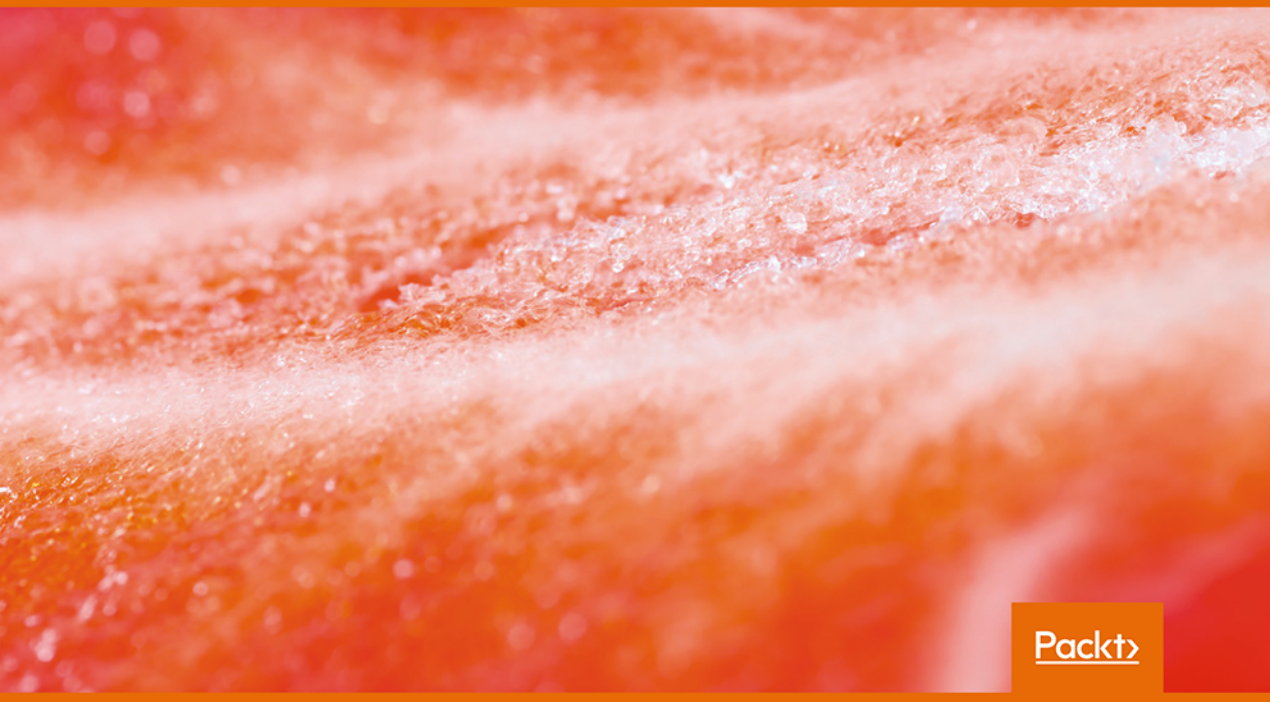
Helion

Poznaj Swifta, tworząc aplikacje

Profesjonalne projekty dla systemu iOS



Packt

Emil Atanasov

Tytuł oryginału: Learn Swift by Building Applications: Explore Swift programming through iOS app development

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-5453-1

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/poswif.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/poswif
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 7

Tworzenie prostej aplikacji prognozy pogody

W rozdziale opracujesz aplikację o nazwie *Weather*. Zaczniesz od zdefiniowania wszystkich ekranów w pliku Storyboard, a następnie przejdziesz do otwierania różnych kontrolerów widoku i tworzenia między nimi animowanych przejść. Dowiesz się również, jak definiować ograniczenia i przekazywać informacje między dwoma kontrolerami. Oto krótka lista tematów poruszonych w rozdziale:

- projektowanie ekranów aplikacji w pliku Storyboard,
- definiowanie modelu,
- wyświetlenie różnych ekranów po naciśnięciu przycisku,
- przekazywanie danych między kontrolerami widoku,
- definiowanie ograniczeń i używanie funkcji Auto Layout,
- przypomnienie informacji na temat `UITableView` i `UICollectionView`.

