface*) per lo sviluppo di un'applicazione *mobile*. Questo genere di “entità” è costituito, da un punto di vista sintattico, da semplice codice HTML, CSS e JavaScript. Nello specifico, la caratteristica fondamentale di un componente Ionic sta nella sua abilità di adattarsi alla piattaforma su cui il *software* viene installato ed eseguito.

Come detto, le applicazioni Ionic sono, nella loro sostanza, costituite da componenti che permettono di realizzare facilmente un'interfaccia grafica. Ionic fornisce un gran numero di componenti da poter utilizzare come, ad esempio, **finestre modali**, **popup** e così via. Ognuno di essi, a sua volta, fornisce uno specifico insieme di **API** che permette di personalizzare tale elemento come meglio si desidera. A titolo esemplificativo, in figura 4.1 sono raffigurati alcuni esempi di componenti di base forniti dal *framework* Ionic:



**Figura 4.1:** Alcuni componenti di base forniti da Ionic: liste, *card* e *badge*

Sebbene questi elementi costituiscano i mattoni base per la strutturazione di una qualsiasi applicazione, al fine di sviluppare il *software* in esame non è stato necessario utilizzare elementi così granulari come i componenti. Invece, il funzionamento della logica del programma è stato tessuto attorno al concetto di **pagina**. Gli abbozzi che sono stati presentati nel paragrafo 3.4 sono veri e propri esempi di pagine che costituiscono l'applicazione che è stata realizzata e tra cui l'utente può liberamente navigare. Una pagina, quindi, costituisce una schermata distinta da tutte le altre e che ricopre uno specifico ruolo.

## 4.2 Come funziona Angular

Finora ci si è soltanto limitati a citare Angular, il *framework* di casa Google sulla cui logica è basata la realizzazione dell'applicazione, così come non è stato posto l'accento sulla filosofia e il funzionamento di tale piattaforma, la quale sta letteralmente alla base di Ionic, regolando in tutto e per tutto la progettazione, il funzionamento e la struttura delle sue applicazioni. Tutto ciò verrà affrontato ora, sebbene da un punto di vista generale, in quanto non rientra negli obiettivi di questo testo affrontare una spiegazione dettagliata del *framework*. In figura 4.2 è riportato il logo del *framework* sviluppato da Google:



**Figura 4.2:** Il logo del *framework* Angular

Com’è stato detto nel paragrafo 4.1, lo sviluppo dell’applicazione si è focalizzato attorno al concetto di “pagina”. Angular rende la creazione di queste pagine un processo veloce e semplice. Una pagina è costituita da **tre file**, custoditi all’interno di una *directory* che identifica la pagina stessa. Questi file comprendono:

1. un file **HTML**, che si occupa di contenere tutto il codice che descrive la struttura fisica di una pagina;
2. un file **CSS**, che custodisce il codice che modifica l’estetica della pagina;
3. un file **TypeScript**, che contiene tutto il codice che gestisce la logica, il funzionamento, il comportamento e le funzionalità della pagina.

Questa strutturazione a tre *file* risulta essere molto comoda in quanto permette di separare tutte le porzioni di codice che svolgono funzioni differenti. Il modo in cui questi *file* sono legati tra di loro avviene mediante l’utilizzo di funzionalità come il **data binding**, il quale ad esempio si occupa della

sincronizzazione tra il **modello** e la **view** di una pagina, secondo il classico *pattern* architetturale **MVC** (*Model View Controller*).

Il *file* TypeScript, in particolare, prevede la definizione ed esportazione di una **classe**, la quale viene utilizzata per referenziare l'omonima pagina: se si fosse creata, ad esempio, una pagina “contatti”, il *file* TypeScript avrebbe definito ed esportato probabilmente una classe **ContactPage**. In questo modo una pagina può essere utilizzata all'interno di un'altra semplicemente scrivendo l'appropriato enunciato di **import** all'inizio del file TypeScript. Questo procedimento è necessario quando, ad esempio, una pagina prevede la presenza di un pulsante che indirizza l'utente verso un'altra pagina: affinché il codice che gestisce la navigazione sia valido e funzionante, questa seconda pagina dev'essere importata all'interno della precedente. Il **data binding** prevede la possibilità di legare le variabili che compaiono all'interno di una classe a dei *placeholder* presenti nel codice HTML. In particolare, Angular prevede l'adozione del **two-way data binding**, tramite cui non solo la modifica di una variabile permette la modifica del relativo *placeholder* nel codice HTML, ma tutto ciò può avvenire anche a parti invertite: settando un *placeholder* a un determinato valore verrà aggiornato anche quello della variabile della classe.

Angular, inoltre, è fornito di un elevato numero di funzionalità e di direttive, come la direttiva **\*ngFor**, che permette di replicare un certo frammento di codice iterando su una collezione di oggetti. Essa risulta essere estremamente utile quando, ad esempio, una pagina mostra una lista di elementi. Infine, Angular permette di associare agli elementi della *user interface* degli eventi precisi, come il classico *click* del mouse. Il codice che dev'essere eseguito in seguito a una specifica azione è memorizzato, naturalmente, all'interno del file TypeScript della pagina.

### 4.3 La navigazione

Il **NavController** è la classe base responsabile della navigazione. Questa classe viene sfruttata mediante l'apposito enunciato di **import** presente in cima a tutti i *file* TypeScript relativi a quelle pagine che prevedono, mediante la presenza di elementi come i pulsanti, una modifica alla cronologia di navigazione dell'applicazione.

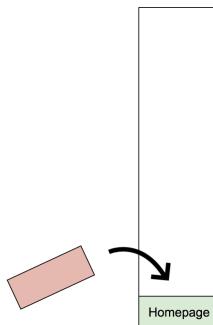
Come detto nel paragrafo 4.1, il *framework* Ionic si focalizza principalmente sul concetto di pagina, la quale costituisce una singola *view* all'interno dell'applicazione. È necessario, ora, capire il modo con cui tale sistema gestisce la navigazione tra di esse, ovvero come viene affrontata la questione del **routing**. Il meccanismo di Ionic si basa su un concetto molto semplice, quello di **stack**: le pagine si accumulano, man mano, l'una sull'altra,

rimosse (aggiunte) dalla (nella) cima della pila quando l'utente naviga tra le *view* dell'applicazione. Il tutto, naturalmente, avviene secondo la classica filosofia **LIFO** (*Last In First Out*) degli *stack*, secondo cui la pagina che si trova in cima corrisponde all'ultima verso cui l'utente ha navigato, è quella attualmente visibile a schermo e su cui egli si trova. Partendo da una pagina predefinita, che funge da “punto di partenza”, si ha la possibilità di navigare verso una qualsiasi altra pagina del programma. Secondo quanto è stato detto, allo stato iniziale, lo *stack* contiene un solo elemento, la suddetta pagina principale:



**Figura 4.3:** All'inizio, lo *stack* contiene un solo elemento, la *homepage*

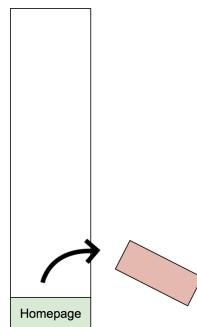
Nel momento in cui, da questa pagina, l'utente clicca un pulsante che lo porta verso un'altra, oppure esegue un'azione che ha la stessa conseguenza (come, nel nostro caso, l'acquisizione con successo di un codice QR), lo *stack* si modifica. Quella determinata pagina viene posizionata in cima alla pila, venendo pertanto visualizzata sul dispositivo:



**Figura 4.4:** Una nuova pagina viene messa in cima allo *stack*

Diventa fondamentale, a questo punto, riuscire a far capire all'utente quale

sia la struttura dell'applicazione. Questo è un elemento da non sottovalutare: un'interfaccia grafica che non riesce a trasmettere un senso di orientamento comporta, spesso, l'abbandono o disinstallazione del *software* stesso. Posizionando un pulsante di ritorno (solitamente nella parte alta, a sinistra del *display*) è un buon metodo in quanto indica, in maniera molto chiara, la possibilità da parte dell'utente di tornare alla pagina (e a uno stato) precedente. Ionic fa tutto questo, automaticamente: come dimostreremo più avanti con il codice, nel momento in cui si implementa il sistema di navigazione base, esso crea un apposito pulsante di ritorno, che fa intuire come, cliccandolo, sia possibile tornare alla *view* precedente. Eseguendo un clic su questo pulsante viene naturalmente modificato lo *stack*:



**Figura 4.5:** L'utente torna alla pagina precedente e lo *stack* si modifica, di nuovo

Una delle caratteristiche (spesso sottovalutate) di un'applicazione è, non a caso, il riuscire a far intuire all'utente quale sia, in ogni momento, la sua posizione rispetto alle varie interfacce. Egli dev'essere in grado di capire come poter tornare a uno stato precedente, così come devono essere chiare le azioni che possono essere effettuate dalla schermata attualmente visualizzata. L'idea di predisporre un pulsante di ritorno, in alto a sinistra della UI, riesce a soddisfare in maniera perfetta il primo requisito. Sapendo di poter contare sulla possibilità di tornare a uno stato precedente rispetto a quello attuale, l'utente non si sentirà mai “spasato”, disperso tra i meandri dell'applicazione. Inoltre, un breve utilizzo del programma lo porterà, in tempi molto rapidi, a intuirne il completo funzionamento, nonché la struttura.

