POLITECNICO
MILANO 1863

Disaster Monitor
Design and Implementation of Mobile Applications Project
Design Document

Stefano Martina, Francesco Peressini

A.Y. 2019/2020

Politecnico di Milano — May 2, 2020

Contents

1 Introduction

1.1 Purpose

The purpose of this Design Document is to give a functional description of the Disaster Monitor application. The focus is on how the whole system is implemented, with particular attention to its architecture.
The application is developed for the course of "Design and Implementation of Mobile Applications" at Politecnico di Milano.

1.2 Choice of the application

Bending Spoons[1] offered us the possibility to develop one of their ideas: among the proposals we decided to implement Disaster Monitor, an application that allows users to visualized the seismic activity around the globe, see earthquakes in different kind of maps, share status on the social to let parents and friends know to be in safe and be notified when there is some seismic activity going on.

The application is available for iOS devices (iPhone and iPad) and uses the two principal frameworks developed by Bending Spoons: Katana and Tempura.
The development of the entire project is assisted by a tutor from Bending Spoons.

[1] Bending Spoons: `https://bendingspoons.com`

2 General Overview

2.1 Idea

Disaster Monitor is an application that allows users to monitor earthquakes globally and receive notifications for specific seismic events. The proposed idea is:

> Be always aware of what's happening around you: be sure to avoid earthquakes!
> With this app, you will always be notified when there is some seismic activity going on.
> Set custom rules to be notified whenever an event is happening.
> See them in beautiful maps and share your status on the social to let your parents know that you are safe!

Starting from this, the goal is to develop a native iOS application using Swift as programming language.

2.2 Core features

Disaster Monitor app is divided in three main pages using a Tab Bar: the first page contains a list of all the principal events, the second one contains multiple kind of maps to allow users to better visualise the earthquakes and the last one contains all the settings of the app itself.

2.2.1 Main events and filtering

As stated above, the first page, which is also the first page displayed once the app is launched, contains a list of all the events. By default, this page displays all the events of the past seven days and it can be filtered by using either a search bar or using a filters, located in a specific separate page.
Filtering values and the searched query in the search bar are made persistent so that they will be maintained for future application launches.

2.2.2 Map

The Map page represents all the events in characteristic maps: in the classic and satellite representations, all the events are shown with a corresponding pin and clusterized by density; it is also available a heat map, again, useful for representing the distribution and density of the events.

2.2.3 Notifications

Disaster Monitor allows to receive notifications for seismic activity: user can add a location to be monitored and add specific rules under which be advised.

2.2.4 Background Data Fetching

One of the main needs is to prepare data before the app is launched.
Apple's APIs provide ***BackgroundTasks***[2]. The documentation states:

> Use the BackgroundTasks framework to keep your app content up to date and run tasks requiring minutes to complete while your app is in the background. Longer tasks can optionally require a powered device and network connectivity. Register launch handlers for tasks when the app launches and schedule them as required. The system will launch your app in the background and execute the tasks.

Disaster Monitor schedules a BGTask to update the list of events every time the app enters in background. iOS does not give any guarantee about when this task will be executed, since the OS decides when to run it based on user's daily routines and app usage.

[2]BackgroundTasks: `https://developer.apple.com/documentation/backgroundtasks`

2.2.5 Background Data Cleaning

Since events older than seven days are no more relevant (and also not displayed by the application as stated before), Disaster Monitor implements a cleaning process to remove those events. A particular class of the already introduced BackgroundTasks is called ***BGProcessingTask***[3]. The documentation states:

> Use processing tasks for long data updates, processing data, and app maintenance.

Disaster Monitor schedules the database cleaning every time the app enters in background.

[3] BGProcessingTask: `https://developer.apple.com/documentation/backgroundtasks/bgprocessingtask`

3 Architectural design

Disaster Monitor communicates directly with different API services gathering data and displaying them in its various pages.
Below is reported a schema which better explains all the interconnections.
For further information concerning integrations see External services and libraries section.