Tworzenie aplikacji rozpoczynamy od zdefiniowania jej przeznaczenia i wyglądu.

Definiowanie ekranów aplikacji

Działanie aplikacji Weather polega na wyświetleniu tygodniowej prognozy pogody dla danej lokalizacji. Najpierw opracujesz wersję działającą dla jednego miasta, a następnie rozbudujesz ją o możliwość pobierania danych z internetu. Dlatego też tutaj wymieniłem ekrany, które zostaną opracowane w tym rozdziale.

- **Ekran wczytywania aplikacji**. Każda aplikacja ma ekran początkowy wyświetlany w trakcie jej wczytywania. Domyślnie każdy projekt zawiera oddzielny plik Storyboard pozwalający na zdefiniowanie tego ekranu. W budowanej aplikacji zastosujesz bardzo proste rozwiązanie dla ekranu wczytywania aplikacji, które będzie można później zmodyfikować.
- **Główny ekran prognozy pogody**. Ta scena to punkt wyjścia, na niej będzie wyświetlana prognoza pogody dla danej lokalizacji. Po dodaniu obsługi dla wielu lokalizacji ten ekran powinien zapewnić łatwą nawigację między lokalizacjami. Przykładowo aplikacja prognozy pogody standardowo dostarczana wraz z systemem iOS pozwala na użycie gestu przewinięcia w prawo i w lewo do przechodzenia między lokalizacjami. Po opracowaniu podstawowej funkcjonalności aplikacji zaimplementujesz podobne rozwiązanie.
- **Ekran ulubionych lokalizacji**. Ten ekran wyświetla listę wszystkich ulubionych lokalizacji użytkownika. Będą one łatwo dostępne z poziomu ekranu głównego, a dla wybranej lokalizacji zostanie wyświetlona prognoza pogody.
- **Ekran dodawania lokalizacji**. Użytkownik powinien mieć możliwość wybrania miasta z listy. Wybrana lokalizacja stanie się elementem na ekranie ulubionych lokalizacji.

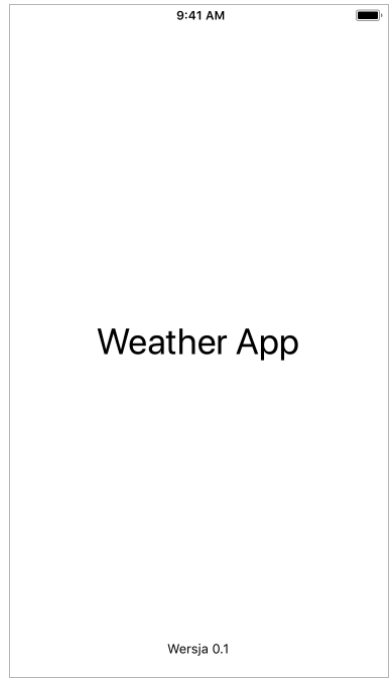
Na początku liczba ekranów jest dość ograniczona. Na dalszym etapie pracy dodasz kolejne, aby poprawić użyteczność aplikacji. Rozpoczniesz od utworzenia bardzo prostej aplikacji, a następnie będziesz po kolei dodawać poszczególne funkcjonalności i jednocześnie poznawać specyfikę systemu iOS.

Spójrz na początkowe prototypy, które później zostaną użyte podczas definiowania w pliku Storyboard scen i związków między nimi.

Na rysunku 7.1 pokazałem ekran wczytywania aplikacji wraz z minimalną liczbą informacji dotyczących programu. Jeżeli chcesz go upiększyć, możesz dodać obrazy przedstawiające np. słońce, deszcz i śnieg.

