POLITECNICO
MILANO 1863

Smart Tourist

Design and Implementation of Mobile Applications
Design Document

Fabio Codiglioni, Alessandro Nichelini

A.Y. 2019/2020

Contents

1 Introduction

This is the *design document* (DD) of the ***Smart Tourist*** iOS application developed by *Fabio Codiglioni* and *Alessandro Nichelini* in the context of the *Design and Implementation of Mobile Application* course at Politecnico di Milano. The authors have been tutored by *Bending Spoons* for the usage of some technologies that are described later. The document explains the most important design choices we made and the motivations behind them, with specific focus on the Redux-like architecture adopted.

2 General Overview

SmartTourist is a multi-device application for iOS, iPadOS and watchOS.

2.1 Assignment

For the reader convenience, here is the original assignment.

- Help your users when they are exploring new cities!
- By using the localization systems on the smartphones and the Google Places API, notify the user when something interesting is nearby.
- Connect to Wikipedia to retrieve interesting information on the selected point of interest.
- Search and filter among several interesting places of the city you are visiting, save them as favorites so you can check on them later.
- Attach some pictures to those pins and eventually share them with your friends!

2.2 Features description

2.2.1 Exploring

The user is given a navigation map filled with the city's points of interest. They have the chance to control which attractions to see on a map. They can choose between *nearest places*, *popular places* and *favorites*. Each attraction is shown on the map and in a list view, and can be tapped from both positions to open the corresponding detail view.

In the map, two circles are displayed to help the user better understand distances. These circles correspond to the distance they can cover in 5 and 15 minutes. Circles' radius is automatically updated with information from user profile and thus they always reflect the user's pace.

2.2.2 Dynamic exploring

SmartTourist works well in both crowded cities and small towns. The user will be always provided with the right amount of attractions. In places where not so many attractions are available, the app will automatically show less known attraction and particular point of interest.

2.2.3 Learning

The user can open the detail view of both cities and attractions. They will be provided with useful information about the point of interest, such as pictures, the Wikipedia description, useful links and a shortcut to open turn-by-turn navigation.

2.2.4 Notifications

The user is notified when they are nearby a "top location". The notification comes with a picture of the attraction and lets them open either the the detail view of the attraction or the turn-by-turn navigation.

2.2.5 Keep track of favorite attractions

The user can view the attractions of her current city or they can have a preview of other cities attractions. In both case they can keep track of them by adding them to the list of favorite attractions. The user is also provided the ability to open a worldwide map with all their favorite places.

2.2.6 Contribute (proof of concept)

SmartTourist is manly based on free data. When a detailed description is missing, the user is given the chance to contribute.

3 Architectural design

SmartTourist is a self-contained application. No external backend services are provided by us: instead, it relies on a multitude of internal Apple services and third-party APIs.

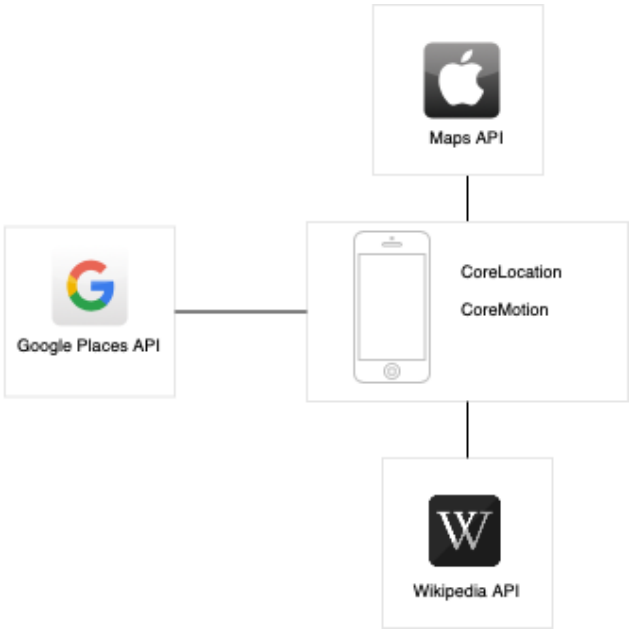

Figure 3.1: Architecture overview

Internal application design is strongly influenced by the choice of the Katana/Tempura framework that forced the adoption of the MVVM paradigm. Moreover, every piece of asynchronous code has been handled with Hydra, as suggested by Katana's documentation.

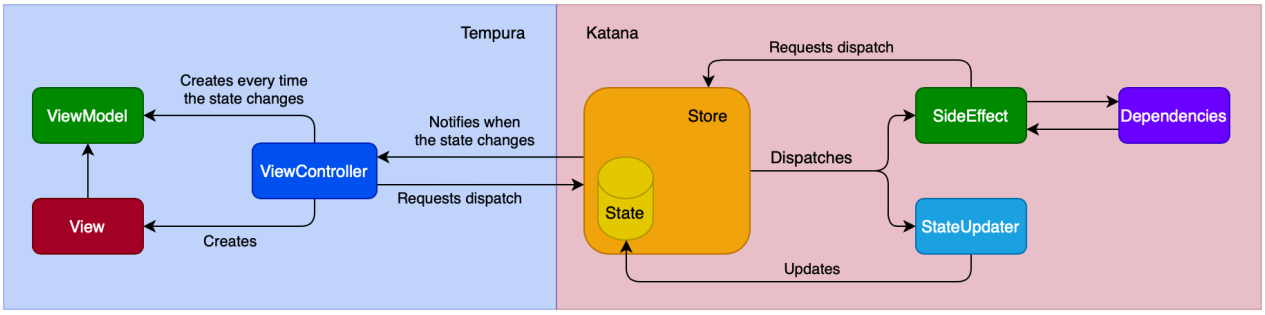

Figure 3.2: Katana/Tempura framework.

3.1 Katana

> *Katana is a modern Swift framework for writing iOS applications' business logic that are testable and easy to reason about. Katana is strongly inspired by Redux.*

Redux was originally a JavaScript library used for managing an application state. It can be described with three fundamental principles.

- *Single source of truth*
  The global state of the application is stored in an object tree within a single store.
- *Read-only state*
  The only way to change the state is to emit an *action*, i.e., an object describing what happened.
- *Changes are made with pure functions*
  To specify how the state tree is transformed by actions, the developer needs to write pure functions called *reducers*.

In Katana, actions and reducers are combined into **`StateUpdaters`**: a `StateUpdater` in an object that describes what happened and provides the code to update the state. All the `StateUpdaters` are executed on the same serial GCD [1] queue: this avoid the occurrence of subtle race conditions.

Updating the application's state using pure functions is nice and it has a lot of benefits. Unfortunately, applications have to deal with the external world (e.g., API calls, disk files management, ...). For all this kind of operations, Katana provides the concept of **`SideEffects`**. `SideEffects` can be used to interact with other parts of your applications and then dispatch new `StateUpdaters` to update your state.

3.1.1 SmartTourist's state

```
1 struct AppState: State, Codable {
2     var locationState = LocationState()
3     var favorites = [WDPlace]()
4     var settings = Settings()
5     var pedometerState = PedometerState()
6     var cache = Cache()
7     var needToMoveMap = false
8 }
```

Following Redux's principles, we preferred to divide the global state in sub-structures, in order to improve code readability and maintainability.