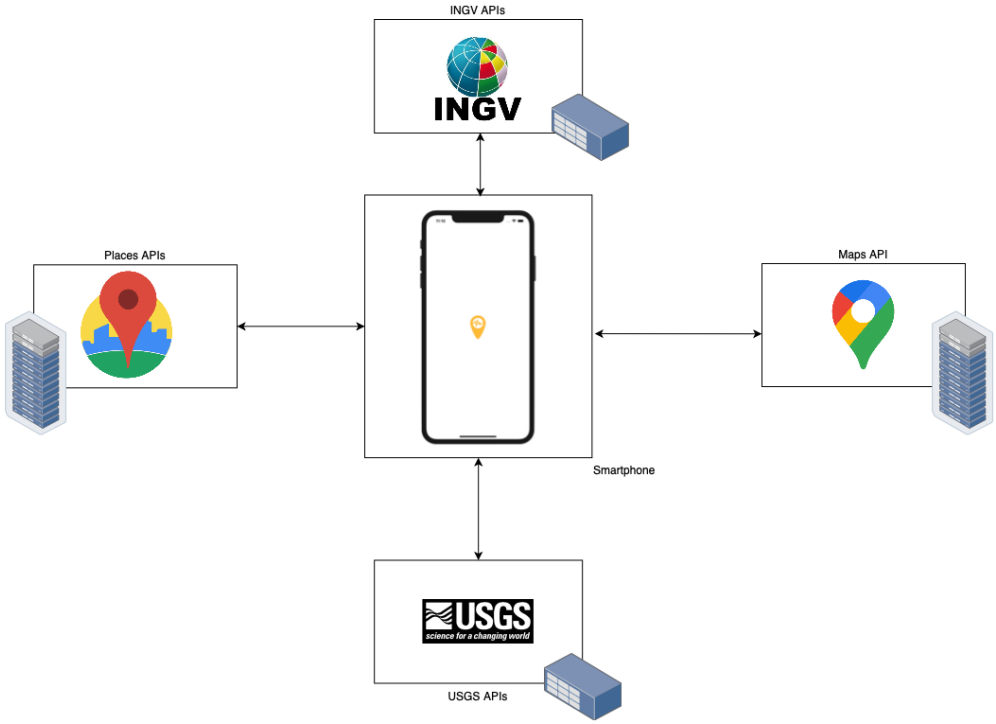


Figure 1: Architectural design schema

4 Data Design

4.1 Katana

Katana[4] 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. The app state is entirely described by a single serializable data structure, and the only way to change the state is to dispatch a *StateUpdater*. A StateUpdater is an intent to transform the state, and contains all the information to do so. Because all the changes are centralized and are happening in a strict order, there are no subtle race conditions to watch out for.

4.1.1 Our data structure

The entire app *State* is defined in a single struct, all the relevant application information are placed here. Reported below a possible representation of Disaster Monitor app state.
The most important element is events, an array of ***Event*** structs; Event contains all the information about a single seismic event such as coordinates (latitude and longitude), the magnitude, timestamp etc.
The regions array is composed of ***Region*** elements, a structure containing the information concerning the user's monitored places such as the selected location and the custom thresholds within which be notified. All the other variables are useful for the customization of the user experience.

AppState.swift

```
struct AppState: State {
        var events = [Event]()
        var regions = [Region]()
        var magnitudeFilteringValue: Float = -1.0
        var displayedDays: Int = 7
        var searchedString: String = ""
        var message: String = "Safe␣Message"
        var isNotficiationEnabled: Bool = true
                        ...
}
```

4.1.2 State Updater

The app State can only be modified by a StateUpdater. A StateUpdater represents an event that leads to a change in the State of the app and it should be a pure function, which means that it only depends on the inputs (that is, the state and the state updater itself) and it doesn't have side effects, such as network interactions.

4.1.3 Side Effects

Updating the application's state using pure functions is nice and it has a lot of benefits. Applications have to deal with the external world though (e.g. API call, disk files management, ...). For all this kind of operations, Katana provides the concept of *side effects*. Side effects can be used to interact with other parts of your applications and then dispatch new StateUpdaters to update your state.
Side Effects are used by Disaster Monitor to retrieve JSONs from the external APIs.

[4]Katana: `https://github.com/BendingSpoons/katana-swift`

4.1.4 Practical example of Katana's behaviour

StateUpdater & SideEffect Example

```
// Main Events Controller
override func viewDidLoad() {
    super.viewDidLoad()
    self.dispatch(GetEvents())
}

// The following SideEffect retrieves all the information from the APIs and
   then calls a StateUpdater to update the state

struct GetEvents: SideEffect {
    func sideEffect( ... ) throws {
        // API call
        let json = APIManager.getEvents()

        // Data are deserialized and prepared for the StateUpdater
        let data = preprareData(json)

        // At the end the StateUpdater is called
        context.dispatch(EventsStateUpdater(data))
    }
}
```

Side Effects are executed in a different (concurrent) queue with respect to the Main Thread so they don't freeze the UI while waiting for data.

Notice: APIs could return a large amount of data (JSONs with several thousand entries).
To solve this problem, data manipulation can be performed inside the SideEffect, executed concurrently with respect to other Side Effects, and by using an optimized data structure (for example a *Set* instead of an *Array* considering that scanning these structures costs $\mathcal{O}(1)$ and $\mathcal{O}(n)$ respectively).

4.1.5 State Persistence - Katana's Interceptors

The *Store* contains and manages the entire app State. It is responsible of managing the dispatched items (e.g., the State Updater).
When defining a Store one can provide a list of interceptors that are triggered in the given order whenever an item is dispatched. An interceptor is like a catch-all system that can be used to implement functionalities such as logging or to dynamically change the behaviour of the store. An interceptor is invoked every time a dispatchable item is about to be handled.
Disaster Monitor uses an interceptor to store part of the State in a file located in the physical Device. This allows the application to be already populated with data once launched and to just update the retrieved State with new or updated events.