In unione al `NavController`, Ionic fornisce anche la classe  `NavParams`, implementata come un oggetto che esiste all'interno di una specifica pagina e che può contenere dati relativi a quella determinata *view*. Il suo utilizzo diventa chiaro quando è necessario navigare tra le pagine, passando però anche delle informazioni tra di esse. In particolare, tale classe rappresenta una funzionalità che permette di passare un oggetto JSON da una pagina

all'altra. Si considerino, a titolo esemplificativo, due pagine, `firstPage` e `secondPage`. Se si volesse navigare dalla prima alla seconda pagina, trasportando anche delle informazioni, tutto ciò di cui si avrebbe bisogno sarebbe passare un oggetto JSON come secondo argomento della funzione `push()`, ad esempio:

```
this.navCtrl.push(secondPage, {status: true});
```

Per ottenere le informazioni all'interno della seconda pagina, sarebbe sufficiente invocare la funzione `get()` della classe `NavParams`:

```
this.navParams.get("status");
```

e memorizzare, ad esempio, il valore dell'attributo `status` all'interno di una variabile appartenente alla classe che rappresenta la pagina e mostrandolo nella *user interface* mediante il *data binding* con un *placeholder* presente nel codice HTML. Per poter usufruire delle funzionalità della classe `NavParams`, però, essa deve essere elencata tra i parametri formali dei **costruttori** delle pagine che intendono farne uso, assieme naturalmente alla classe `NavController`:

```
constructor(public navCtrl: NavController, public navParams: NavParams) {
  //...
}
```

Riprendendo quanto citato all'inizio di questo paragrafo, quindi, un *controller* di navigazione non è altro che un *array* di pagine che rappresenta una particolare cronologia, vettore che può essere manipolato per navigare all'interno di un'applicazione eseguendo operazioni di **push** e **pop** sullo *stack* che rappresenta la cronologia stessa, con la possibilità, sfruttando la classe `NavParams`, di passare anche dati tra le pagine.

## 4.4 I *plugin*

I **plugin** costituiscono uno strumento essenziale quando si sviluppa un'applicazione ibrida. Infatti, essi fungono da vere e proprie porte d'accesso verso tutte (o quasi) le funzionalità di un dispositivo. L'accesso al **file system**, alla **fotocamera**, ai **contatti**, per esempio, sono tutte operazioni che richiedono di usufruire di risorse che vengono rese possibili grazie all'utilizzo di questi componenti aggiuntivi. Ciò significa che esiste una moltitudine di questi strumenti: è presente quello per poter accedere ad una specifica

risorsa del sistema (come può essere, per l'appunto, il *file system*), quello che permette l'accesso ad un'altra funzionalità (come il Bluetooth), e così via.

Nel paragrafo 2.4 si è discusso su come Ionic sia un *framework* completo e in costante aggiornamento, un ecosistema che rende lo sviluppo di applicazioni ibride un processo rapido e senza particolari difficoltà. Proseguendo su questa filosofia, il *team* che ha sviluppato tale piattaforma ha ideato **Ionic Native**. Si tratta di un vero e proprio contenitore, scritto in linguaggio TypeScript, che racchiude i *plugin* di Cordova e rende davvero semplice aggiungere una funzionalità nativa del *device* a un'applicazione *mobile* Ionic. Questi stessi *plugin*, però, non sono frutto di Ionic e dei suoi sviluppatori: essi sono, come detto, appartenenti al *framework* Cordova. Gli sviluppatori non hanno fatto altro che creare un contenitore che rendesse soltanto molto più immediata l'integrazione di uno specifico componente nel processo di creazione di un'applicazione.

Nel paragrafo 2.2, inoltre, si è fatto riferimento alla delicata questione del corretto funzionamento di questi strumenti, che è necessario ricordare essere *di terze parti*. Ciò significa che non è possibile essere certi del loro completo funzionamento su un determinato dispositivo. Non a caso anche Ionic, sul sito ufficiale, precisa questo fatto:

*“Ionic Native is largely a set of community maintained plugins, and while the community is quick to find and fix issues, certain plugins may not function properly.”*

Fatta questa doverosa precisazione, vale la pena capire come funzionano questi componenti aggiuntivi e come possono essere integrati all'interno di un'applicazione.

#### 4.4.1 *Promise* e *Observable*

Addentrandoci più in profondità e cercando di capire come questi *plugin* operano, si scopre che Ionic Native racchiude le *callback* dei *plugin* in una **promise** o all'interno di un **observable**. Questo permette di fornire un'interfaccia comune per tutti i componenti aggiuntivi. Un esempio viene fornito con il seguente frammento di codice:

```
1 // Importazione di funzionalità e classi necessarie
2 // ...
3 import { Geolocation } from '@ionic-native/geolocation';
4 import { Platform } from 'ionic-angular';
5
6 // Definizione di una classe
7 class MyComponentOrService {
```

```

8  constructor(private platform: Platform, private geoloc: Geolocation) {
9    platform.ready().then(() => {
10      geoloc.getCurrentPosition().then(pos => {
11        console.log('lat: ' + pos.coords.latitude + ', lon: ' +
12          ↵ pos.coords.longitude);
13      });
14      const watch = geoloc.watchPosition().subscribe(pos => {
15        console.log('lat: ' + pos.coords.latitude + ', lon: ' +
16          ↵ pos.coords.longitude);
17      });
18      watch.unsubscribe();
19    });
}

```

#### 4.4.2 Installazione e utilizzo

Per poter aggiungere Ionic Native a una applicazione è sufficiente eseguire il seguente comando per installare il **core package**:

```
>_ npm install @ionic-native/core --save
```

Tale pacchetto è comunque incluso, di *default*, in ogni applicazione Ionic. Fatto ciò, è necessario installare il *package* Ionic Native per ogni *plugin* che si intende aggiungere. Per fare un esempio, se si volesse installare il componente relativo alla fotocamera, sarebbe necessario lanciare il comando:

```
>_ npm install @ionic-native/camera --save
```

Poi, si passerebbe a installare il *plugin* utilizzando Cordova o la Ionic CLI:

```
>_ ionic cordova plugin add cordova-plugin-camera
```

Tutti i nomi dei pacchetti sono certificati dalla documentazione del *plugin* stesso. Inoltre, è raccomandato seguire le istruzioni di installazione di ogni singolo, specifico componente aggiuntivo, dato che alcuni di essi richiedono delle ulteriori fasi successive per poter portare a termine la loro completa installazione. Al termine dell'operazione di installazione del pacchetto è sufficiente aggiungere il componente al **NgModule** dell'applicazione. Ad esempio, nel caso dell'installazione e utilizzo del *plugin* relativo alla fotocamera, il codice sarebbe il seguente:

```

1 // Importazione di funzionalità e classi necessarie
2 // ...
3 import { Camera } from '@ionic-native/camera';
4

```

```

5 // Aggiornamento del NgModule
6 @NgModule({
7   // ...
8   providers: [
9     // Altri provider utilizzati dall'applicazione
10    Camera
11  ]
12})
13
14 // Definizione ed esportazione della classe AppModule
15 export class AppModule { }

```

Naturalmente il codice dei *plugin* che è stato riportato in questo paragrafo non è quello effettivamente utilizzato per la realizzazione dell'applicazione in esame, il quale verrà invece riportato nel capitolo 5.

## 4.5 Il formato dei dati nei codici QR

Per capire alcuni dei frammenti di codice che verranno presentati nel corso di questo capitolo, è necessario spiegare innanzitutto quale sia il formato dei dati e delle informazioni che sono state nascoste all'interno dei codici QR. Questi codici possono essere posizionati in prossimità dell'ingresso di un luogo (come una stanza), oppure in prossimità di un artefatto. In base all'elemento che essi identificano, la loro scannerizzazione avrà un effetto, naturalmente, diverso: l'acquisizione di uno relativo a un luogo farà aprire la pagina che elenca le caratteristiche del luogo stesso, mentre se se ne acquisisce uno relativo a un oggetto, l'applicazione mostra la pagina di descrizione dell'oggetto stesso. Come esempio, in figura 4.6 è riportato il codice QR originale relativo a uno dei luoghi descritti all'interno dell'applicazione, l'Aula Baratto:



**Figura 4.6:** Il codice QR relativo all'Aula Baratto

Come detto nella sezione 3.1, questo luogo custodisce, al suo interno, due affreschi che è possibile considerare come “arteфatti”: *Venezia, l'Italia e gli*

*Studi* e *La Scuola*. La figura 4.7 riporta, sempre a titolo esemplificativo, il codice QR relativo al secondo affresco:



**Figura 4.7:** Il codice QR dell'affresco *La Scuola*

Le informazioni all'interno dei codici sono memorizzate sotto forma di **testo semplice**, secondo una sintassi precisa, ossia quella **JSON**. Per quanto concerne l'Aula Baratto, e in generali tutti i luoghi, la sintassi è la seguente:

```
{  
  "type": "place",  
  "place": "Aula Baratto"  
}
```

Come si vede, sono sufficienti soltanto due proprietà. Il codice relativo, invece, all'affresco *La Scuola* contiene il seguente testo:

```
{  
  "type": "artifact",  
  "place": "Aula Baratto",  
  "name": "La Scuola"  
}
```

In questo caso, le proprietà diventano tre. È importante sottolineare che quelli che sembrano *snippet* di codice sono, in realtà, solo puro e semplice testo, ossia **strighe**, scritte tra una coppia di doppi apici “...”.

## 4.6 I dati e i *provider*

L'applicazione che è stata realizzata opera, com'è stato spiegato nel paragrafo 3.3, in modalità *offline*. Questo significa che i dati, all'interno del programma, sono sempre disponibili per l'accesso e la consultazione e possono popolare le pagine dell'applicazione senza necessitare di una connessione a Internet. L'indubbio vantaggio che da tutto ciò deriva è il fatto che non

è richiesta una costante connessione alla rete, connessione che, come si sa, pregiudica un prolungato uso della batteria di ogni dispositivo. L'altro lato della medaglia, però, non è così favorevole, in quanto l'impossibilità di poter accedere a Internet comporta, inevitabilmente, l'aggiornamento dell'intera applicazione per l'inserimento di nuovi dati.