[1] https://developer.apple.com/documentation/DISPATCH

3.2 Tempura

> *Tempura is a holistic approach to iOS development. It borrows concepts from Redux (through Katana) and MVVM.*

Figure 3.3: MVVM architecture overview

Tempura uses Katana to handle the logic of your app: the portion of the state needed to render the UI of a screen is selected by a `ViewModel`.

The UI of each screen of your app:

- is composed in a `ViewControllerModellableView`, that exposes callbacks (we call them *interactions*) to signal that a user action occurred, and renders itself based on the content of the `ViewModel`;
- is managed by a `ViewController`. Out of the box it will automatically listen for state updates and keep the UI in sync. The only other responsibility of a `ViewController` is to listen for interactions from the UI and dispatch actions to change the state.

```
1  struct CounterViewModel: ViewModelWithState {
2      var countDescription: String
3
4      init(state: CounterState) {
5          self.countDescription = "\(state.counter)"
6      }
7  }
8
9
10 class CounterView: UIView, ViewControllerModellableView {    //
11     var counterLabel = UILabel()
12     var addButton = UIButton(type: .system)
13
14     var didTapAdd: Interaction?
15
16     func setup() {
17         self.addButton.on(.touchUpInside) { _ in
18             self.didTapAdd?()
19         }
20         addSubview(self.counterLabel)
21         addSubview(self.addButton)
22     }
23
24     func style() {
25         self.backgroundColor = .systemBackground
26         self.counterLabel.font = UIFont.systemFont(ofSize: 32 + 16)
```

```
27          self.addButton.setImage(UIImage(systemName: "plus.circle.fill"),
28                                  for: .normal)
29          self.addButton.tintColor = .systemGreen
30      }
31
32      func update(oldModel: CounterViewModel?) {
33          guard let model = self.model else { return }
34          self.counterLabel.text = model.countDescription
35          self.setNeedsLayout()
36      }
37
38      override func layoutSubviews() {
39          super.layoutSubviews()
40          counterLabel.pin.center().sizeToFit()
41          addButton.pin
42              .below(of: counterLabel, aligned: .center)
43              .marginTop(20)
44              .sizeToFit()
45      }
46  }
47
48
49  class CounterViewController: ViewController<CounterView> {
50      override func setupInteraction() {
51          rootView.didTapAdd = { [unowned self] in
52              self.dispatch(IncrementCounter())
53          }
54      }
55  }
```

Listing 3.1: Tempura example.

3.3 Hydra

Hydra is full-featured lightweight library which allows you to write better async code. It's partially based on JavaScript A+ specs and also implements modern construct like `await`, which allows you to write async code in a sync manner.

Hydra is based on ***promises***. A promise represents the eventual result of an asynchronous operation. The primary way of interacting with a promise is through its **`then`** method, which registers callbacks to receive either a promise's eventual value or the reason why the promise cannot be fulfilled.

With Hydra, developers are also able to resolve many independent async operations simultaneously and get all values at the end, or to resolve dependent async operations by passing the result of each value to the next operation, and then get the final result.

```
// Create a promise
func getPhoto(url: String) -> Promise<UIImage> {
    return Promise<UIImage>(in: .background) { resolve, reject, status in
        if let data = try? Data(contentsOf: url), let img = UIImage(data: data) {
            resolve(img)
        } else {
            reject(ApiError())
        }
    }
}

// Chain promises together
func loginUser(_ name: String, _ pwd: String) -> Promise<User>
func getFollowers(user: User) -> Promise<[Follower]>
func unfollow(followers: [Follower]) -> Promise<Int>

loginUser(username, pass).then(getFollowers).then(unfollow).then { count in
    print("Unfollowed \(count) users")
}.catch { error in
    print(error.localizedDescription)
}
```

Listing 3.2: Hydra examples.

4 Services and libraries

4.1 Internal libraries

SmartTourist uses consistently internal libraries and services:

- ***Core Location***: for location updates.
- ***MapKit***: for maps.
- ***Core Motion***: for accessing accelerometer, gyroscope, pedometer, and environment-related events.

4.2 External services

SmartTourist relies almost only on *free data and services*.

Info: The choice to rely on free services has been carefully pondered and it didn't come without troubles. SmartTourist has been designed to be a free application in an eventual public launch, thus it would be impossible to support the cost of Google Places API without any premium subscription. Moreover, we strongly believe in the Wikidata project and wanted to work with it.

Here, it follows the description of the external services used.

Wikipedia As can be seen in Section 5.1, the application displays attractions divided in three categories: nearest, popular and favorites. Nearest attractions are entirely taken from the Wikidata knowledge base, while a slightly different strategy is used for popular places. Favorites places are instead selected from the other two lists.
The Wikidata API is quite basic: attraction entries are retrieved with a SPARQL query embedded in a HTTP API call, while details and pictures are retrieved with standard HTTP API calls from different endpoints.

```
1 SELECT DISTINCT ?place ?placeLabel ?location ?image
2                 ?instance ?website ?wikipediaLink WHERE {
3     SERVICE wikibase:label { bd:serviceParam wikibase:language "en, it" }
4     SERVICE wikibase:around {
5         ?place wdt:P625 ?location .
```

```
        bd:serviceParam wikibase:center "Point(\(location.longitude) \(location.
            latitude))"^^geo:wktLiteral .
        bd:serviceParam wikibase:radius "\(radius)" .
    }
    ?place wdt:P31 ?instance  .
    ?place wdt:P18 ?image .
    OPTIONAL {?place wdt:P856 ?website} .
    OPTIONAL {?wikipediaLink schema:about ?place;
        schema:inLanguage "en";
        schema:isPartOf [ wikibase:wikiGroup "wikipedia" ]} .
}
```

Listing 4.1: SPARQL query for attractions retrieving.

```
SELECT DISTINCT ?city ?cityLabel ?country ?countryLabel ?population ?area
                ?elevation ?link ?facebookPageId ?facebookPlacesId
                ?instagramUsername ?twitterUsername ?image ?coatOfArmsImage
                ?cityFlagImage ?countryCode ?wikipediaLink WHERE {
    BIND( <http://www.wikidata.org/entity/\(cityId)> as ?city ).
    OPTIONAL {?city wdt:P17 ?country}.
    OPTIONAL {?city wdt:P1082 ?population}.
    OPTIONAL {?city wdt:P2046 ?area}.
    OPTIONAL {?city wdt:P2044 ?elevation}.
    OPTIONAL {?city wdt:P856 ?link}.
    OPTIONAL {?city wdt:P2013 ?facebookPageId}.
    OPTIONAL {?city wdt:P1997 ?facebookPlacesId}.
    OPTIONAL {?city wdt:P2003 ?instagramUsername}.
    OPTIONAL {?city wdt:P2002 ?twitterUsername}.
    OPTIONAL {?city wdt:P18 ?image}.
    OPTIONAL {?city wdt:P94  ?coatOfArmsImage}.
    OPTIONAL {?city wdt:P41 ?cityFlagImage}.
    OPTIONAL {?country wdt:P297 ?countryCode}.
    OPTIONAL {?wikipediaLink schema:about ?city;
      schema:inLanguage "en";
      schema:isPartOf [ wikibase:wikiGroup "wikipedia" ]}.
    SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
```

Listing 4.2: SPARQL query for city detail retrieving.