5 User Interface

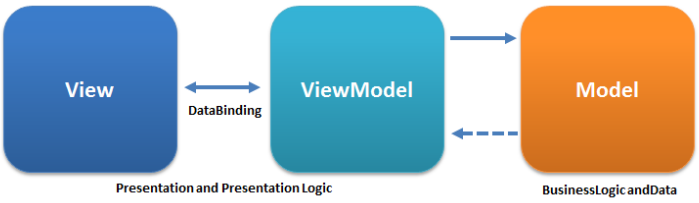
The user interface is strongly connected to the application State through Tempura. The entire development process was focused on the simplicity of interaction, thinking where to place graphical elements in base of their importance.

5.1 Tempura

Tempura[5] is a holistic approach to iOS development, it borrows concepts from Redux[6] (through Katana) and MVVM.

5.1.1 Model–view–viewmodel

> *Model–view–viewmodel (MVVM) is a software architectural pattern that facilitates the separation of the development of the graphical user interface (the view) – be it via a markup language or GUI code – from the development of the business logic or back-end logic (the model) so that the view is not dependent on any specific model platform. The view model of MVVM is a value converter, meaning the view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented. In this respect, the view model is more model than view, and handles most if not all of the view's display logic.*



[5]Tempura: `https://github.com/BendingSpoons/tempura-swift`
[6]Redux: `https://redux.js.org`

5.1.2 Practical example of Tempura's behaviour

Tempura uses Katana to handle the logic in our app: the part of the state needed to render the UI of a screen is selected by a *ViewModelWithState*.
The UI of each screen in the app is composed in a *ViewControllerModellableView*. It exposes callbacks (called interactions) to signal that a user action occurred. It renders itself based on the *ViewModelWithState*.
Each screen is managed by a *ViewController*: it automatically listens 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.

TempuraExample.swift

```
struct ListViewModel: ViewModelWithState {
  var todos: [Todo]

  init(state: AppState) {
    self.todos = state.todos
  }
}

class ListView: UIView,
                ViewControllerModellableView {
  // subviews
  var todoButton: UIButton = UIButton(type: .custom)
  var todoButton: UIButton = UIButton(type: .custom)
  var list: CollectionView<TodoCell,
                SimpleSource<TodoCellViewModel>>

  // interactions
  var didTapAddItem: ((String) -> ())?
  var didCompleteItem: ((String) -> ())?

  // update based on ViewModel
  func update(oldModel: ListViewModel?) {
    guard let model = self.model else { return }
    let todos = model.todos
    self.list.source =
        SimpleSource<TodoCellViewModel>(todos)
  }
}

class ListViewController: ViewController<ListView> {
  // listen for interactions from the view
  override func setupInteraction() {
    self.rootView.didCompleteItem = {
    [unowned self] index in
      self.dispatch(CompleteItem(index: index))
    }
  }
}
```

In the example above are presented all the main components described before: *ListViewModel* contains the part of the state needed to render the UI, *ListView* contains the UI elements, the interactions to handle the user's actions and the *update* function, called every time the ViewModel is updated.
ListViewController manages the *ListView* View and listens for interactions from the UI (in the example is presented the handling of the *didCompleteItem* interaction).

5.2 Screenshots

In this section we provide some screenshots of the application. All the pages follow the Human Interface Guidelines[7] provided by Apple and are able to auto-adapt their appearance depending if the system appearance is set in Light or Dark mode taking advantage of the semantic colors introduced in iOS 13.

5.2.1 Launch Screen

Figure 2: Launch screen

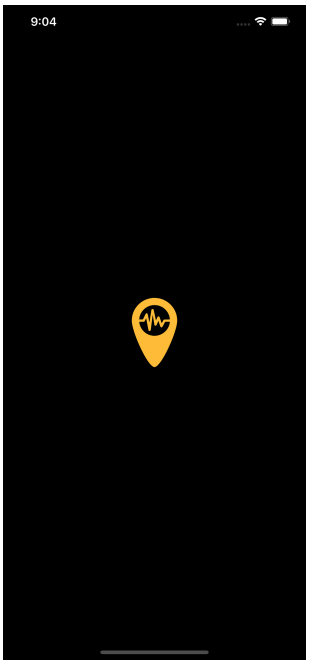


Figure 3: Launch screen (Dark Mode)

Once the app is launched, the Launch Screen is the first displayed View: in our case it simply consists in the application icon.
The Apple guidelines mentioned above state: *A Launch Screen appears instantly when your app starts up and is quickly replaced with the app's first screen, giving the impression that your app is fast and responsive.*

[7] Human Interface Guidelines: `https://developer.apple.com/design/human-interface-guidelines/ios/`

5.2.2 Main Events

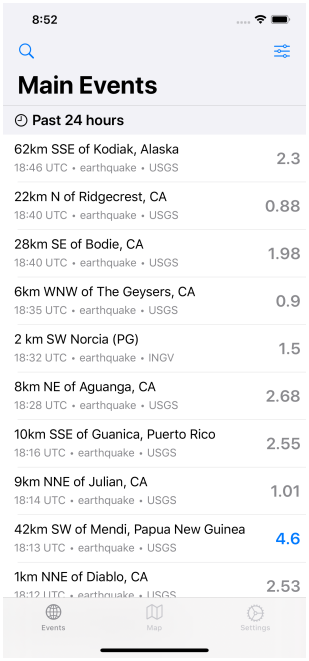


Figure 4: Main Events

Figure 5: Main Events (Dark Mode)

The first screen is the first tab of the Tab Bar and contains all the events in a list.
The events are listed in chronological order. In each cell are reported the most important information of an event such as time, the type of the seismic event and its source. On the right there is the magnitude of such event.
By tapping on the magnifying glass on the top left the user can access to a search field, otherwise, by tapping on the filtering icon on the top right, the user can access to a filtering page (view Filters).
Finally, by tapping on a certain event, the user can access to a page containing all the detailed information of such event.