La gestione di questi dati è stata organizzata mediante un **provider** Ionic. I *provider* costituiscono un elemento fondamentale di una qualsiasi applicazione e sono un elemento altrettanto essenziale per Ionic. Se i componenti costituiscono la struttura, la spina dorsale di un'applicazione ibrida, essi sono pur sempre responsabili solo per ciò che è visibile a schermo in un determinato momento. Invece, i *provider* si occupano di fornire informazioni all'applicazione e di eseguire specifici *task*. Questi strumenti permettono di creare servizi indipendenti che permettono al *software* di godere di qualche tipo di funzionalità. Alcuni degli esempi in cui trova uso l'utilizzo di un *provider* sono:

- **leggere dati** da un *server* remoto;
- **eseguire specifiche operazioni** su dei dati;
- **condividere** dati;
- fornire un insieme di **complicate operazioni matematiche**.

In generale, un *provider* viene sfruttato ogni volta che si richiede all'applicazione di eseguire dei compiti "pesanti". Facendo ricorso a uno strumento di questo tipo, è possibile astrarre la funzionalità che esso fornisce da quelle di un qualsiasi componente, cosa che rende il codice più mantenibile, pulito e chiaro e permette di riutilizzare lo stesso *provider* in più posti differenti all'interno della medesima applicazione (o addirittura, in una applicazione completamente diversa).

Per creare un *provider*, Ionic fornisce un apposito comando da eseguire da terminale:

```
>_ ionic generate provider [name]
```

in cui **name** viene sostituito con il nome che si vuole attribuire al *provider* stesso. Il comando crea automaticamente una sotto cartella, che per questo progetto è stata denominata **data-manager**, all'interno della *directory* `./src/providers/`, contenente un unico *file*, `data-manager.ts`.

# Capitolo 5

## Il codice

### Introduzione

In questo capitolo viene dapprima presentata la struttura della cartella in cui è stata sviluppata l'applicazione e successivamente tutto il codice scritto per la sua realizzazione. Nella sezione 5.1 sono elencati gli elementi principali che compongono la *directory* del progetto, tralasciando quelli di minore importanza al fine di non disperdersi in dettagli non rilevanti e rendere il testo troppo prolisso. Nella sezione 5.2 viene presentato, in maggior dettaglio, il contenuto della *directory* `./src/`, al cui interno si svolge la maggior parte dell'opera di scrittura del codice. Nella sezione 5.3 viene affrontato l'*iter* di creazione delle *view* che danno vita alla *user interface*.

### 5.1 La struttura della cartella di progetto

Il comando `ionic start`, utilizzato per generare la *directory* in cui è stato creato e gestito il progetto, genera una struttura complessa, composta di molte cartelle, sottocartelle annidate e molteplici file. Lo stesso discorso è valso per il progetto dell'applicazione che è stata creata per il tirocinio. In questo caso, tale gerarchia era variabile, in quanto venivano aggiunte nuove *directory* man mano che si includevano nuove piattaforme per la produzione del *software*, nuovi *plugin* o che si creavano, ad esempio, nuovi *provider*. All'interno della *directory* era presente la tipica struttura di un progetto Cordova, in cui era possibile installare componenti aggiuntivi e creare *file* di progetto specifici per una determinata piattaforma. La spiegazione esaustiva di come tale struttura ad albero era composta esula dallo scopo di questo testo, però è necessario discutere, almeno, riguardo agli elementi principali. Per quanto concerne le cartelle, tra le più importanti annoveriamo:

- la cartella `./src/`, in cui è contenuto il codice del *software* e dove avviene la maggior parte del lavoro quando si tratta di progettare un'applicazione con il *framework* Ionic. All'invocazione del comando `ionic serve` (che avvia un server in locale, utile per le fasi di sviluppo e *testing*) viene eseguito il *transpiling* del codice (il quale si trova all'interno di questa *directory*) in un'appropriata versione di JavaScript, supportata dal *browser*. Questo *modus operandi* permette di scrivere codice “a più alto livello” mediante il linguaggio TypeScript, codice che viene successivamente tradotto in un'opportuna versione del naturale linguaggio JavaScript. Molto importante, soprattutto, è il file `./src/app/app.module.ts`, sulla cui parte superiore è possibile individuare il seguente frammento di codice:

```

1  @NgModule({
2    declarations: [
3      MyApp,
4      HelloIonicPage,
5      ItemDetailsPage,
6      ListPage
7    ],
8    imports: [
9      BrowserModule,
10     IonicModule.forRoot(MyApp)
11    ],
12    bootstrap: [
13      IonicApp
14    ],
15    entryComponents: [
16      MyApp,
17      HelloIonicPage,
18      ItemDetailsPage,
19      ListPage
20    ],
21    providers: []
22  })
23
24  export class AppModule {}

```

Ogni applicazione possiede un “modulo radice” il cui compito è, essenzialmente, quello di controllare il resto dell'applicazione. È questo il *file* in cui avviene la fase di avvio (*bootstrap*) dell'applicazione. All'interno di questo modulo viene impostato il componente principale del *software*, presente nel file `./src/app/app.component.ts`. Esso costituisce il primo elemento che viene caricato dall'applicazione ed è, tipicamente, una sorta di *shell*, un guscio vuoto in cui altri componenti possono essere caricati.

- il file `./src/index.html`, che funge da punto di partenza per l'intera applicazione, sebbene il suo unico compito sia quello di impostare gli *script*, includere i file CSS e avviare il *software*. Affinché l'applicazione funzioni Ionic ricerca, all'interno di questo file, il tag `<ion-app>...</ion-app>`, assieme ai seguenti *script*, posizionati verso la fine del codice HTML:

```
<script src="build/polyfills.js"></script>
<script src="build/vendor.js"></script>
<script src="build/main.js"></script>
```

Tutti e tre gli *script* vengono generati automaticamente durante il processo di compilazione del *software*.

- la cartella `./node_modules/`, che contiene tutti i moduli di Node.js necessari per il corretto funzionamento del programma;
- la cartella `./platforms/`, al cui interno è possibile vedere l'elenco completo di tutte le piattaforme per le quali l'applicazione viene compilata (nel nostro caso, sono presenti le sottocartelle `ios` e `android`).
- la cartella `./plugins/`, contenente tutti i componenti aggiuntivi che sono necessari, anche in questo caso, per assicurare il corretto funzionamento dell'applicazione.

## 5.2 La directory `./src/`

Una menzione particolare è doveroso farla per la cartella, poc'anzi citata, in cui è custodito tutto il codice che determina il funzionamento del *software*: la cartella `./src/`. Il suo contenuto si articola in file e sottocartelle, secondo una gerarchia che rispecchia la struttura logica stessa del programma. Come abbiamo detto, l'applicazione si basa principalmente sul concetto di “pagina” ed è suddivisa in tante *view* che svolgono ruoli e azioni ben precise. È presente, ad esempio, una pagina specifica adibita a mostrare tutti i dettagli di uno preciso artefatto, comprensivi di una descrizione, URL utili, video, testo e immagini.

All'interno di questa cartella si possono notare:

- la directory `./src/app/`, che contiene file fondamentali per l'avvio e la gestione dell'intera applicazione, tra cui `app.component.ts` ed `app.module.ts`;
- la directory `./src/assets/`, contenente tutte le risorse statiche che popolano l'interfaccia grafica del *software*, come immagini, icone e *font*;

- il file `./src/index.html` che, come abbiamo detto, funge da *starting point* per il programma;
- la cartella `./src/pages/`, al cui interno si trovano tutte le pagine, suddivise per cartelle, che compongono l'applicazione;
- la cartella `./src/themes/`, che si occupa della gestione dello stile e del tema della *user interface*.

Dopo aver elencato tutti gli elementi principali che costituiscono una *directory* di progetto Ionic, è il momento di discutere riguardo alla creazione delle pagine e della loro composizione in termini di codice sorgente. In seguito verrà trattata la gestione dei *plugin* che sono stati utilizzati per questa applicazione, compresa la loro installazione e la spiegazione di tutto il codice necessario per il loro corretto funzionamento.

### 5.3 La creazione e la gestione delle pagine

Come spiegato nella sezione 3.2, l'applicazione è costituita da cinque pagine, ciascuna delle quali ricopre, nella logica del *software*, un ruolo ben preciso. Esse sono:

1. la pagina *Home*, che viene mostrata all'apertura dell'applicazione e funge da pagina principale;
2. la pagina *Places*, che presenta la lista con tutti i luoghi;
3. la pagina *Artifacts*, che presenta la lista con tutti gli artefatti;
4. la pagina *Place*, che elenca i dettagli di un preciso luogo (sebbene sembri omonima alla precedente, si noti la mancanza della “s” finale, ad indicare che questa pagina tratta un singolo luogo);
5. la pagina *Artifact*, che elenca i dettagli di uno specifico artefatto.