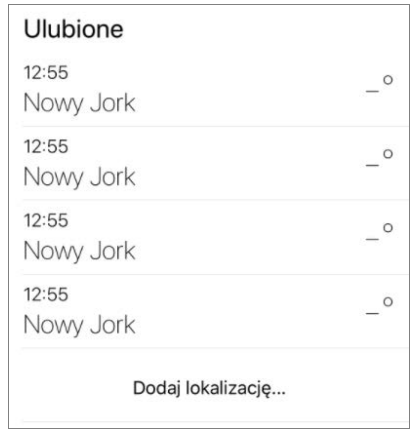
Na rysunku 7.2 możesz zobaczyć ekran główny aplikacji wyświetlający bieżącą datę i temperaturę oraz nazwę lokalizacji. Dalej znajdują się temperatury na kolejne 12 godzin, a niżej prognoza na kolejne 5 – 7 dni. Dobrym pomysłem jest użycie różnych obrazów i ikon odzwierciedlających warunki atmosferyczne. Potrzebny jest również przycisk pozwalający przejść do ekranu ulubionych lokalizacji, pokazanego na rysunku 7.3.

Lista ulubionych lokalizacji przechowuje wszystkie te, które zostały wybrane przez użytkownika. Listę można rozbudować po naciśnięciu przycisku *Dodaj lokalizację…*. Naciśnięcie tego przycisku powinno spowodować wyświetlenie **ekranu dodawania lokalizacji**, który zostanie zdefiniowany jako kolejny. Każdy element na liście może zostać usunięty, jeśli użytkownik nie potrzebuje danej lokalizacji. W przypadku braku jakiejkolwiek lokalizacji na liście użyta będzie domyślna. W budowanej aplikacji lokalizacją domyślną jest **Nowy Jork**. Na rysunku 7.4 możesz zobaczyć ekran dodawania lokalizacji.

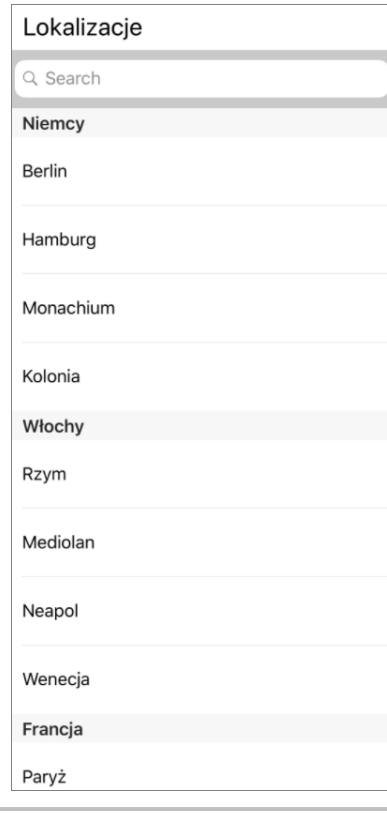


Rysunek 7.1. Ekran wczytywania aplikacji

Rysunek 7.2. Ekran główny aplikacji



Rysunek 7.3. Ekran ulubionych lokalizacji



Rysunek 7.4. Ekran dodawania lokalizacji

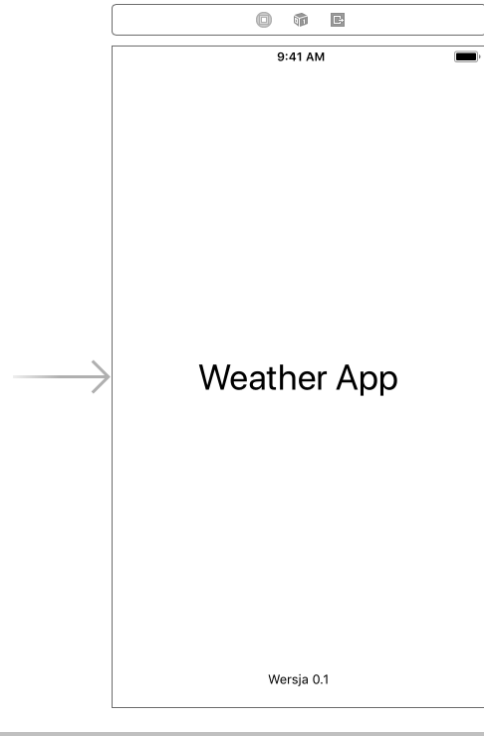
Ekran pokazany na rysunku 7.4 wyświetla listę wszystkich lokalizacji dostępnych w programie i oferuje funkcję wyszukiwania ułatwiającą poruszanie się po danych. Lokalizacja po wybraniu zostanie dodana do listy **ulubionych**.