5.2.3 Event

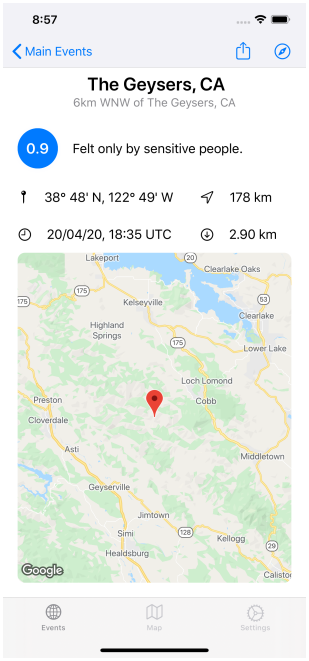


Figure 6: Event

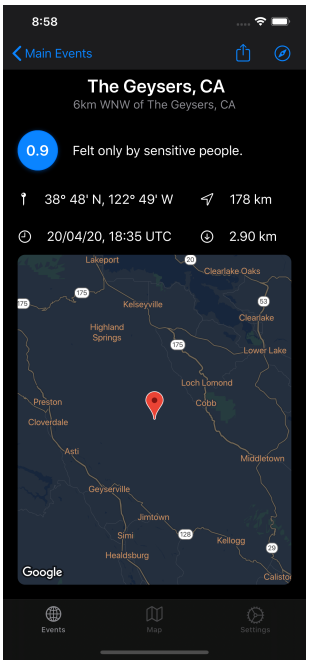


Figure 7: Event (Dark Mode)

The Event Page contains all the detailed information of a certain event. Analyzing the page from top to bottom we can find a share button to share the event through the Social Media or via message and a Safari button, to access the web page of the scientific agency that is providing the information.
Then, we have information about the place where the event happened, the magnitude, and, if available, a brief description with the number of users that felt the seismic event.
Under them we have more technical informations such as coordinates (latitude and longitude), the depth where the earthquake begins to rupture, the complete timestamp and the distance of the event with respect to the user current location.
Lastly, the event is showed in a large map.

5.2.4 Filters

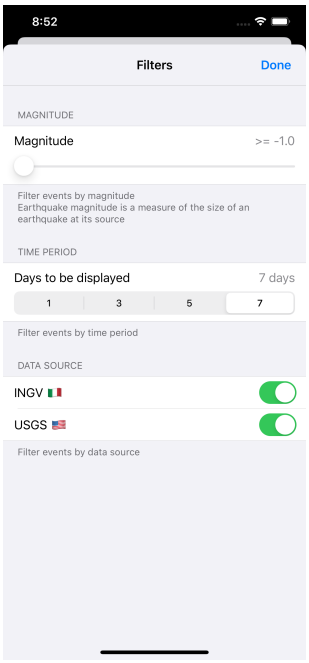


Figure 8: Filters

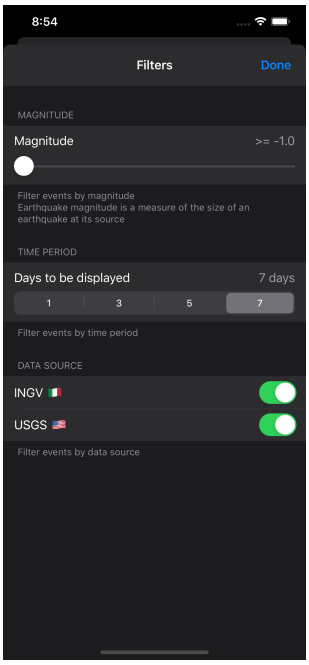


Figure 9: Filters (Dark Mode)

Filters allow the user to customize the Main Events page view, avoiding showing events that are not of his/her interested. In this page is possible to filter events for magnitude, displayed days and data source.

5.2.5 Map

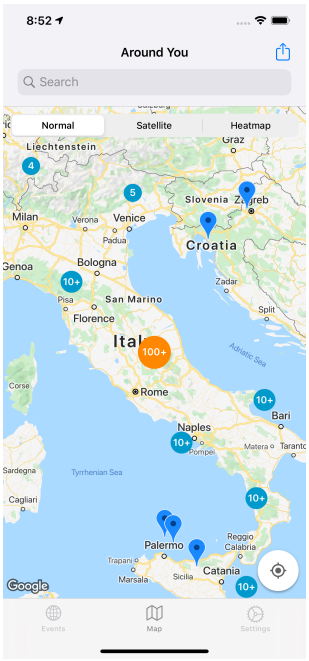


Figure 10: Map

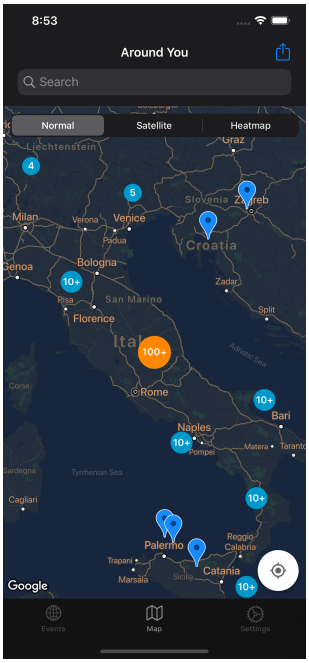


Figure 11: Map (Dark Mode)