Per creare una pagina nel *framework* Ionic, viene utilizzato il seguente comando:

```
>_ ionic generate page [name]
```

in cui, al posto di `[name]`, viene specificato il nome da attribuire alla pagina stessa. Vale la pena notare che è proprio con il comando `ionic generate` che è possibile dar vita a tutti gli elementi che permettono di strutturare la logica del *software*. Tra questi elementi annoveriamo i componenti, le direttive, le *pipe*, i *provider* e i *tab*. Una volta eseguito il precedente comando, il *framework* Ionic genera una sotto cartella all'interno della directory `./src/pages/`, la quale risulterà avere un nome uguale a quello specificato

nel momento in cui è stato lanciato il comando. Naturalmente, è stato necessario eseguire tale istruzione cinque volte, affinché si potessero creare le pagine basilari:

```
>_ ionic generate page home
>_ ionic generate page places-list
>_ ionic generate page place
>_ ionic generate page artifact
>_ ionic generate page artifacts-list
```

A titolo esemplificativo è possibile generalizzare il discorso considerando una generica pagina definita, ad esempio, all'interno della cartella fittizia `./src/pages/example/` che supponiamo essere stata appena creata. Aprendo la *directory* sarebbe possibile notare tre *file*, che sono stati creati automaticamente dal *framework*:

1. un *file* denominato `example.html`, ossia un *file* di *markup*;
2. un *file* denominato `example.scss` (nello specifico, un *file* SASS, ossia un'estensione del linguaggio CSS che permette di utilizzare variabili, di creare funzioni e di organizzare il foglio di stile in più *file*);
3. un *file* denominato `example.ts`, ossia un *file* contenente codice sorgente, scritto in linguaggio TypeScript.

Non è un caso che, estensioni a parte, tutti e tre questi file presentino lo stesso nome, il quale viene direttamente ereditato dall'omonima *directory* a cui appartengono (per l'appunto, la cartella `./src/pages/example/`). Ognuno di essi, naturalmente, assolve un compito specifico:

1. il *file* `example.html` contiene il codice HTML che compone la pagina, in termini di semantica e struttura e che genera tutti quegli elementi che corrispondono alle varie sezioni visibili nella *view* relativa a quella specifica pagina;
2. il *file* `example.scss` contiene tutte le regole di stile che vengono applicate agli elementi definiti nel *file* precedente e relativi, quindi, alla pagina in sé;
3. il *file* `example.ts` costituisce il *file* principale, sulla cui elaborazione viene, senza dubbio, impiegata la maggior parte del tempo durante il processo di sviluppo, in quanto è proprio al suo interno che vengono definiti il comportamento della pagina, il ruolo di ogni pulsante e di ogni elemento grafico presente con cui è possibile interagire, quali azioni sono consentite dal *software*, qual'è l'effetto di ognuna di esse, etc.

È poi possibile notare, tra l'altro, una certa analogia tra la pagina, seppur fittizia, appena presa in esame (ma che rispecchia, comunque, le reali sembianze di una vera e propria pagina dell'applicazione) e il contenuto della cartella `./src/app/`, la quale contiene infatti un *file* HTML, un *file* SASS e un *file* TypeScript. Bisogna però specificare che, com'è stato visto, quest'ultima “pagina” svolge un ruolo molto importante e definito e che la cartella `./src/app/`, mediante i *file* che contiene, regola tutto il funzionamento generale dell'applicazione, compresa la scelta della pagina d'avvio, il caricamento di tutti i *plugin* e *provider*, etc.

Ritornando all'elenco dei tre *file* poc'anzi creati e citati, si riesce facilmente a capire come, mentre i primi due (quelli relativi al *markup* e allo stile, `example.html` e `example.scss`) definiscono la semantica e la struttura che è possibile stabilire per la pagina e sono, pertanto, piuttosto semplici in termini di codice, il file `example.ts` è, di sicuro, ben più complicato da comprendere e gestire. In una forma puramente e volutamente semplice, questo file risulta essere così composto:

```

1 // Sezione dedicata all'importazione di tutte le risorse
2 // esterne, come plugin e pagine
3 import { Component } from '@angular/core';
4 import { NavController } from 'ionic-angular';
5
6 // Sezione dedicata al decorator @Component, in cui si
7 // indicano il selettor e il template da usare
8 @Component({
9   selector: 'page-example',
10   templateUrl: 'example.html'
11 })
12
13 // Sezione costituita dalla dichiarazione della pagina
14 // in termini di definizione ed esportazione della classe
15 export class ExamplePage {
16   // ...
17 }
```

Com'è stato evidenziato già nel codice sorgente, il file può essenzialmente essere suddiviso in tre zone principali:

1. la parte superiore è costituita dagli enunciati di `import`, tramite cui è possibile, per l'appunto, importare tutti gli elementi, le funzionalità (*provider*), i componenti e le pagine di cui si fa uso all'interno del codice. Nell'esempio preso in esame, vengono importate le classi `Component` e `NavController`, appartenenti alle rispettive librerie `@angular/core` e `ionic-angular`, reperibili all'interno della cartella `./node_modules/`;

2. la parte centrale (che risulta essere estremamente ridotta e compatta) è costituita da un *decorator*, un elemento reperibile all'interno di ogni classe di un'applicazione Ionic. Il compito di questi elementi è quello di fornire dei metadati riguardo alla classe che viene definita. Essi si trovano immediatamente sopra la definizione della classe stessa. Come si può notare, il *decorator* prende in *input* un oggetto JavaScript, dotato di due proprietà:
  - (a) la proprietà **selector**, che indica, mediante il suo valore, il selettore o, per meglio dire, il *tag* HTML che bisogna utilizzare per “inglobare”, “incorporare”, “richiamare” la pagina;
  - (b) la proprietà **templateUrl**, che indica, sempre mediante il suo valore, il *file* da utilizzare come *template* e da cui ricavare quella che deve essere la struttura della pagina stessa. Si noti che se il *template* è sufficientemente semplice, è possibile evitare di utilizzare un *file* esterno, indicando direttamente il codice stesso come valore di questa proprietà.
3. la parte inferiore, quella di sicuro predominante, ospita la definizione della classe, completa di attributi, costruttore e metodi. Si noti che essa viene, al tempo stesso, esportata (**export class ...**), in modo tale da poter essere così successivamente importata e utilizzata anche in altri *file*.

## 5.4 La gestione della navigazione

Nel paragrafo 4.3 è stata affrontata la spiegazione di come funzioni la navigazione all'interno di un'applicazione Ionic. In questo paragrafo si riprende l'argomento, stavolta dal punto di vista del codice.

La classe basilare che coordina questo importante aspetto del *software* è denominata **NavController**. Di base, un *controller* di navigazione non è altro che un *array* di pagine che rappresenta una precisa cronologia. Tale *array* può essere manipolato per navigare all'interno di un'applicazione eseguendo operazioni di *push* e *pop*, o inserendo e rimuovendo le pagine stesse da locazioni arbitrarie nella cronologia. La pagina attuale corrisponde all'ultima presente nel vettore, ossia quella che si trova in cima allo *stack*. Effettuando un'operazione di *push* sulla cima dello *stack*, la nuova pagina viene messa in mostra mediante un'animazione, mentre un'operazione di *pop* fa sì che l'utente venga reindirizzato alla pagina precedente. Quando si tratta di un'applicazione Ionic, molto spesso è sufficiente utilizzare un riferimento al **NavController** più prossimo al fine di manipolare la pila.

Il modo più semplice per navigare all'interno di un'applicazione è quello di creare e inizializzare un nuovo *controller* di navigazione sfruttando il componente `<ion-nav>...</ion-nav>`. Iniettando un `NavController` tra i parametri formali di un costruttore si ottiene sempre un'istanza del *controller* più prossimo:

```

1 // Importazione di funzionalità e classi necessarie
2 /**
3  import { NavController } from 'ionic-angular';
4
5 // Definizione di una classe che intende sfruttare il
6 // NavController per modificare lo stack di navigazione
7 class MyComponent {
8   constructor(public navCtrl: NavController) { }
9 }
```

Le *view* sono create automaticamente quando vengono aggiunte allo *stack* di navigazione. Per effettuare un'operazione di inserimento di una *view* nello *stack* (quindi, per visualizzarla a schermo), si utilizza il metodo `push()`. Se la pagina in questione è dotata di un elemento `<ion-nav>...</ion-nav>`, viene automaticamente aggiunto un pulsante *back* alla nuova vista. Essa può ricevere poi dei dati, accedendo a questi ultimi mediante l'apposita classe  `NavParams`. Questa classe rappresenta un oggetto che esiste all'interno di una pagina e che può contenere dei dati per quella particolare *view*. In particolare,  `NavParams` si basa su un metodo molto semplice per effettuare questo tipo di operazioni, il `get()`.

## 5.5 L'installazione e l'importazione dei *plugin*

Nell'ecosistema Ionic i *plugin* costituiscono un elemento per eccellenza, in quanto capaci di rendere un'applicazione in grado di usufruire delle risorse e delle funzionalità del dispositivo su cui vengono installate ed eseguite. Sono strumenti fondamentali, senza i quali il *software* non potrebbe accedere alla fotocamera del *device*, al *file system*, ai contatti, allo stato della rete, etc. Com'è stato detto, l'applicazione si basa, principalmente, su due *plugin* specifici, ossia quello relativo ai *beacon* BLE e quello relativo ai QR *code*, sebbene lo sviluppo del *software* abbia previsto, però, soltanto la manuale installazione e utilizzo del secondo. Detto ciò, la gestione dei *plugin* avviene mediante un processo *standard*.

L'elenco dei *plugin* che sono stati installati all'interno della cartella del progetto e che sono, pertanto, disponibili all'importazione e all'utilizzo è consultabile all'interno della directory `./plugins/`, in cui sono presenti tante sotto cartelle, una per ogni componente aggiuntivo. Subito dopo aver creato,

mediante la Ionic CLI, un progetto Ionic, è possibile notare come, all'interno della suddetta *directory*, siano già presenti dei *plugin*, che non sono stati installati in modo manuale dallo sviluppatore. Tra di essi è possibile notare i seguenti:

1. `cordova-plugin-device`;
2. `cordova-plugin-ionic-keyboard`;
3. `cordova-plugin-ionic-webview`;
4. `cordova-plugin-splashscreen`;
5. `cordova-plugin-whitelist`.