Pracę trzeba rozpocząć od utworzenia projektu aplikacji iOS na podstawie szablonu *Single View App*. Budowana tutaj aplikacja będzie miała więcej niż tylko jeden ekran; dalej w tym rozdziale dowiesz się, jak można je wyświetlać. Zdefiniujesz też niestandardowe przejścia między poszczególnymi egzemplarzami `ViewController`, które powinny być wyświetlane na ekranie.

Utwórz nową aplikację o nazwie *Weather* i jako jej domyślny język programowania wybierz Swift. Aplikacja Xcode jest bardzo pomocna i generuje wszystkie niezbędne pliki startowe projektu. Przejdź do pliku *Main.storyboard*, w którym opracujesz poszczególne sceny. Następnym krokiem będzie ich połączenie i utworzenie w pełni funkcjonalnej aplikacji.

> Informacje na temat tworzenia nowego projektu aplikacji iOS przedstawiłem w rozdziale 5. Wykonaj omówione w nim kroki, a otrzymasz nowy projekt wraz z lokalnym repozytorium systemu kontroli wersji Git, które będzie można udostępnić np. w serwisie GitHub. Nie zapominaj o przekazywaniu od czasu do czasu zmian do repozytorium, aby zapisywać informacje o postępie pracy nad aplikacją.

Zanim przejdziesz do zagłębienia się w szczegóły dotyczące aplikacji, na rozgrzewkę przygotujesz jej ekran początkowy. To jest pierwszy ekran wyświetlany po uruchomieniu aplikacji. Pełną kontrolę nad tym ekranem masz za pomocą pliku *LaunchScreen.storyboard*. Otwórz ten plik i dodaj dwie etykiety. Pierwsza powinna zostać umieszczona na środku ekranu, natomiast druga na dole. Ostateczny wygląd tego ekranu pokazałem na rysunku 7.5.

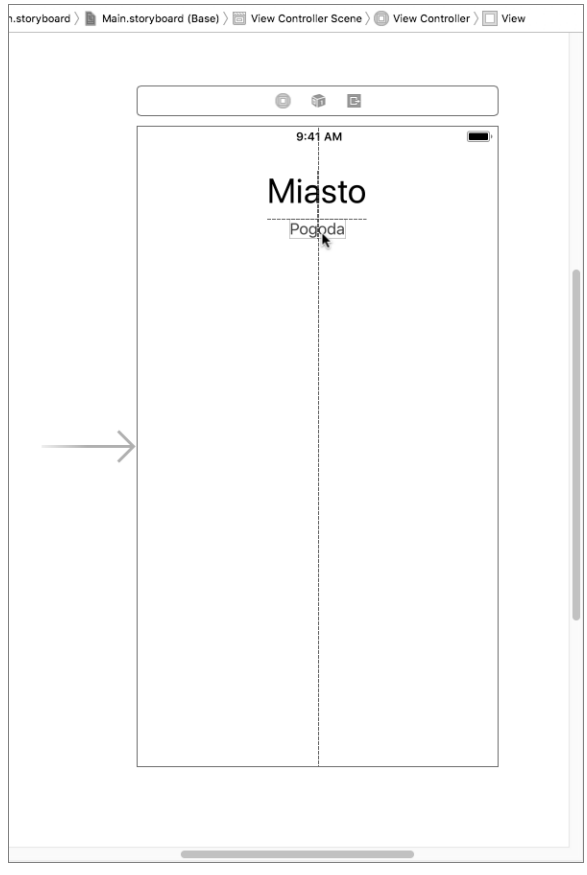


Rysunek 7.5. Ekran początkowy aplikacji zdefiniowany w pliku LaunchScreen.storyboard

W następnym punkcie przedstawię utworzenie sceny ekranu głównego. Nazwałem ją tak, ponieważ użytkownik będzie widział ten ekran po każdym uruchomieniu aplikacji.