The Map page is probably the most intuitive way to display events. To have a better visual effect, events are grouped once the map is not enough zoomed.
By tapping on a cluster, the map is automatically zoomed in. On the top right is located a share button which allows users to share a custom Safe Message with relatives and friends.
In this page are also available a search bar to move into the map in an easy way and a Segmented Control to change the map view from Normal to Satellite and Heatmap (see figures below).

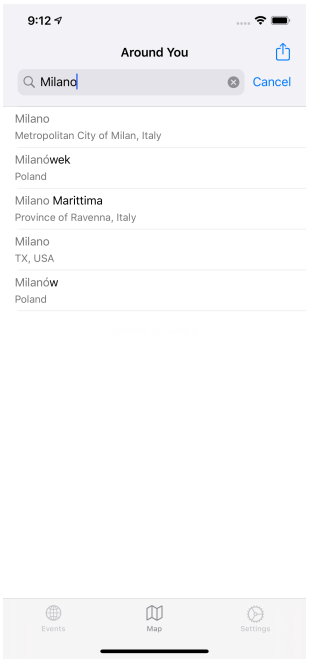


Figure 12: Google Places Search

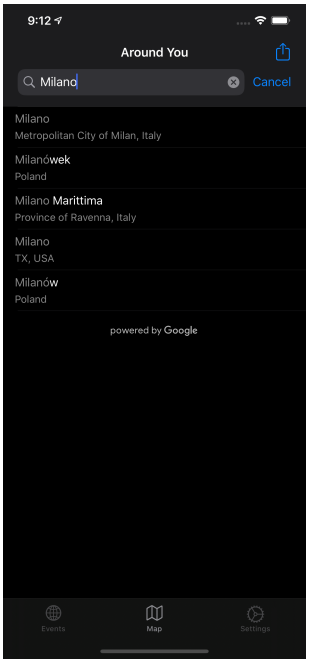


Figure 13: Google Places Search (Dark Mode)

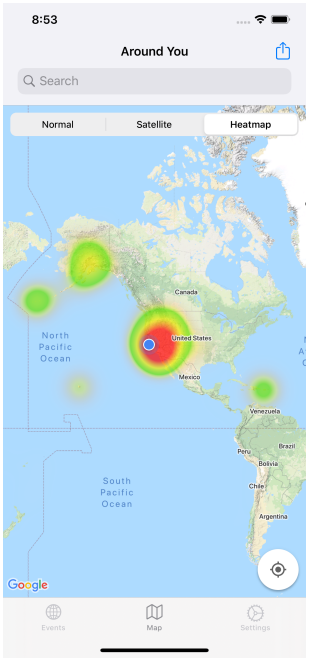


Figure 14: Heatmap

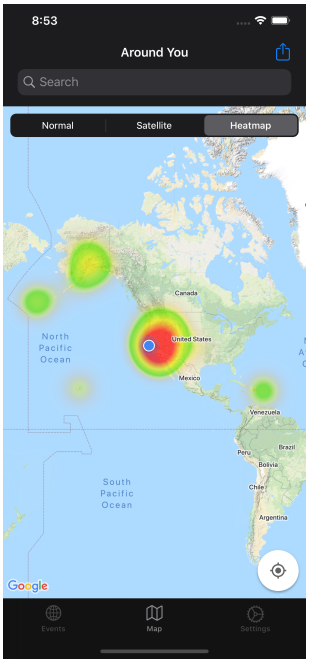


Figure 15: Heatmap (Dark Mode)

5.2.6 Settings

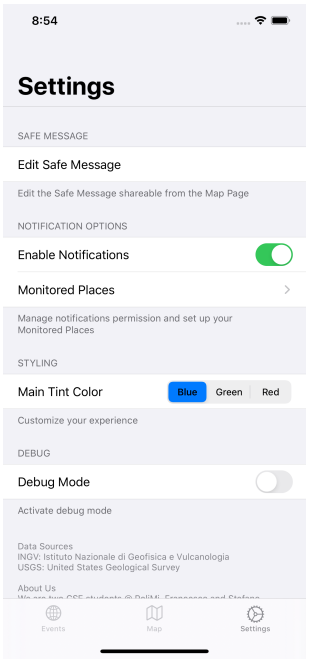


Figure 16: Settings

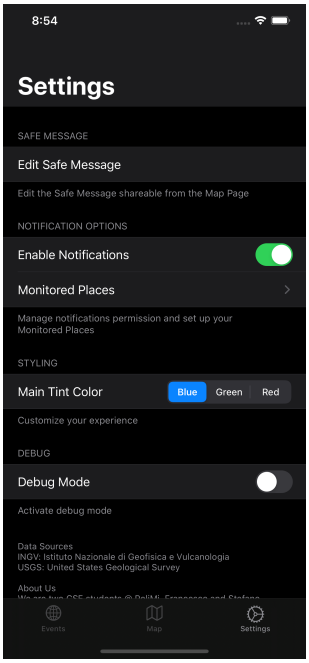


Figure 17: Settings (Dark Mode)