Essi costituiscono un insieme di *plugin* di *default*, presenti da subito nella *directory* del progetto. Tutti gli altri *plugin* è stato necessario installarli manualmente, sempre sfruttando l'interfaccia da riga di comando.

### 5.5.1 Lo *scanner* dei codici QR

Il *framework* Ionic mette a disposizione il **QR Scanner**, un *plugin* che permette allo sviluppatore di aprire e utilizzare un vero e proprio *scanner* di codici QR. Esso è appositamente pensato per le applicazioni realizzate con Cordova, il quale risulta essere, inoltre, altamente configurabile, veloce ed efficiente dal punto di vista energetico (ossia attento all'utilizzo della batteria del dispositivo). Tale componente aggiuntivo richiede, per poter funzionare, il `cordova-plugin-qrscanner`, un componente del *framework* Cordova. Per poterlo installare, è stato sufficiente eseguire i seguenti comandi da terminale:

```
>_ ionic cordova plugin add cordova-plugin-qrscanner  
>_ npm install --save @ionic-native/qr-scanner
```

Fatto questo, l'unica cosa che restava da fare era quella di aggiungere il *plugin* al modulo dell'applicazione (`./src/app/app.module.ts`). Le piattaforme che questo componente supporta sono:

- il *browser*;
- il sistema operativo Android;
- il sistema operativo iOS;
- la piattaforma Windows.

Una volta installato, il *plugin* risulta essere presente, assieme a tutti gli altri, all'interno della *directory* `./plugins/`. Di seguito viene riportato il codice necessario per poter utilizzare il componente aggiuntivo:

```

1 // Importazione di funzionalità e classi necessarie
2 // ...
3 import { QRScanner, QRScannerStatus } from '@ionic-native/qr-scanner';
4
5 // Definizione ed esportazione della classe Example
6 export class Example {
7   constructor(private qrScanner: QRScanner) { }
8
9   // Metodo d'esempio che sfrutta lo scanner
10  useScanner() {
11    this.qrScanner.prepare().then((status: QRScannerStatus) => {
12      if (status.authorized) {
13        // È possibile effettuare la scannerizzazione
14        // utilizzando la fotocamera
15        let scanSub = this.qrScanner.scan().subscribe((text: string) => {
16          console.log('Scannerizzazione avvenuta');
17          this.qrScanner.hide(); // Nascondi preview della fotocamera
18          scanSub.unsubscribe(); // Ferma la scannerizzazione
19        });
20      } else if (status.denied) {
21        // Il permesso per usare la fotocamera è stato negato in modo
22        // permanente. È necessario utilizzare il metodo
23        // QRScanner.openSettings() per condurre l'utente alle pagine
24        // delle impostazioni in cui può, stavolta, garantire il permesso
25      } else {
26        // Il permesso della fotocamera è stato negato in modo non
27        // permanente. E' possibile chiedere nuovamente il permesso
28        // in un secondo momento
29      }
30    }).catch((e: any) => console.log('Errore: ', e));
31  }
32}

```

## 5.6 La pagina principale

La pagina principale è quella che viene visualizzata per prima quando si lancia l'applicazione. In figura 5.1 è riportato uno *screenshot* che mostra l'interfaccia grafica della *homepage* effettivamente realizzata:



**Figura 5.1:** Uno *screenshot* della pagina principale

Com'è stato spiegato, una pagina, nel senso largo del termine, è rappresentata da una cartella, al cui interno sono presenti sostanzialmente tre *file*. Nel caso della *homepage*, la cartella è stata denominata `home` e al suo interno si trovano i seguenti tre *file*:

1. `home.html`;
2. `home.scss`;
3. `home.ts`.

Per quanto concerne il file HTML, il codice è il seguente:

```
1 <ion-header>
2   <ion-navbar>
3     <ion-title>
4       inSight
5     </ion-title>
6   </ion-navbar>
7 </ion-header>
8 <ion-content>
9   <ion-grid>
10    <ion-row>
11      <h1>Scegli da dove cominciare</h1>
12      <button (click)="scanQrCode()">Scannerizza codice QR</button>
13      <button (click)="goToPlacesList()">Sfoglia i luoghi</button>
```

```

14      <button (click)="goToArtifactsList()">Sfoglia gli artefatti</button>
15    </ion-row>
16  </ion-grid>
17 </ion-content>

```

In esso sono state volutamente evitate alcune istruzioni e frammenti di righe di codice, al fine di renderlo il più snello e comprensibile possibile. Naturalmente, nulla è stato tolto che non fosse assolutamente superfluo per la realizzazione del *software*: si trattava, infatti, di pure istruzioni stilistiche. In ambito CSS, le istruzioni presenti all'interno del foglio di stile sono soltanto quelle relative alla griglia che Ionic fornisce:

```

1 ion-grid {
2   height: 100%;
3   justify-content: center;
4   text-align: center;
5 }

```

Infine, il *file* più interessante, ossia quello TypeScript. Il codice riportato in questo *file* è il seguente:

```

1 // Importazione di funzionalità e classi necessarie
2 import { Component } from '@angular/core';
3 import { NavController } from 'ionic-angular';
4 import { DataManagerProvider } from
  ↪ '../providers/data-manager/data-manager';
5 import { PlacesListPage } from '../places-list/places-list';
6 import { ArtifactsListPage } from "../artifacts-list/artifacts-list";
7 import { BarcodeScanner } from '@ionic-native/barcode-scanner';
8 import { PlacePage } from '../place/place';
9 import { ArtifactPage } from "../artifact/artifact";
10
11 // Decoratore
12 @Component({
13   selector: 'page-home',
14   templateUrl: 'home.html'
15 })
16
17 // Definizione ed esportazione della classe
18 export class HomePage {
19   // Costruttore, in cui si iniettano le classi esterne usate
20   constructor(public navCtrl: NavController, private dataManager:
  ↪ DataManagerProvider, private barcodeScanner: BarcodeScanner) { }
21
22   // Funzione invocata al caricamento della view
23   ionViewDidLoad() {
24     this.dataManager.load();
25   }

```

```

26
27 // Funzione che gestisce la scannerizzazione di un codice QR
28 scanQrCode() {
29     this.barcodeScanner.scan().then(barcodeData => {
30         var qrHiddenInfo = JSON.parse(barcodeData.text);
31         var type = qrHiddenInfo.type;
32         let item = this.dataManager.findItemWithQR(barcodeData);
33         if (type === "place") {
34             this.navCtrl.push(PlacePage, {
35                 name: item.name,
36                 thumbnail: item.thumbnail,
37                 img: item.img,
38                 description: item.description,
39                 artifacts: item.artifacts,
40                 architect: item.architect
41             });
42         } else {
43             this.navCtrl.push(ArtifactPage, {
44                 img: item.img,
45                 name: item.name,
46                 type: item.type,
47                 date: item.date,
48                 description: item.description,
49                 author: item.author
50             });
51         }
52     }).catch(err => {
53         console.log("Si è verificato un errore: ", err);
54     });
55 }
56
57 // Funzione che modifica lo stack di navigazione,
58 // visualizzando la pagina PlacesListPage
59 goToPlacesList() {
60     this.navCtrl.push(PlacesListPage);
61 }
62
63 // Funzione che modifica lo stack di navigazione,
64 // visualizzando la pagina ArtifactsListPage
65 goToArtifactsList() {
66     this.navCtrl.push(ArtifactsListPage);
67 }
68 }
```

È lecito, a questo punto, domandarsi perché sia proprio questa pagina ad essere visualizzata per prima, ossia per quale motivo l'applicazione sa che, all'apertura, deve essere mostrata proprio la *homepage*. La risposta a questo quesito giace tra le righe di codice che si trovano all'interno del file `./src/app/app.component.ts`, tra le quali è infatti possibile notare il seguente frammento di codice:

```

export class MyApp {
  rootPage:any = HomePage;
  // ...
}

```

Il valore dell'attributo `rootPage`, appartenente alla classe `MyApp`, è stato posto uguale a `HomePage`. In tal modo si indica che è proprio lei a rivestire il ruolo di pagina principale dell'applicazione. Tale attributo è fornito dalla classe `NavController`, la quale è stata adeguatamente importata e inserita nel costruttore.

Nella definizione della classe sono stati definiti **tre metodi** (oltre al classico costruttore e alla funzione `ionViewDidLoad()`, lanciata automaticamente quando la *view* termina il suo caricamento):

1. `scanQrCode()`, il cui compito è quello di aprire la fotocamera del *device*, gestire l'acquisizione di un codice QR e mostrare, infine, l'apposita pagina del luogo o dell'artefatto;
2. `goToPlacesList()`, il cui compito è quello di indirizzare l'utente alla pagina che elenca tutti i luoghi possibili;
3. `goToArtifactsList()`, che mostra la pagina contenente l'elenco di tutti gli artefatti.