Ekran główny aplikacji

Po uruchomieniu pustego projektu na ekranie będzie widoczny jeden kontroler widoku. Dodasz teraz kluczowe funkcje z prototypu, aby przygotować ekran główny aplikacji. Rozpocznij od nazwy lokalizacji i prognozy pogody. Na rysunku 7.6 możesz zobaczyć wynik.



Rysunek 7.6. Aktualna postać ekranu głównego

Aby otrzymać tę postać, należy użyć dwóch etykiet i zmienić w nich wielkość czcionki. Następnie trzeba odpowiednio rozmieścić te etykiety.

> Dalej w tym rozdziale dowiesz się, jak utworzyć układ elastyczny, który będzie elegancko wyświetlany na ekranach smartfonów o różnej wielkości. W tym momencie masz skoncentrować się na standardowych urządzeniach iPhone 6/7/8.

Przechodzimy do wykonania wymienionych tutaj kroków.

1. Potrzebna jest etykieta wyświetlająca temperaturę. Ta etykieta powinna być wyśrodkowana. Potrzebny jest również znak oznaczający temperaturę. Aby go dodać, należy wybrać opcję menu *Edit/Emoji & Symbols*, a następnie w wyświetlonym na górze polu wyszukiwania wpisać `degree`. Po dodaniu nowej etykiety ekran powinien wyglądać tak, jak pokazałem na rysunku 7.7.

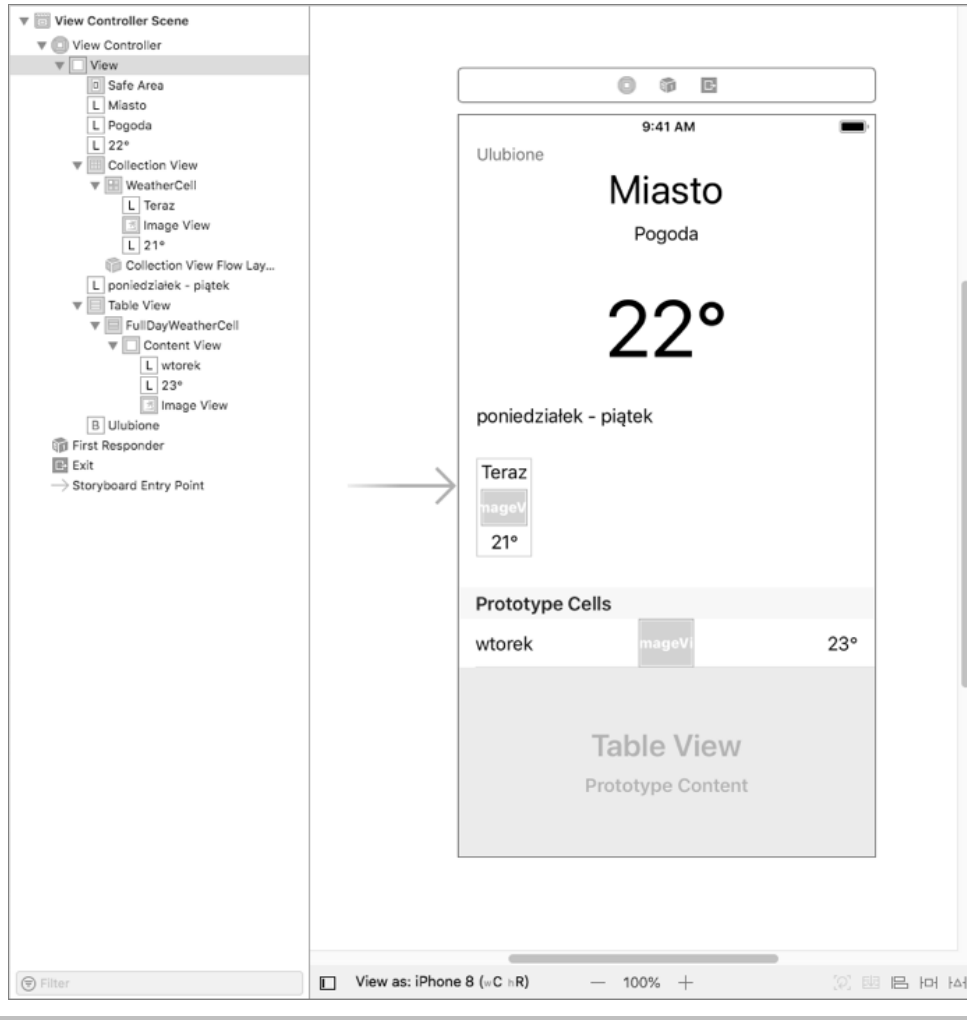
Rysunek 7.7. Ekran główny po dodaniu etykiety temperatury