Google Unfortunately, attractions retrieved from Wikidata do not contain any properties that could be used to measure popularity. For this the reason, we decided to use Google Places API to retrieve popular places. Google Places API is not as expensive as other places APIs, and include ratings gathered from a quite active user base. Further details about the data merging process is given in the next section.

4.2.1 Data merging

While nearest places retrieval is pretty straightforward – we get them from the Wikidata API and we request pictures and the description only when needed – we wanted to high-

light the processes for retrieving popular places from Google Places API and merging the data with Wikidata's. Once these places are converted and enriched, both kinds of places are treated equally by the application.

Algorithm 4.1 Popular place retrieving.

```
 1: procedure GetPopularPlaces(city)
 2:     if places are cached and the cache has not expired then
 3:         return cachedPlaces
 4:     else
 5:         places ← googleAPI.getPopularPlaces(city)
 6:         popularPlaces ← ∅
 7:         for popularPlace in place do
 8:             p ← convertToWikidataFormat(popularPlace)
 9:             articleName ← findArticleName(p)
10:             wikidataId ← getWikidataId(articleName)
11:             p ← getMissingDetails(p, wikidataId)
12:             popularPlaces ← popularPlaces + p
13:         end for
14:         updateCache(city, popularPlaces)
15:         return popularPlaces
16:     end if
17: end procedure
```

In order to help the merging process between Google and Wikidata attractions, we adopted a fuzzy search library, that together with some heuristics is able to reliably merge data based on the attraction name.

4.3 External libraries

The app uses a multitude of third-party libraries that can be roughly divided into three kind: architectural libraries, back-end libraries and front-end libraries.

Kind	Library	Description
Architectural	Katana	Katana is a modern Swift framework for writing iOS applications' business logic that are testable and easy to reason about. Katana is inspired by Redux.
	Tempura	Tempura is a holistic approach to iOS development, it borrows concepts from Redux (through Katana) and MVVM.
	Hydra	Hydra is full-featured lightweight library which allows you to write better async code. It's partially based on JavaScript A+ specs and also implements modern construct like await which allows you to write async code in sync manner.
Back-end	DeepDiff	DeepDiff tells the difference between 2 collections and the changes as edit steps.
	Fuse	Fuse is a super lightweight library which provides a simple way to do fuzzy searching.
	Alamofire	Alamofire is an HTTP networking library written in Swift.
	SwiftyXMLParser	Simple XML Parser implemented in Swift.
Front-end	PinLayout	Extremely fast views layouting without auto layout. No magic, pure code, full control and blazing fast. Concise syntax, intuitive, readable and chainable.
	FlexLayout	Angular Flex Layout provides a sophisticated layout API using Flexbox CSS + mediaQuery.
	Cosmos	A star rating control for iOS/tvOS written in Swift.
	ImageSlideshow	Customizable Swift image slideshow with circular scrolling, timer and full screen viewer.
	MarqueeLabel	A drop-in replacement for UILabel, which automatically adds a scrolling marquee effect when the label's text does not fit inside the specified frame.
	FontAwesome	Use Font Awesome in your Swift projects.
	FlagKit	Beautiful flag icons for usage in apps and on the web.

5 User interface

For the user interface design, we followed Apple’s UI guidelines, but we preferred also to use custom UI element to better fit Katana/Tempura paradigm’s requirements and to let the app have a modern and captivating style.

5.1 Screenshots

Here follow some screenshots of the application in both Light and Dark mode.

5.1.1 MapView

MapView - nearest places

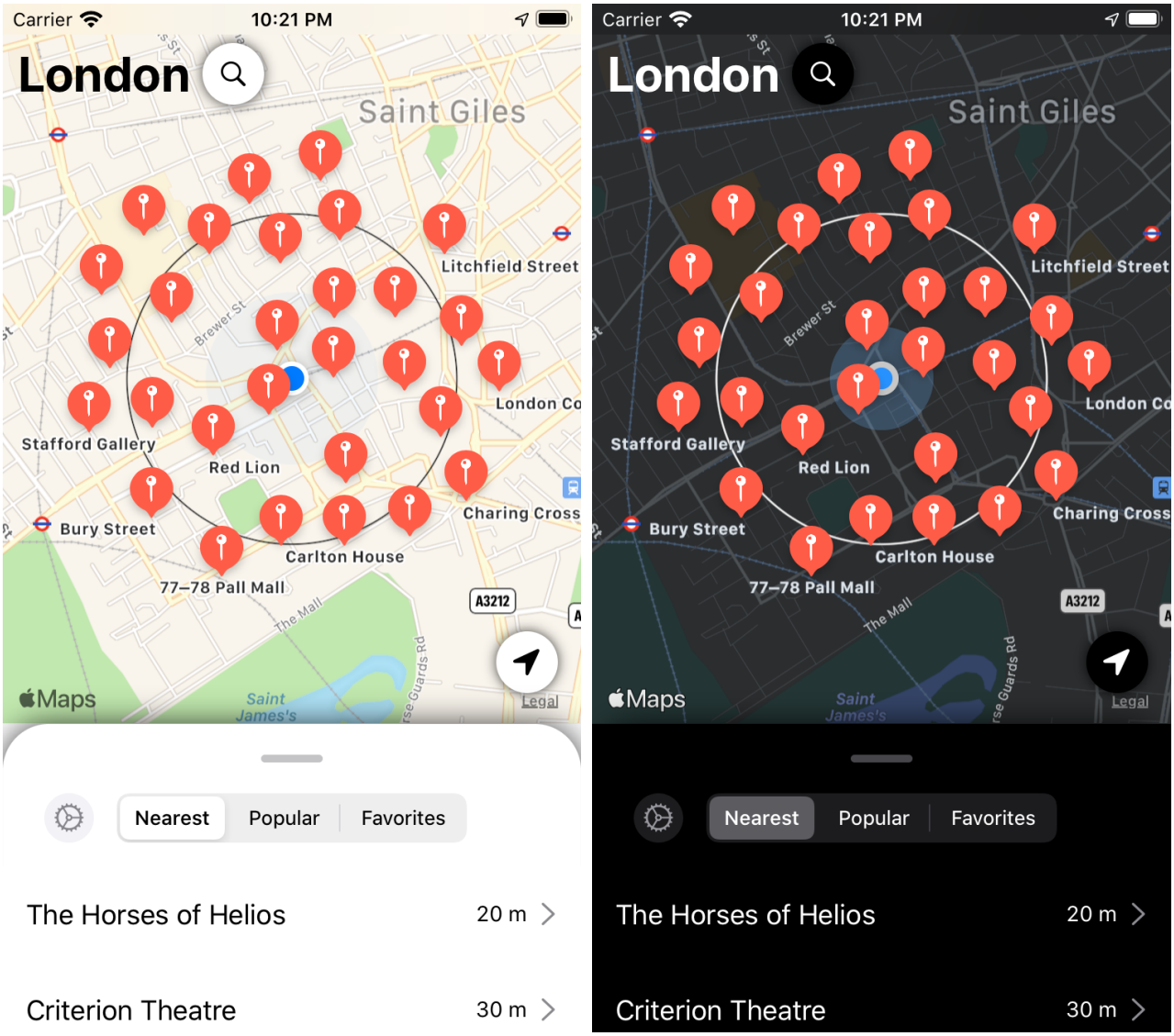

Figure 5.1: Map view with nearest tab selected in both light and dark mode

This is the first view that appears when you open the app. Attractions, up to the maximum selected number, are displayed both in the maps and in the list below. In the list, they are ordered by their distance to the user current position.
In this view you can tap the city name to open the city detail view or an attraction to open the corresponding detail view. By tapping the search button, the user is presented with the familiar search bar. Settings are reachable by tapping the gear button near the selectors.
The tab view for attraction list is resizable.