È opportuna soffermarsi sul primo dei tre metodi elencati. Come detto, il codice scritto al suo interno gestisce tutto il processo di apertura della fotocamera, di acquisizione di un codice e di navigazione verso la pagina appropriata. Si nota da subito l'utilizzo delle *promise* e degli *observable*. Il metodo esegue le seguenti operazioni:

1. acquisito un codice, viene invocata la funzione `JSON.parse` per analizzare la stringa in *input*, controllare l'eventuale *match* con un oggetto JavaScript *e*, in caso di esito positivo, restituire l'oggetto stesso. La stringa in questione è proprio quella nascosta all'interno del codice QR (ecco perché essa è stata formattata così come spiegato nella sezione 4.5);
2. memorizza in una variabile il valore della proprietà `type` dell'oggetto restituito;
3. lancia la funzione `findItemWithQR()` del provider `dataManager`, che effettua una ricerca del luogo mediante la proprietà `place` del codice QR: se il codice rappresenta proprio un luogo, viene restituito soltanto l'oggetto che identifica lo stesso altrimenti, se rappresenta un artefatto, viene restituito un oggetto che identifica l'artefatto in questione (in altre parole, viene letteralmente restituito l'oggetto JavaScript);

4. se il codice è quello relativo a un luogo (controllo che viene eseguito nella guardia del `if`, analizzando la proprietà `type`) il metodo modifica lo *stack* di navigazione e visualizza l'apposita pagina di descrizione del luogo, passando in *input* un oggetto che serve al `NavController` e nelle cui proprietà vengono memorizzati i dati che servono per popolare la nuova pagina;
5. se il codice è relativo, invece, a un artefatto, il metodo modifica sempre lo *stack* di navigazione e porta l'utente verso la *view* di descrizione dell'oggetto, passando ancora in *input* un oggetto che serve al `NavController` e nelle cui proprietà vengono memorizzati i dati che servono per la pagina di destinazione;
6. se la scannerizzazione non dovesse andare a buon fine, allora in tal caso viene restituito un errore.

Come si nota, i metodi appartenenti alla classe della pagina principale fanno un pesante uso delle funzioni definite all'interno della classe del *provider*.

## 5.7 La pagina della lista dei luoghi

Partendo dalla *homepage* è possibile, mediante l'apposito pulsante, visualizzare la *view* contenente la lista di tutti i luoghi che sono disponibili alla consultazione. Eseguendo un *tap* su una voce presente in questo elenco, l'utente viene indirizzato alla pagina che racchiude la descrizione e le risorse disponibili per il singolo luogo, compresa la lista di tutti gli artefatti che sono ivi custoditi. In figura 5.2 è riportato uno *screenshot* che mostra l'interfaccia grafica della pagina così come compare realmente all'interno dell'applicazione:



**Figura 5.2:** Uno screenshot della pagina dei luoghi

Come per la *homepage*, anche questa pagina è rappresentata da una cartella, denominata `places-list`, contenente i soliti tre file:

1. `places-list.html`;
2. `places-list.scss`;
3. `places-list.ts`.

Per quanto concerne il file HTML, il suo contenuto è il seguente:

```

1 <ion-header>
2   <ion-navbar>
3     <ion-title>Luoghi</ion-title>
4   </ion-navbar>
5 </ion-header>
6 <ion-content>
7   <ion-card *ngFor="let place of dataManager.places"
8     (click)="goToPlacePage(place)">
9     
10    <ion-card-content>
11      <ion-card-title>{{ place.name }}</ion-card-title>
12      <p>{{ place.description }}</p>
13    </ion-card-content>
14  </ion-card>
</ion-content>

```

Nella sostanza, il contenuto principale della pagina viene creato mediante un'apposita direttiva Angular, `*ngFor`, la quale permette di **iterare** su una collezione di oggetti, ripetendo un certo frammento di codice. La collezione, in questo caso, è costituita dall'*array* contenente tutti i luoghi, memorizzato all'interno della classe che rappresenta il *provider*, `DataManagerProvider`. La direttiva permette di riprodurre l'elemento grafico `ion-card` (fornito da Ionic) sulla base di quanti elementi sono presenti all'interno del vettore. A ognuno di essi, poi, viene associato un evento, ossia il *click*, mediante l'apposita istruzione `(click) = "..."`, che indirizza l'utente alla pagina del singolo luogo. Sono presenti, quindi, un *tag* `img` che incorpora un'immagine il cui URL viene ottenuto sfruttando la proprietà `thumbnail` del luogo, e un *tag* `ion-card-content` che riporta il nome e una breve descrizione. Il file CSS è vuoto, in quanto non è stato necessario specificare alcuna regola di stile specifica. Il *file* TypeScript, infine, contiene il seguente codice:

```

1 // Importazione di funzionalità e classi necessarie
2 import { Component } from '@angular/core';
3 import { IonicPage, NavController, NavParams } from 'ionic-angular';
4 import { DataManagerProvider } from
5   './providers/data-manager/data-manager';
6 import { PlacePage } from '../place/place';
7
8 // Decoratori
9 @IonicPage()
10 @Component({
11   selector: 'page-places-list',
12   templateUrl: 'places-list.html',
13 })
14
15 // Definizione ed esportazione della classe
16 export class PlacesListPage {
17   // Costruttore
18   constructor(public navCtrl: NavController, public navParams: NavParams,
19   public dataManager: DataManagerProvider) { }
20
21   // Funzione che modifica lo stack di navigazione,
22   // visualizzando la pagina PlacePage
23   goToPlacePage(place) {
24     this.navCtrl.push(PlacePage, {
25       name: place.name,
26       thumbnail: place.thumbnail,
27       img: place.img,
28       description: place.description,
29       artifacts: place.artifacts,
30       architect: place.architect
31     });
32   }
33 }
```

Come si vede è presente un solo metodo, `goToPlacePage()`, il quale indirizza l'utente verso la pagina `PlacePage`, ossia quella pagina che riporta la descrizione di uno specifico luogo. Il metodo viene invocato in seguito alla selezione di un *record* dalla lista, per mezzo del legame con l'istruzione `(click) = "..."` presente nel file HTML. Il metodo non fa altro che invocare la funzione `push()` del gestore della navigazione passandogli in *input* un oggetto contenente alcune proprietà che permettono di passare le informazioni necessarie alla pagina successiva.

## 5.8 La pagina che descrive un singolo luogo

Questa pagina riporta tutte le informazioni che appartengono a uno specifico luogo, dati che sono memorizzati all'interno dell'oggetto che rappresenta il luogo stesso. Tra questi dati ne sono stati elencati alcuni, come il **nome** del luogo, una breve **descrizione**, l'**architetto**, nonché naturalmente una lista completa di tutti gli **artefatti** che si possono trovare al suo interno. In figura 5.3 è riportato uno *screenshot* che mostra la *user interface* della pagina così come compare realmente all'interno dell'applicazione:



**Figura 5.3:** Uno *screenshot* della pagina di un luogo

La pagina è rappresentata, all'interno della *directory* di progetto, dalla cartella `./src/pages/place/`, al cui interno sono memorizzati i seguenti

*file:*

1. place.html;
2. place.css;
3. place.ts.

Questa pagina può essere visualizzata secondo due modalità differenti, ossia o mediante la scannerizzazione del codice QR, oppure mediante il *tap* sul *record* presente nella pagina della lista dei luoghi. Il codice HTML è così composto:

```
1 <ion-header>
2   <ion-navbar>
3     <ion-title>{{ name }}</ion-title>
4   </ion-navbar>
5 </ion-header>
6 <ion-content padding>
7   
16      <p>{{ artifact.name }}</p>
17    </ion-item>
18  </ion-list>
19 </ion-content>
```

Come si legge, sono state riportate tutte le informazioni e i dati relativi al luogo. Da notare poi l'utilizzo di un componente Ionic per creare l'elenco degli artefatti. Questo elenco è stato realizzato mediante il componente **ion-list** che crea, per l'appunto, una **lista**, i cui singoli *record* sono stati renderizzati mediante l'utilizzo della direttiva Angular **\*ngFor**, che itera su tutti gli elementi presenti nel vettore degli artefatti del luogo stesso. Da notare la differenza con la pagina della lista dei luoghi: in quel caso, non si era deciso di optare per il componente **ion-list**, in quanto il componente **ion-card** forniva un'estetica decisamente più accattivante, adatta a elencare i singoli luoghi e riportando, anche, un'immagine. Nel caso degli artefatti, l'immagine non è stata necessaria e quindi l'utilizzo di una semplice lista è risultato più che sufficiente. Naturalmente è ancora presente il legame con l'evento **(click) = "..."** che, in questo caso, invoca la funzione **goToPlacePage()**. Anche in questo caso, non è stato necessario scrivere precise regole di stile, pertanto il *file place.css* risulta essere, anche in questo caso, vuoto. Il *file* TypeScript, infine, contiene il seguente *snippet*:

```

1 // Importazione di funzionalità e classi necessarie
2 import { Component } from '@angular/core';
3 import { IonicPage, NavController, NavParams } from 'ionic-angular';
4 import { DataManagerProvider } from
5   ↪  '../providers/data-manager/data-manager';
6 import { ArtifactPage } from "../artifact/artifact";
7
8 // Decoratori
9 @IonicPage()
10 @Component({
11   selector: 'page-place',
12   templateUrl: 'place.html',
13 })
14
15 // Definizione ed esportazione della classe
16 export class PlacePage {
17
18   // Variabili che memorizzano i dati di un luogo
19   name: string;
20   thumbnail: string;
21   img: string;
22   description: string;
23   artifacts: object[];
24   architect: string;
25
26   constructor(public navCtrl: NavController, public navParams: NavParams,
27     ↪  public dataManager: DataManagerProvider) {
28     this.name = navParams.get("name");
29     this.thumbnail = navParams.get("thumbnail");
30     this.img = navParams.get("img");
31     this.description = navParams.get("description");
32     this.artifacts = navParams.get("artifacts");
33     this.architect = navParams.get("architect");
34   }
35
36   // Funzione che modifica lo stack di navigazione,
37   // visualizzando la pagina ArtifactPage
38   goToArtifactPage(artifact) {
39     this.navCtrl.push(ArtifactPage, {
40       img: artifact.img,
41       name: artifact.name,
42       type: artifact.type,
43       date: artifact.date,
44       description: artifact.description,
45       author: artifact.author
46     });
47   }
48 }

```

Da notare l'utilizzo dei vari metodi `get()` per memorizzare tutte le informazioni.

zioni provenienti dalla pagina precedente, le quali vengono salvate in apposite variabili locali. È presente un solo metodo, `goToArtifactPage()`, che invoca la funzione `push()` passando, in *input*, la `ArtifactPage`, ossia la pagina di un artefatto, assieme al solito oggetto contenente i dati di quell'elemento, dati ricavati dall'apposito oggetto che rappresenta il luogo.