The Settings page contains all the settings of the application: the menu to edit the already mentioned Safe Message (see figures below), a toggle for enabling notifications and a segmented control to change the main tint color of the entire application.
The Monitored Places menu is where the user can add (or remove) places to be monitored and, when certain conditions are verified, be notified (see Monitored Places sub-section).
Only for demo purposes we added a Debug Mode toggle to trigger a special debug action (see Notifications section).

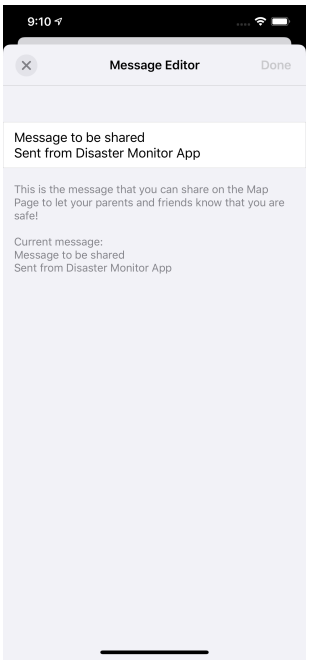


Figure 18: Safe Message Editor

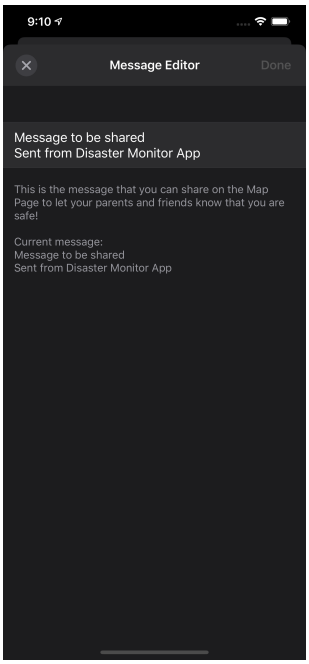


Figure 19: Safe Message Editor (Dark Mode)

5.2.7 Monitored Places

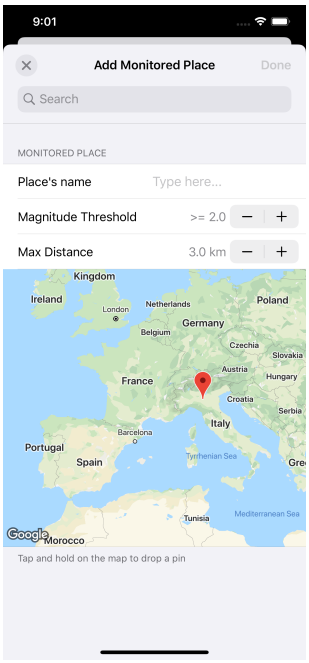


Figure 20: Add Monitored Place

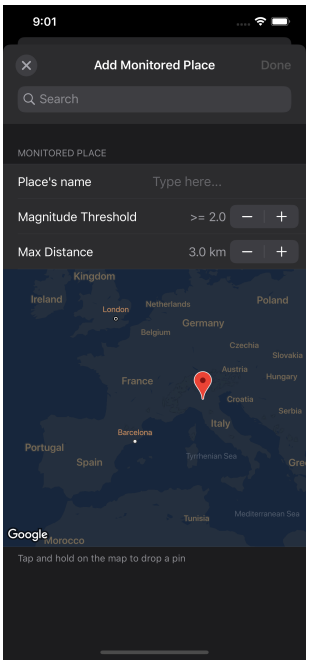


Figure 21: Add Monitored Place (Dark Mode)

In the Settings of the application, the user has the possibility to add a place to be monitored for seismic events: if a seismic event happens within a certain distance from the inserted position and with a magnitude greater or equal than a certain threshold, a local notification is scheduled to warn the user.
In the figures below, the list of monitored places added by a user.