5.1.2 MapView - popular places

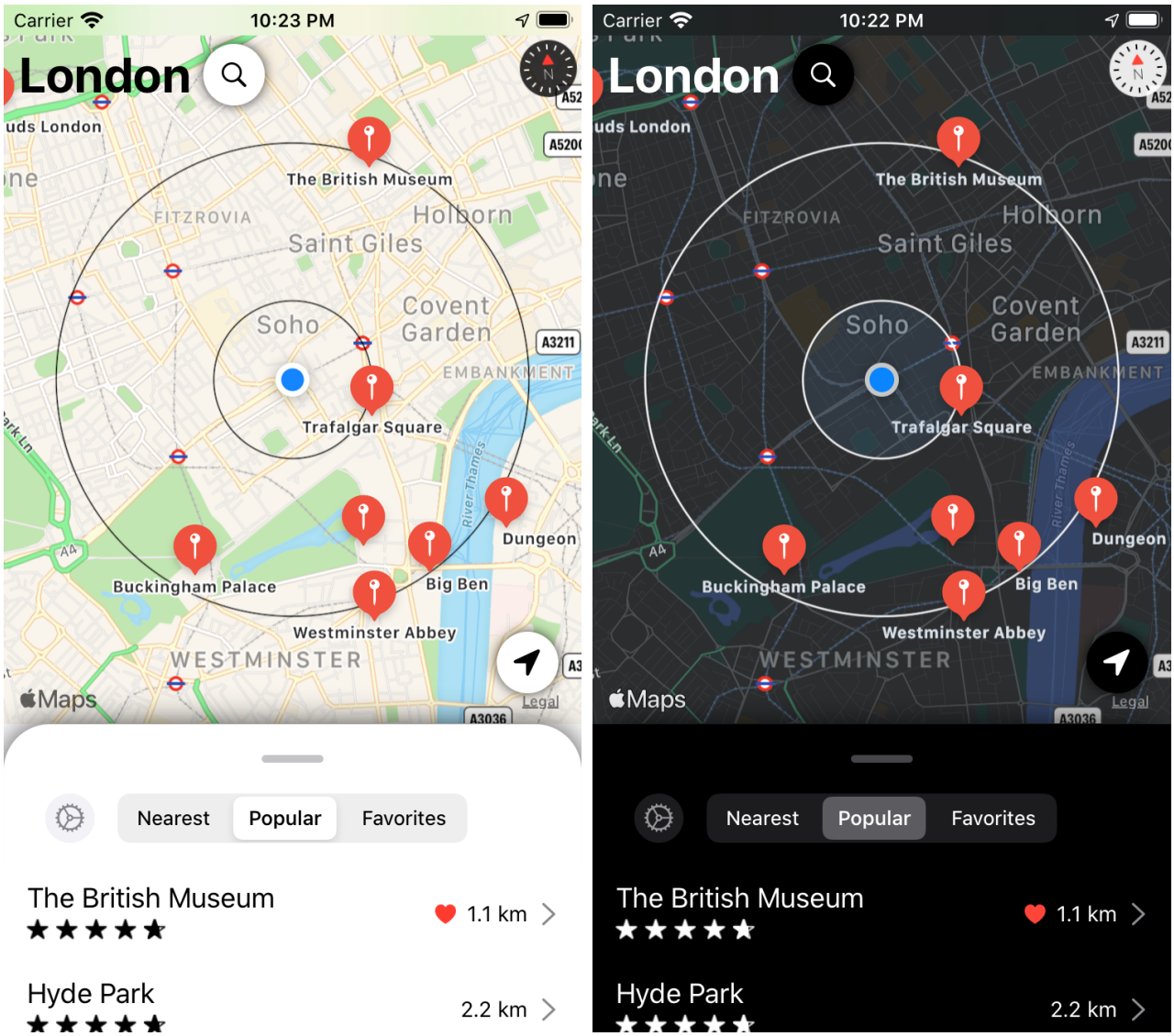

Figure 5.2: Map view with popular tab selected in both light and dark mode

Same as the previous view, but here the popular tab is selected. Thus, only the most popular places are displayed. In the list, favorite items are identified by a hearth icon and the rating of each attraction is also displayed.
In this screenshot, circles that represent distance are more visible. They respectively represent the distance that the user can cover by walking at their current walking speed in 5 and 15 minutes, respectively.