## 5.9 La pagina che elenca gli artefatti

Sempre partendo dalla *homepage* è possibile, sempre mediante un apposito pulsante, visualizzare la *view* che presenta la lista di tutti gli artefatti che sono disponibili alla consultazione. Eseguendo un *tap* su una voce presente in questo elenco, l'utente viene indirizzato alla pagina che racchiude la descrizione e le risorse disponibili per il singolo elemento. In figura 5.4 è riportato uno *screenshot* che mostra l'interfaccia grafica della pagina, così come compare realmente all'interno dell'applicazione:



**Figura 5.4:** Uno *screenshot* della pagina degli artefatti

La pagina è rappresentata, all'interno della *directory* di progetto, dalla cartella `./src/pages/artifacts-list/`, al cui interno sono memorizzati i seguenti *file*:

1. `artifacts-list.html`;

2. artifacts-list.css;

3. artifacts-list.ts.

Il file HTML è così composto:

```
1 <ion-header>
2   <ion-navbar>
3     <ion-title>Artefatti</ion-title>
4   </ion-navbar>
5 </ion-header>
6 <ion-content>
7   <ion-list *ngFor="let place of dataManager.places">
8     <h5 padding-left>{{ place.name }}</h5>
9     <ion-card *ngFor="let artifact of place.artifacts"
10       <click>="goToArtifactPage(artifact)">
11         
12         <ion-card-content>
13           <ion-card-title>{{ artifact.name }}</ion-card-title>
14           <p>{{ artifact.description }}</p>
15         </ion-card-content>
16       </ion-card>
17     </ion-list>
18 </ion-content>
```

Per poter iterare sulla collezione di tutti gli artefatti, ed essendo essi memorizzati in un *array* che è una proprietà dell'oggetto che rappresenta un luogo, è stata utilizzata una lista, *ion-list*, che itera sui luoghi della collezione; per ogni luogo, poi, è stato iterato il componente *ion-card* per il numero di artefatti presenti. In questo modo è stata ottenuta la lista di tutti gli elementi consultabili. Sono state sfruttate, a questo scopo, due direttive *\*ngFor*, in cui il ciclo esterno memorizza un luogo, mentre quello interno itera sulla collezione degli elementi che custodisce. Si nota, come sempre, l'utilizzo del *binding* di Angular per riportare, sull'interfaccia grafica, i valori delle proprietà (che rappresentano i dati), memorizzate all'interno dell'apposita classe.

Anche in questo caso, il file CSS si è rivelato superfluo, in quanto non è stato necessario specificare alcune regole di stile aggiuntiva. Per quanto concerne il file TypeScript, invece, esso contiene il seguente codice:

```
1 // Importazione di funzionalità e classi necessarie
2 import { Component } from '@angular/core';
3 import { IonicPage, NavController, NavParams } from 'ionic-angular';
4 import { DataManagerProvider } from
5   '../providers/data-manager/data-manager';
6 import { ArtifactPage } from '../artifact/artifact';
7
8 // Decoratori
```

```

8  @IonicPage()
9  @Component({
10    selector: 'page-artifacts-list',
11    templateUrl: 'artifacts-list.html',
12  })
13
14  // Definizione ed esportazione della classe
15  export class ArtifactsListPage {
16    // Costruttore
17    constructor(public navCtrl: NavController, public navParams: NavParams,
18      public dataManager: DataManagerProvider) { }
19
20    // Funzione che modifica lo stack di navigazione,
21    // visualizzando la pagina ArtifactPage
22    goToArtifactPage(artifact) {
23      this.navCtrl.push(ArtifactPage, {
24        img: artifact.img,
25        name: artifact.name,
26        type: artifact.type,
27        date: artifact.date,
28        description: artifact.description,
29        author: artifact.author
30      });
31    }

```

Il funzionamento è molto simile alla `PlaceslistPage`, con la presenza di un solo metodo che permette di navigare direttamente alla pagina dell'artefatto in questione, sempre utilizzando un oggetto che viene passato al metodo `push()` del `NavController`.

## 5.10 La pagina che descrive un artefatto

Il ruolo svolto da questa pagina è unicamente quello di mostrare all'utente tutte le informazioni relative a un preciso artefatto. In figura 5.5 è riportato uno *screenshot* che mostra l'interfaccia grafica della pagina, così come compare realmente all'interno dell'applicazione:



**Figura 5.5:** Uno *screenshot* della pagina di un artefatto

Quest’ultima pagina è rappresentata, sempre all’interno della *directory* di progetto, dalla cartella `./src/pages/artifacts/`, al cui interno sono memorizzati i seguenti *file*:

1. `artifact.html`;
2. `artifact.scss`;
3. `artifact.ts`.

Lo scopo di questa pagina è quello di elencare tutte le caratteristiche di un artefatto, informazioni che sono memorizzate all’interno dell’oggetto che rappresenta l’artefatto stesso. Così come per la pagina di un luogo, anche in questo caso, tra questi dati, ne sono stati elencati alcuni, come il **nome** dell’oggetto, una sua **immagine**, la sua **tipologia**, la sua **data di realizzazione**, il suo **autore** e **descrizione**. Questa pagina è raggiungibile, proprio come nel caso di un luogo, mediante la scannerizzazine dell’apposito codice QR, mediante la selezione della voce relative a tale artefatto, presente nella pagina che riporta la lista di tutti gli artefatti oppure mediante la selezione dalla lista degli artefatti presente in un determinato luogo. Per quanto concerne il codice HTML, esso è il seguente:

```

1 <ion-header>
2   <ion-navbar>

```

```

3      <ion-title>{{ name }}</ion-title>
4    </ion-navbar>
5  </ion-header>
6  <ion-content padding>
7    
8    <h5>Tipo</h5>
9    <p>{{ type }}</p>
10   <h5>Data</h5>
11   <p>{{ date }}</p>
12   <h5>Autore/i</h5>
13   <p>{{ author }}</p>
14   <h5>Descrizione</h5>
15   <p>{{ description }}</p>
16 </ion-content>

```

Si nota la totale mancanza, in quanto non necessaria, di alcun ciclo, come invece era accaduto per diverse altre pagine. Anche in questo caso, è stato effettuato un semplice uso del *data binding* di Angular. Il file CSS relativo a questa pagina è vuoto, dal momento che non è stato necessario definire alcune regole di stile specifica. Infine, il file TypeScript è così composto:

```

1 // Importazione di funzionalità e classi necessarie
2 import { Component } from '@angular/core';
3 import { IonicPage, NavController, NavParams } from 'ionic-angular';
4
5 // Decoratori
6 @IonicPage()
7 @Component({
8   selector: 'page-artifact',
9   templateUrl: 'artifact.html',
10 })
11
12 // Definizione ed esportazione della classe
13 export class ArtifactPage {
14   // Variabili che memorizzano i dati di un artefatto
15   img: string;
16   name: string;
17   type: string;
18   date: string;
19   description: string;
20   author: string;
21
22   // Costruttore
23   constructor(public navCtrl: NavController, public navParams: NavParams) {
24     {
25       this.img = navParams.get("img");
26       this.name = navParams.get("name");
27       this.type = navParams.get("type");
28       this.date = navParams.get("date");
29     }
30   }
31 }

```

```

28     this.description = navParams.get("description");
29     this.author = navParams.get("author");
30   }
31 }
```

Sono state utilizzate, naturalmente, delle variabili che potessero memorizzare le informazioni di un determinato artefatto. Queste variabili, inizialmente, non hanno un valore preciso, in quanto esso viene determinato a *runtime* in base all'artefatto che si sceglie di analizzare. I dati vengono recuperati mediante la solita funzione `navParams.get()`, che permette di memorizzare i valori che sono stati passati da una pagina precedente, all'interno delle variabili poc'anzi citate.