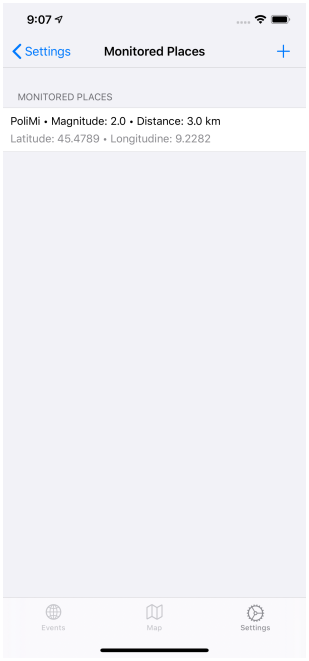


Figure 22: Monitored Places

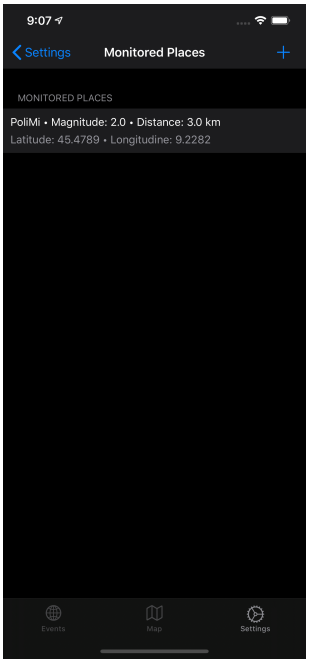


Figure 23: Monitored Places (Dark Mode)

6 Notifications

6.1 Local Notifications

Disaster Monitor supports Local Notifications to warn the user of seismic events.
On the first startup of the application, a popup is prompted on the screen asking permission to send notifications including alerts, sounds and icon badges.
If the user does not allow the app to send notifications, at each subsequent application startup a new popup is prompted remarking the fact that the notifications are turned off and, if the user taps the "Turn notifications on" button is redirected to the iOS Settings to eventually activate the notifications for Disaster Monitor (see figures below).

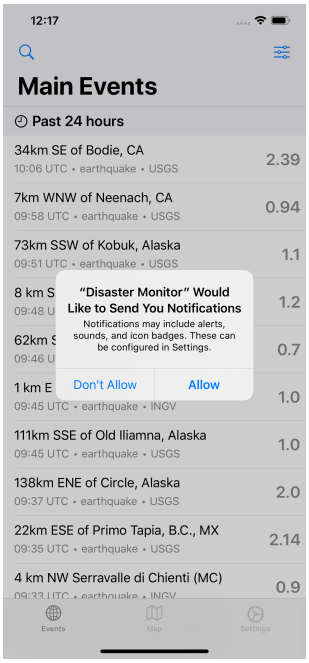


Figure 24: Notifications Authorization

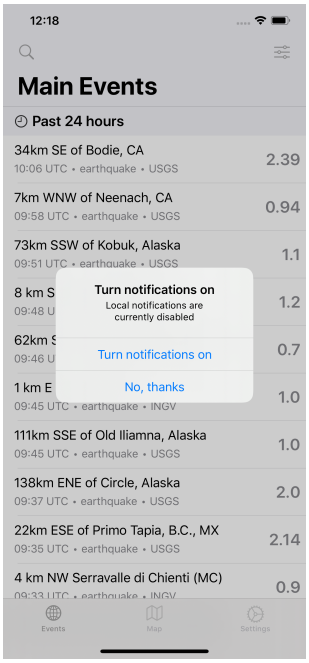


Figure 25: Notifications Authorization Denied

As stated before, in the application settings the user can access the Monitored Places menu where is possible to add (or remove) a location to be monitored by Disaster Monitor; in the Add Monitored Place page one can set, besides the place's name, a threshold for the magnitude of the seismic event and a max distance within which the notification should be scheduled.
After the user correctly insert a place to be monitored, every time the list of the seismic events is updated (on application startup or by a pull to refresh action in the Main Events page), every new entry is checked: if the current event happens inside the region entered by the user with a magnitude greater or equal than the one's set, a Local Notification is scheduled to appear on the screen after an interval of 5 seconds.
Notifications are displayed both when the application is in foreground and background (Lock screen included).
Lastly, the scheduling of local notifications also works during the background data fetching process.

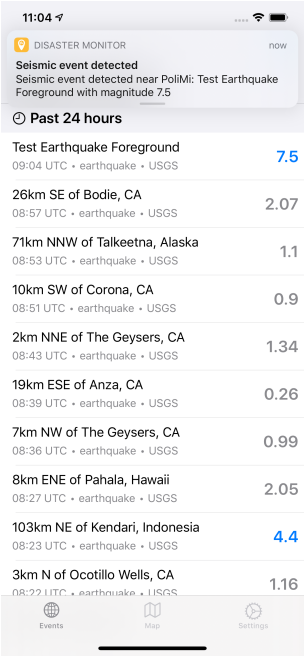


Figure 26: Foreground Notification

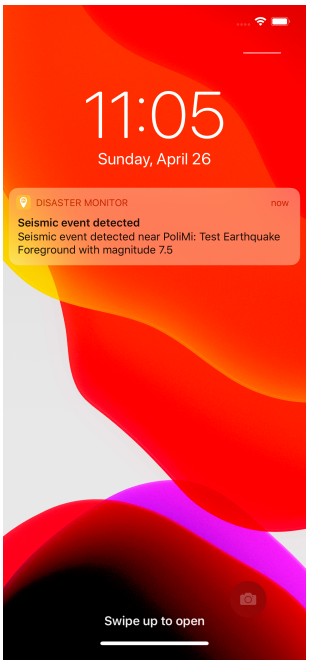


Figure 27: Foreground Notification (Lock screen)

6.2 Debug Mode

As introduced in the previous section, in the application settings there is a Debug Mode toggle: this toggle is meant to be used only for demo purposes and it only inserts in the list of events a fictitious "test event"; as it happens for "normal" events, this insertion will trigger the process of scheduling a Local Notification. The fictitious event is located at the Politecnico di Milano with a magnitude of 7.5 ML and so the local notification will be effectively displayed only if this location is present in the Monitored Places menu.