5.1.3 MapView - favorite places

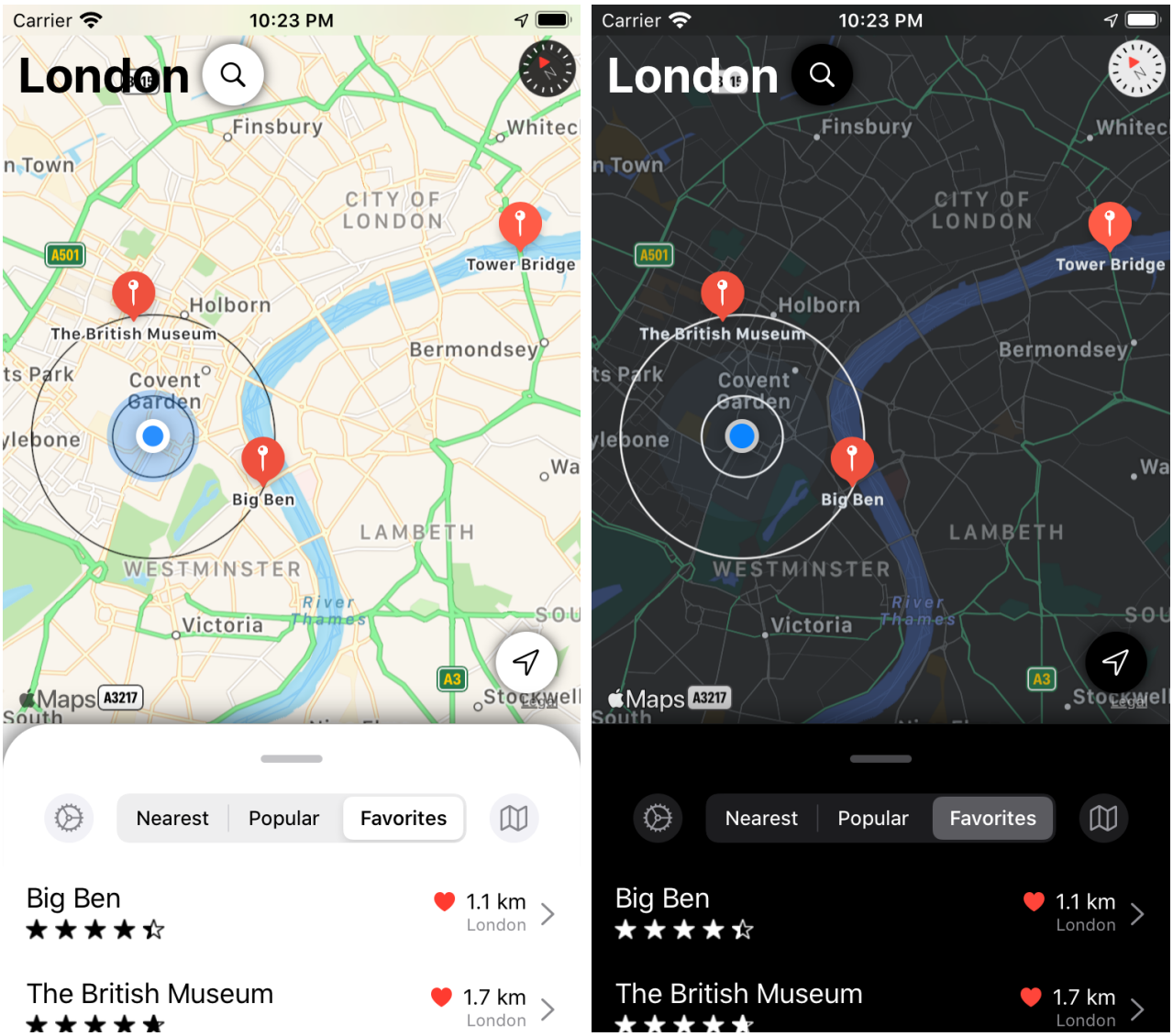

Figure 5.3: Map view with favorite tab selected in both light and dark mode

Same as the previous view, but here the favorite tab is selected. Since it's a global tab, they are grouped by city, that is also displayed.
An additional button with a map is displayed, to let the user access the global view of favorite items.

5.1.4 Global favorite view

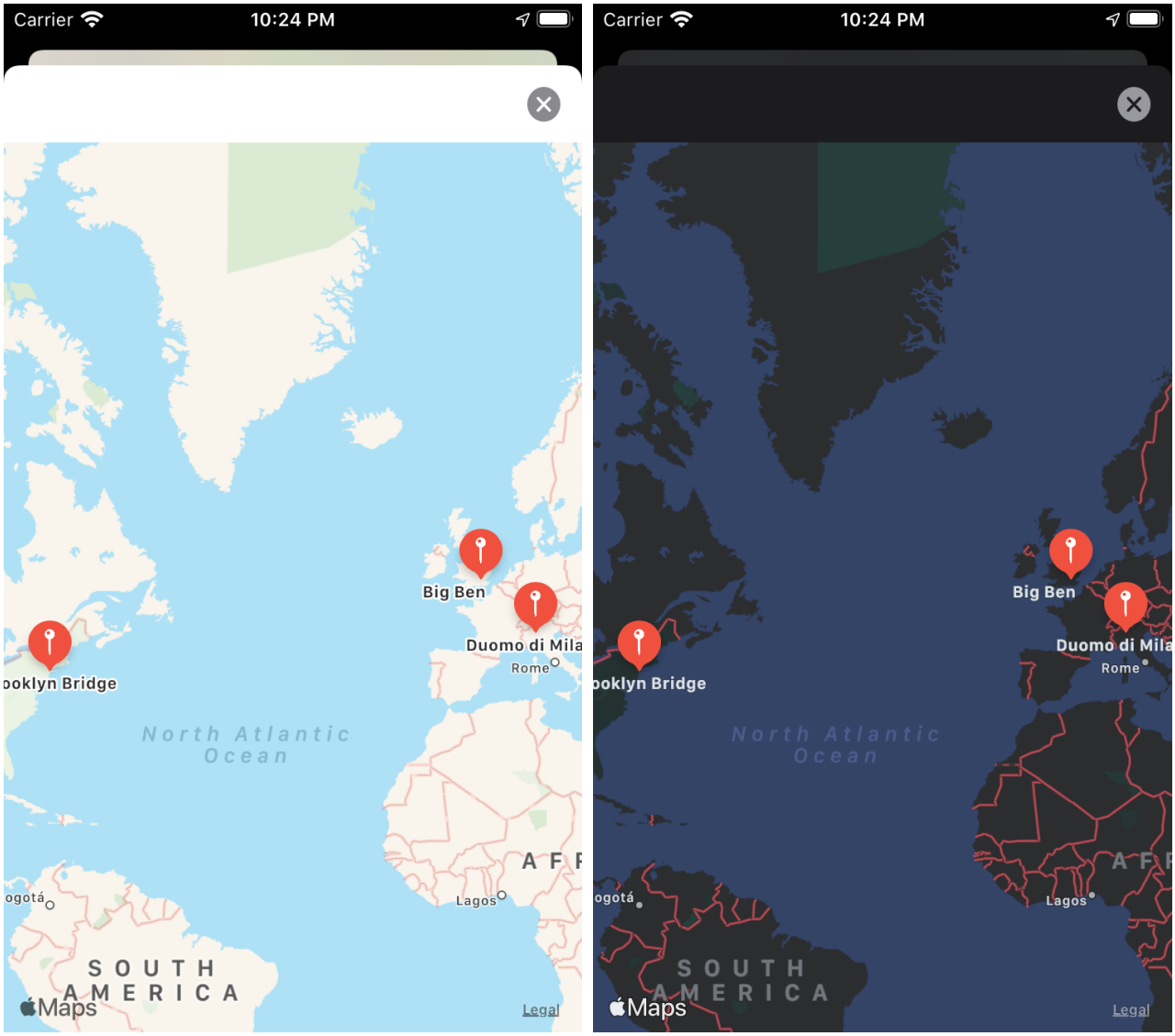

Figure 5.4: Favorite worldwide view in both light and dark mode

In this view, all user's favorite attractions are displayed in a fullscreen map.