## 5.11 Il *provider*

Come spiegato nel paragrafo 4.6, per gestire la lettura dei dati e delle informazioni a cui l'applicazione deve poter fare ricorso è stato utilizzato un *provider*. È stato anche spiegato che il codice che regola il funzionamento del provider è custodito all'interno di un unico file, `data-manager.ts`, localizzato all'interno della cartella `./src/providers/data-manager/`. Il codice relativo a questo fornitore è il seguente:

```

1 // Importazione delle risorse necessarie
2 import { Injectable } from '@angular/core';
3
4 // Definizione ed esportazione della classe
5 @Injectable()
6 export class DataManagerProvider {
7
8   // Array che conterrà i dati, inizialmente vuoto
9   private places = [];
10
11  constructor() { }
12
13  // Metodo per caricare i dati sul vettore all'avvio dell'applicazione
14  public load() {
15    this.places = [
16      {
17        name: "Aula Baratto",
18        thumbnail: "https://bit.ly/2onobeB",
19        img: "https://bit.ly/2Nql3JH",
20        description: "L'aula Baratto si trova al secondo piano nobile di
21          → Ca' Foscari, sede dell'Università Ca' Foscari Venezia...",
22        artifacts: [
23          {
24            img: "https://bit.ly/2BUFPzM",
25          }
26        ]
27      }
28    ];
29  }
30}
```

```

24         name: "Venezia, l'Italia e gli Studi",
25         type: "Affresco",
26         date: "1935-1936",
27         description: "L'affresco contiene una serie di figure
28             ↪ allegoriche che raccontano...",
29         author: "Mario Sironi"
30     },
31     {
32         img: "https://bit.ly/2Nud924",
33         name: "La Scuola",
34         type: "Affresco",
35         date: "1936",
36         description: "Nell'affresco si affronta il tema della \"scuola
37             ↪ dei filosofi\"...",
38         author: "Mario Deluigi"
39     }
40 ],
41 architect: "Carlo Scarpa"
42 },
43 {
44     name: "Il Cortile della Niobe",
45     thumbnail: "https://bit.ly/2LzwyN8",
46     img: "https://bit.ly/2LzwyN8",
47     description: "Ospita il complesso monumentale della Niobe, uno dei
48         ↪ luoghi più significativi...",
49     artifacts: [
50         {
51             img: "https://bit.ly/2MPpbWH",
52             name: "Scultura della Niobe",
53             type: "Scultura",
54             date: "1946",
55             description: "Una statua fontana di marmo realizzata dallo
56                 ↪ scultore Muranese...",
57             author: "Napoleone Martinuzzi"
58         }
59     ],
60     architect: "Bartolomeo e Giovanni Bon"
61 }
62 ];
63 }
64
65 public findItemWithQR(item) {
66     var foundPlace = 0, foundArtifact = 0;
67     var qrHiddenInfo = JSON.parse(item.text);
68
69     // Cerco il luogo, mediante la proprietà "place" del codice QR
70     for (let place of this.places) {
71         if (!foundPlace) {
72             if (place.name === qrHiddenInfo.place) {
73                 foundPlace = 1;
74                 // Se il QR rappresenta proprio un luogo, restituisco solo
75                     ↪ l'oggetto che identifica il luogo stesso...
76                 if (qrHiddenInfo.type === "place") {
77

```

```

72         return place;
73     } else { // ... altrimenti, se il QR rappresenta un artefatto,
    ↵      restituisco un oggetto che identifica l'artefatto stesso
74         for (let artifact of place.artifacts) {
75             if (!foundArtifact) {
76                 if (artifact.name === qrHiddenInfo.name) {
77                     return artifact;
78                 }
79             }
80         }
81     }
82   }
83 }
84 }
85 }
86
87 }

```

Come si vede, all'interno della classe che definisce il *provider* sono presenti **tre elementi** fondamentali:

1. la variabile `places`, che ha il compito di custodire tutti i dati relativi ai luoghi e agli artefatti. A dire il vero, il nome sembrerebbe indicare che essa riguardi solo ai luoghi, ma invece non è così e sono presenti anche le informazioni dei singoli elementi, informazioni che sono semplicemente custodite all'interno dell'oggetto che rappresenta un luogo (dopotutto, un artefatto appartiene a un luogo, quindi è possibile vederlo di diritto come una sua proprietà);
2. il metodo `load()`, che effettua semplicemente un aggiornamento del vettore `places` salvando i dati all'interno della variabile che lo rappresenta. Tale metodo viene invocato all'avvio dell'applicazione: nel codice della classe `HomePage`, infatti, il corpo di `ionViewDidLoad()`, invocato automaticamente da Ionic quando la pagina è stata caricata, è costituito da un'unica istruzione, `this.dataManager.load()`. Per mezzo di questa istruzione, viene invocato il metodo `load()` del *provider*, per mezzo del suo precedente inserimento tra i parametri formali del costruttore della classe `HomePage` stessa;
3. il metodo `findItemWithQR()`, che ricerca le informazioni relative a un luogo o a un artefatto mediante il codice QR acquisito attraverso la fotocamera. Questo metodo viene invocato quando si apre lo *scanner* dei codici QR mediante il pulsante presente nella *homepage*, al cui click è associata la funzione `scanQrCode()` la quale, al suo interno, riporta l'istruzione `this.barcodeScanner.scan()`. Anche in questo caso, la risorsa relativa a `barcodeScanner` è disponibile grazie

al suo inserimento tra i parametri formali del costruttore della classe `HomePage`.

Alla luce di tutto ciò, sarebbe lecito domandarsi perché sia stato implementato un metodo il cui unico obiettivo è quello di aggiornare **manualmente** i dati di una variabile privata del *provider*. La risposta è che una tale implementazione risulta essere perfetta per eventuali aggiornamenti futuri. Si è ribadito più volte come l'applicazione funzioni in modalità *offline* e che è stato deciso di memorizzare i dati fisicamente all'interno del *device*. Secondo questa metodologia di sviluppo, il metodo scritto funziona perfettamente. Se però, un domani, si volesse optare per una lettura dei dati provenienti un *server* remoto, sarebbe possibile, senza alcuna modifica aggiuntiva, editare il codice all'interno del metodo per far sì che i dati vengano letti proprio dal *server*. In altre parole, il metodo porterebbe a termine la stessa, identica funzione, con la sola differenza che ora i dati sono al momento *hard-coded*, mentre in futuro potrebbero essere ricavati da remoto.

# Capitolo 6

## Conclusioni

Il progetto del tirocinio ha richiesto la realizzazione di un'applicazione *mobile* ibrida, ossia progettata partendo da un unico *codebase*, costituito da codice HTML, CSS e JavaScript, poi compilato per creare due versioni differenti del *software*, installabili sui due principali sistemi operativi *mobile* presenti sul mercato, iOS e Android. Il funzionamento del programma è stato basato su due tecnologie specifiche: i codici QR e i *beacon* Bluetooth. A tale scopo, sono stati utilizzati dei *framework* specializzati nella realizzazione di questo tipo di *software*. Il *framework* principale è stato Ionic, il quale si concentra soprattutto sul *look and feel* del programma, basando la sua logica di funzionamento su un altro *framework*, Angular. Al fine di poter creare una versione del *software* che fosse installabile su un *device*, si è fatto affidamento ad un terzo *framework*, Cordova, il quale permette di creare un contenitore nativo che racchiude il programma, rendendolo a tutti gli effetti un'applicazione. Prima di passare alla stesura vera e propria del codice, è stato necessario passare per una fase di *wireframing*: utilizzando il *software* Adobe XD, è stato creato il prototipo dell'applicazione, ideando cinque *view* differenti, una per ognuna delle cinque pagine tra cui è possibile navigare all'interno dell'applicazione. Il passo successivo, nonché quello finale, è stato scrivere il codice che gestisce la logica e l'interfaccia grafica dell'applicativo. L'integrazione delle tecnologie sopra citate è stata portata a termine sfruttando appositi *plugin* di terze parti, che hanno permesso di accedere alle funzionalità dei dispositivi su cui è stata installata l'applicazione.

# Sitografia

**Sito web ufficiale di Ionic**

<https://ionicframework.com/>

**Sito web ufficiale di Angular**

<https://angular.io/>

**Sito web ufficiale di Cordova**

<https://cordova.apache.org/>

**Sito web ufficiale di Android Studio**

<https://developer.android.com/studio/>

**Sito web ufficiale di Xcode**

<https://developer.apple.com/xcode/>

***Bluetooth low energy beacon***

[https://en.wikipedia.org/wiki/Bluetooth\\_low\\_energy\\_beacon](https://en.wikipedia.org/wiki/Bluetooth_low_energy_beacon)

**QR code**

[https://it.wikipedia.org/wiki/Codice\\_QR](https://it.wikipedia.org/wiki/Codice_QR)

***Radio-frequency identification***

[https://it.wikipedia.org/wiki/Radio-frequency\\_identification](https://it.wikipedia.org/wiki/Radio-frequency_identification)

***Near Field Communication***

[https://it.wikipedia.org/wiki/Near\\_Field\\_Communication](https://it.wikipedia.org/wiki/Near_Field_Communication)

# Bibliografia

- [1] Smolenaers, F., Chestney, T., Walsh, J., Mathieson, S., Thompson, D., Gurkan, M., Marshall, S. *User centred development of a smartphone application for wayfinding in a complex hospital environment.* (2019) *Advances in Intelligent Systems and Computing*, 818, pp. 383-393.
- [2] Choi, M., Park, W.-K., Lee, I. *Smart office energy management system using bluetooth low energy based beacons and a mobile app.* (2015) 2015 IEEE International Conference on Consumer Electronics, ICCE 2015, art. no. 7066499, pp. 501-502.
- [3] Tsai, T.-H., Shen, C.-Y., Lin, Z.-S., Liu, H.-R., Chiou, W.-K. *Exploring location-based augmented reality experience in museums.* (2017) *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10278 LNCS, pp. 199-209.
- [4] Fino, E.R., Martín-Gutiérrez, J., Fernández, M.D.M., Davara, E.A. *Interactive tourist guide: Connecting web 2.0, augmented reality and QR codes.* (2013) *Procedia Computer Science*, 25, pp. 338-344.
- [5] Liu, C.-H., Lee, C.-F. *The design of a mobile navigation system based on QR codes for historic buildings.* (2009) 2009 TAIWAN CAA-DRIA: Between Man and Machine-Integration, Intuition, Intelligence-Proceedings of the 14th Conference on Computer-Aided Architectural Design Research in Asia, pp. 103-112.