7 External services and libraries

Disaster Monitor uses different external services and libraries to obtain data to populate the application and to enrich the overall experience of the user.

7.1 Google Maps SDK for iOS[8]

Google Maps SDK adds maps based on Google Maps data to our application. The SDK automatically handles access to the Google Maps servers, map display and response to user gestures such as clicks and drags. Maps are widely present throughout the application: they are useful to present to the user the seismic events and their concentration around the globe through markers' clusterization and heat map.

7.2 Google Places SDK for iOS[9]

Google Places SDK gets data from the same database used by Google Maps. Places features over 100 million businesses and points of interest that are updated frequently through owner-verified listings and user-moderated contributions. We focused our attention on the Place Autocomplete feature: the autocomplete service returns place predictions in response to user search queries. As the user types, the autocomplete service returns suggestions for places.
This functionality is useful for the user for searching for a specific location in the Map Page and in the Monitored Places Page to search and add a location to be monitored.

7.3 Earthquake APIs

All the earthquakes information are loaded relying on the scientific agencies of two States: United States of America (by USGS (United States Geological Survey)[10]) and Italy (by INGV (Istituto Nazionale di Geofisica e Vulcanologia)[11]).
They both use a common format to provide data called GeoJSON, a JSON format for encoding a variety of geographic data structures.
Unfortunately, the JSON files provided by the APIs don't have the exactly same structure, and so, we had to implement a standardization layer to make them consistent with each other.

[8] Maps SDK for iOS: `https://developers.google.com/maps/documentation/ios-sdk/`
[9] Places SDK for iOS: `https://developers.google.com/places/ios-sdk/`
[10] USGS: `https://earthquake.usgs.gov/fdsnws/event/1/`
[11] INGV: `https://developers.italia.it/en/api/terremoti-opendata.html`

7.4 CocoaPods

CocoaPods is a dependency manager for Swift and Objective-C Cocoa projects. The following are some libraries offered by manager that we used in our application:

- Alamofire[12]: an HTTP networking library written in Swift;
- PromiseKit[13]: library that simplify asynchronous programming;
- SwiftyJSON[14]: library for managing JSON files (works great with Alamofire!);
- MarqueeLabel[15]: a drop-in replacement for UILabel, which automatically adds a scrolling marquee effect when needed.
- Google-Maps-iOS-Utils[16]: a utilities library for use with Google Maps SDK for iOS. This open-source library contains classes that are useful for a wide range of applications such as Marker clustering and Heatmaps;
- Quick[17]: a behavior-driven development framework for Swift and Objective-C;
- Nimble[18]: a Matcher Framework for Swift and Objective-C.

[12]Alamofire: `https://cocoapods.org/pods/Alamofire`
[13]PromiseKit: `https://cocoapods.org/pods/PromiseKit`
[14]SwiftyJSON: `https://cocoapods.org/pods/SwiftyJSON`
[15]MarqueeLabel: `https://cocoapods.org/pods/MarqueeLabel`
[16]Google-Maps-iOS-Utils: `https://cocoapods.org/pods/Google-Maps-iOS-Utils`
[17]Quick: `https://cocoapods.org/pods/Quick`
[18]Nimble: `https://cocoapods.org/pods/Nimble`

8 Testing

8.1 Test cases

After the implementation, all the significant logic components of Disaster Monitor need to be tested: since the application does not contain particular algorithms, testing only the State Updaters seems to be a reasonable choice; as said before, State Updaters are the only way to update the entire state of the application. Most of the State Updaters are simple boolean variables modifiers so the attention needs to be put on, for example, the *Events*, principal elements of the application: their insertion, cancellation and update must be verified by means of Unit Testing.
Two other main elements of Disaster Monitor are the *Monitored Places* and the *Safe Message* to be shared with parents and friends: also these two components need to be tested in the same way as the Events.

8.2 Background Tasks during development

The delay between the time an iOS application schedules a background task and when the system launches the app to run the task can be many hours. While developing, it is possible to use two private functions to start a task and to force early termination of the task according to the selected timeline.
The debug functions work only on devices.
For more information on how to debug Background Tasks, please refer to the official documentation[19].

[19]Starting and Terminating Tasks During Development: `https://developer.apple.com/documentation/backgroundtasks/starting_and_terminating_tasks_during_development`

9 Effort spent

Considering all the design phases, modelling and learning of the new technologies, we report below the estimated time spent on this project:

Component	Learning time	Designing time	Tutoring time	Development time	Total
Stefano Martina	15 hours	30 hours	4 hours	200 hours	249 hours
Francesco Peressini	15 hours	30 hours	4 hours	200 hours	249 hours
Total	30 hours	60 hours	8 hours	400 hours	498 hours

Info: Please note that the above hours are just estimation, we did not use any kind of tracking application for this purpose.