5.1.5 Searching view

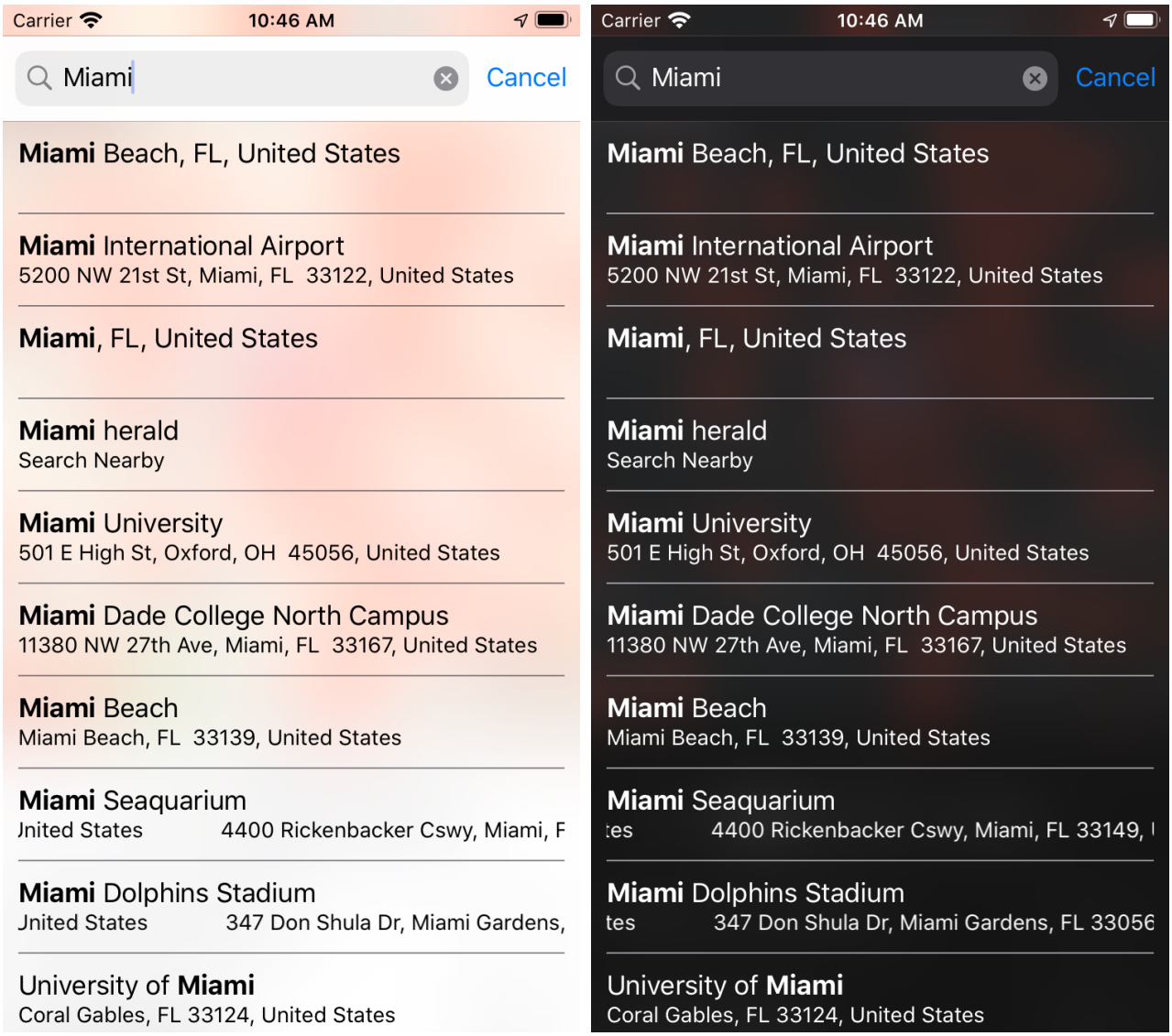

Figure 5.5: Searching view in both light and dark mode

When you tap the lens button on the map view, the familiar searching view provided by iOS is opened. Both cities and attractions can be searched.

5.1.6 Attraction detail view

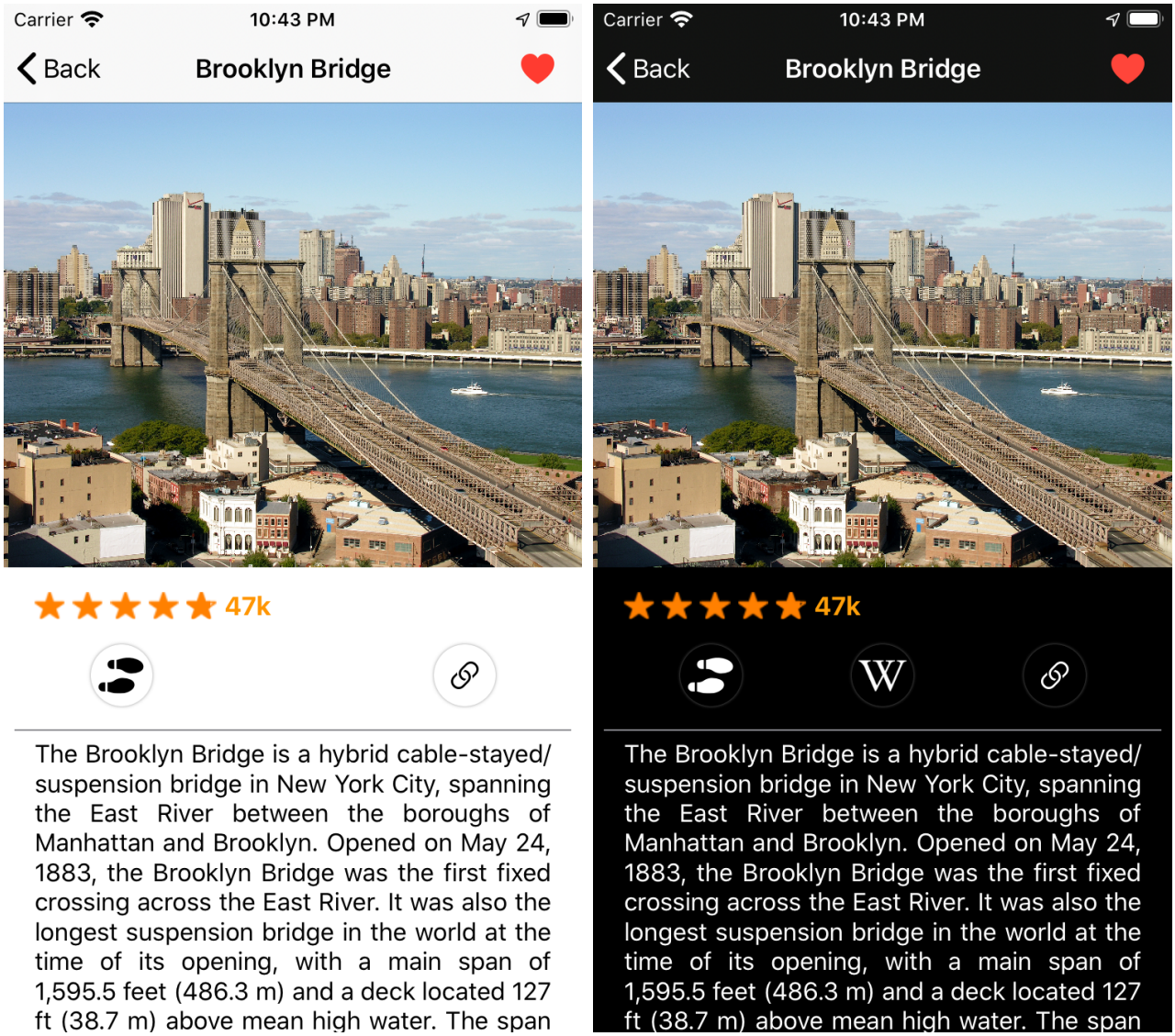

Figure 5.6: Attraction detail view in both light and dark mode

This is the detailed view that is presented when the user tap a specific attraction. A slideshow of pictures of the attraction is displayed together with the rating (if available) and buttons that let you access the map direction, the complete Wikipedia Article and the related website. In the navigation bar, the back button is available together with the hearth button that lets the user mask the attraction as favorite. The view is scrollable to display the entire place description extract. A detailed map with the specific attraction location is also provided at the bottom.

5.1.7 City detail view

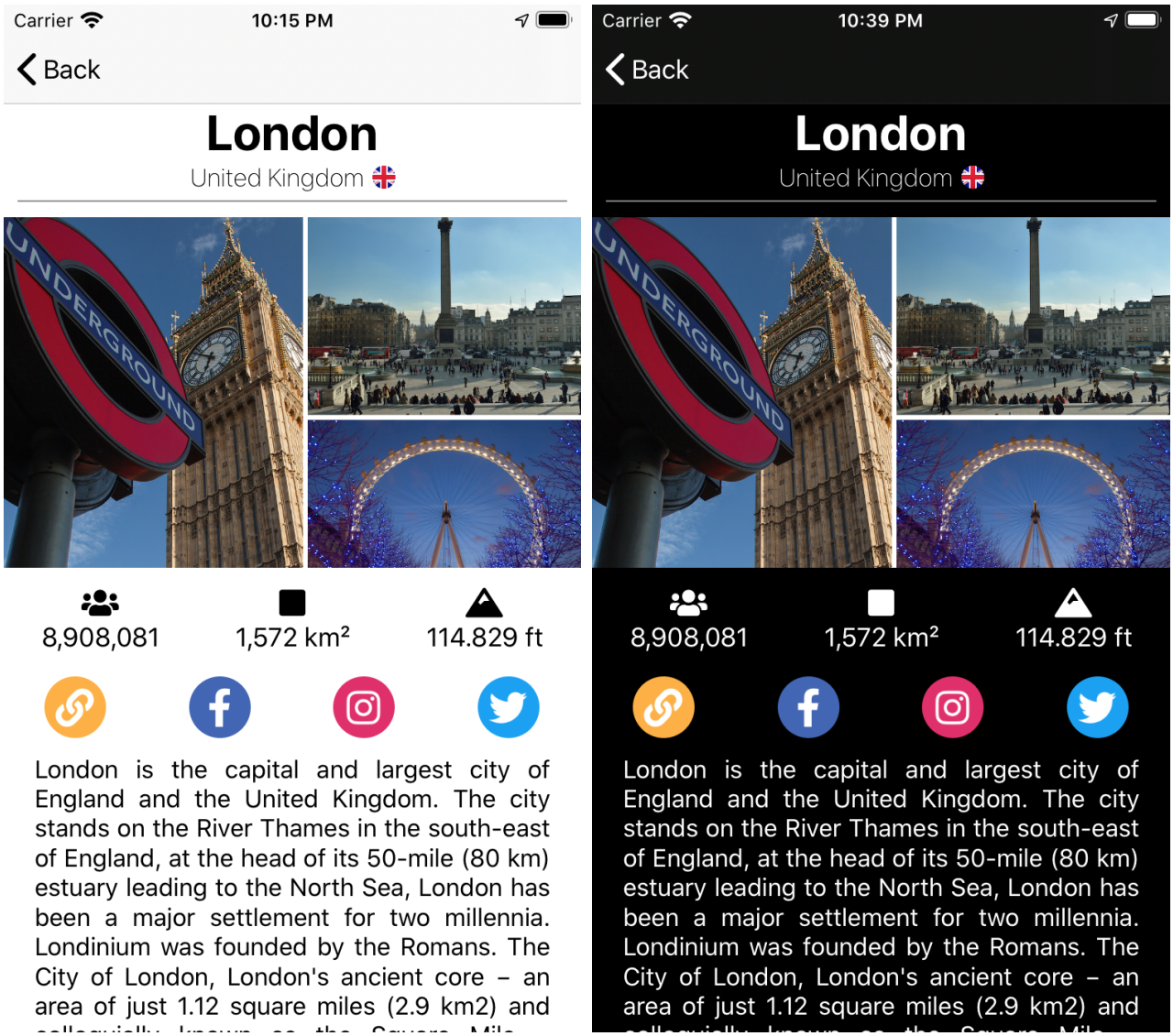

Figure 5.7: City detail view in both light and dark mode

This is the city detail view that is displayed when the user tap the city name in the main view. A pictures slideshow is displayed together with some information such as: population, area and altitude. A set of social links is retrieved and presented to the user. The view is scrollable and a summary description of the city is given.

5.1.8 Settings view

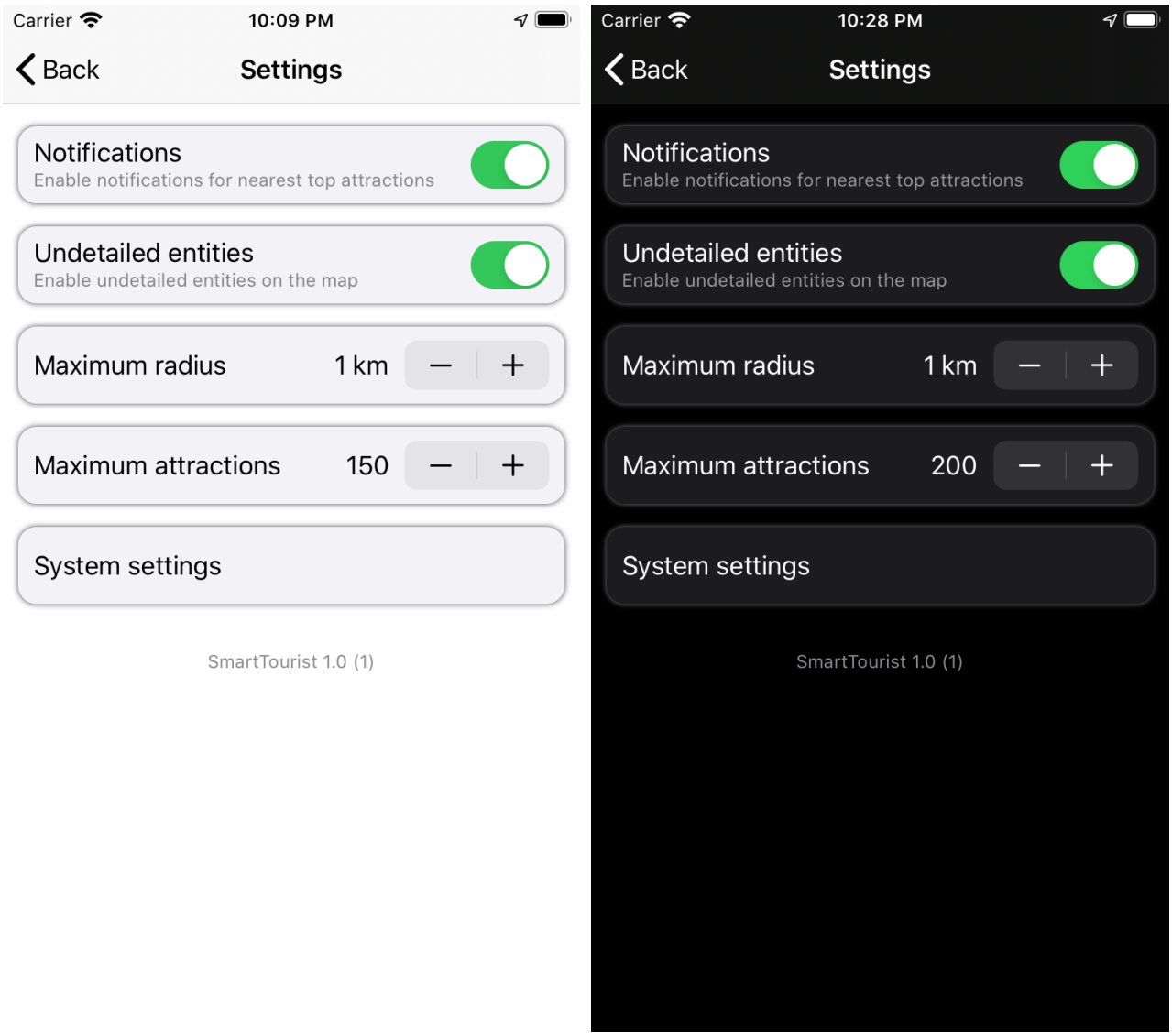

Figure 5.8: Settings view in both light and dark mode

Settings view is displayed when the gear icon on the main view is tapped. This is a quite standard settings view where some parameters about the application can be set.

6 Notifications

Notifications are designed as a way for the user not to miss popular attraction near them. Each time the user is in a 5 minutes range of a popular attraction, a notification is triggered. This works in both background and foreground mode ("*always active*" location tracking is required).
The shown notification is a *rich notification* with callbacks: **take me there** will directly open navigation app with that attraction as destination and **view** will open SmartTourist detail attraction view.

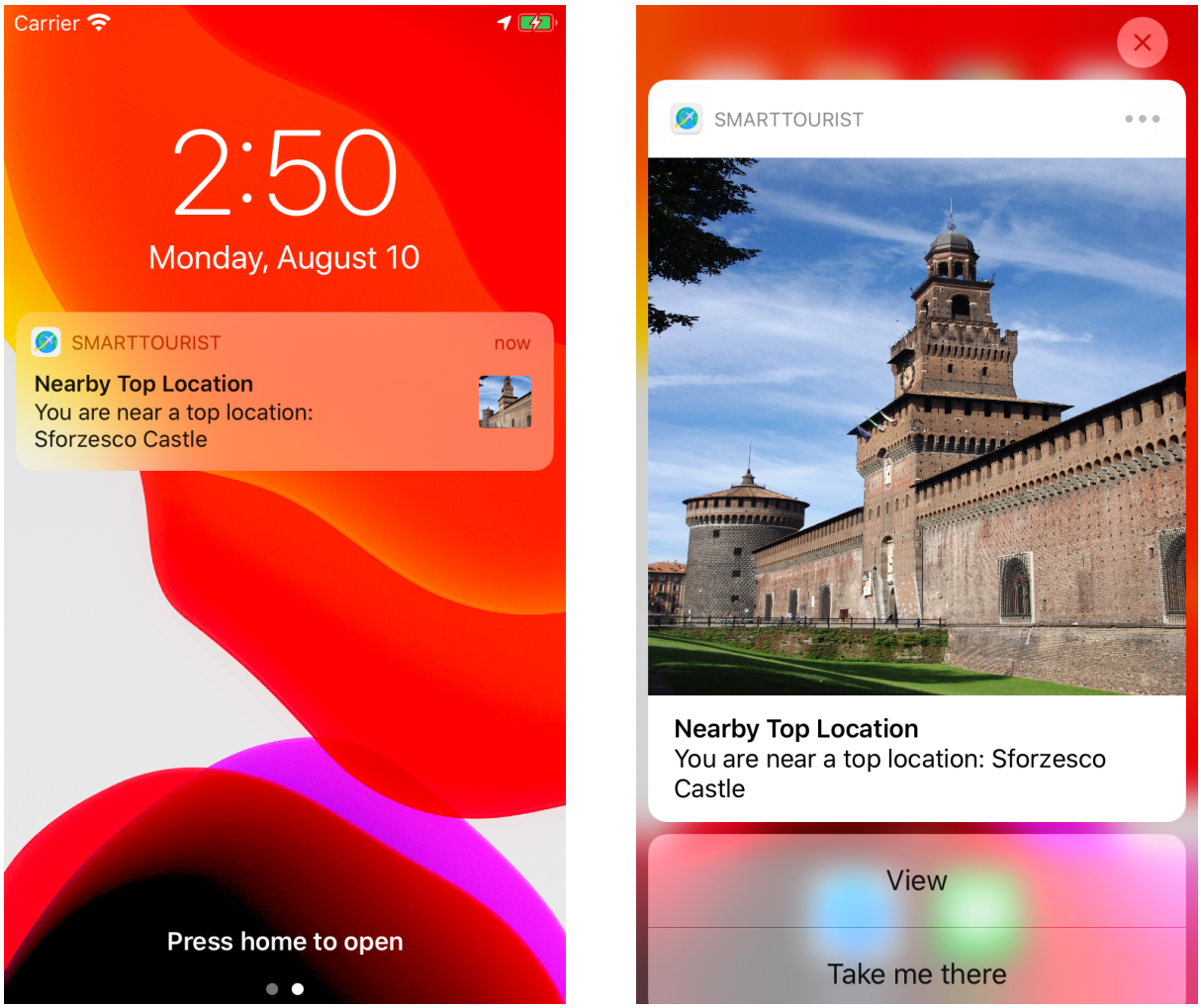

Figure 6.1: Notification layouting

7 Testing

7.1 Unit testing

The application does not contain any part that is suitable for unit testing. This is due to the fact that its core functionality can be reduced to making API calls and displaying the results back to the user. It is well known that there are too many variables involved in API calls that make them unsuitable for unit testing. For this reason – except for a small piece of code, a sorted insert into an array – we decided not to do any unit testing.

7.2 UI testing

Since we didn't use the standard Apple SDK to layout views – we just used its components – we couldn't use the provided UI testing framework. Luckily, Tempura provides its own UI testing framework, which is a little different than Apple's. Basically, it allows to take screenshots of a view in all possible states, with all devices and all supported languages. This way you can then manually check that all the components of a view are correctly displayed.

Unfortunately, we had some difficulties while trying to make this framework work due to the highly asynchronous nature of our main views, an approach suggested by our Bending Spoons tutor aimed to make the views more responsive. For this reason, only the simpler views have been completely tested.

8 Future development

During the development of the application, we encountered a series of issues that would require some more work in case of a public release.
The most important aspect that needs some improvement, in order to provide a premium-grade service to our users, is a better data integration system. The perfect solution would be a custom backend that allows to automatically integrate data with the possibility of manually tweaking the results.

These are other ideas that have come to our mind during the development process.

- An AR experience where the user is able to point the camera to an attraction and see its name and a short description.
- The integration with some social network able to provide information about local events such as art exhibitions, music shows and other cultural events.
- The possibility of shooting pictures and then adding some text or stickers that identify the attraction or the city where the picture was taken.

9 Effort spent

The actual development took place between January 2020 and May 2020, while during July/August 2020 just some minor changes have been put in place, together with the drafting of all the needed documents and materials.

The authors worked together on site and remotely to setup the crucial parts of the application, then, they worked mostly on their own with continuous mutual contacts. They also had a couple of starting tutoring session kindly offered by Bending Spoons at their HQ to learn the Katana/Tempura framework.

They didn’t use any professional time tracker, but it’s safe to assume that the total amount of time spent is around 300 hours per worker, roughly divided in this way:

- 20% platform meet and greet
- 40% actual development
- 15% free services refactoring
- 5% testing
- 20% documents drafting