2. Następnym krokiem jest dodanie kontrolki `UICollectionView`, która będzie wyświetlała prognozę pogody na kolejne godziny. Po przeciągnięciu `UICollectionView` wskazujesz miejsce wygenerowania danych, ale dla każdego elementu tych danych należy dostarczyć komórkę, która jest potrzebna do wyświetlania informacji. Komórka powinna mieć etykietę wyświetlającą godzinę i obraz pokazujący aktualne warunki atmosferyczne. To pozwoli na zaoszczędzenie miejsca i dodanie kolejnej etykiety wyświetlającej temperaturę. Spójrz na rysunek 7.8.

Rysunek 7.8. Kontrolka UICollectionView i jej komórka

Na obecnym etapie pracy interfejs użytkownika nie jest doskonały, poprawisz go nieco później. Teraz najważniejsze jest połączenie wszystkich najważniejszych komponentów aplikacji. Na jej ostatecznym wyglądzie i sposobie działania skoncentrujesz się na dalszym etapie pracy.

3. Nie zapomnij o ustawieniu w kontrolce `UICollectionView` poziomego przewijania elementów (opcja *Horizontal* w rozwijanym menu *Scroll Direction*). Możesz to zrobić za pomocą *inspektora Attributes*, który jest wyświetlany na czwartej karcie od lewej strony w panelu narzędziowym.
4. Prognoza pogody, choć już nie tak szczegółowa, ma zostać wyświetlona dla pięciu kolejnych dni. Do pokazania tych informacji wykorzystasz kontrolkę `UITableView`. Po jej dodaniu ekran powinien wyglądać tak, jak pokazałem na rysunku 7.9.



Rysunek 7.9. Kontrolka UITableView dodana do sceny

5. Ostatnim krokiem jest dodanie przycisku o nazwie *Ulubione* pozwalającego na wybór innej lokalizacji. Później dodasz obsługę listy ulubionych lokalizacji użytkownika, do których będzie miał łatwy dostęp. Na razie aplikacja będzie jak najprostsza.

   Wszystkie elementy można zobaczyć w panelu *Document Outline* po lewej stronie pliku Storyboard. Aby wyświetlić ten panel, należy kliknąć przycisk  .

> Do wyświetlenia lub ukrycia panelu *Document Outline* służy pokazany wcześniej przycisk, który znajduje się na dole okna Xcode.

W ten sposób w pliku Storyboard utworzyłeś ekran główny aplikacji. Przedstawiony tutaj proces polegał na konwersji początkowego pomysłu na elementy graficzne. Teraz przejdziesz do utworzenia pozostałych ekranów aplikacji. Natomiast później połączysz je ze sobą i dodasz pewne dane, aby przygotowane komponenty można było zobaczyć w akcji.

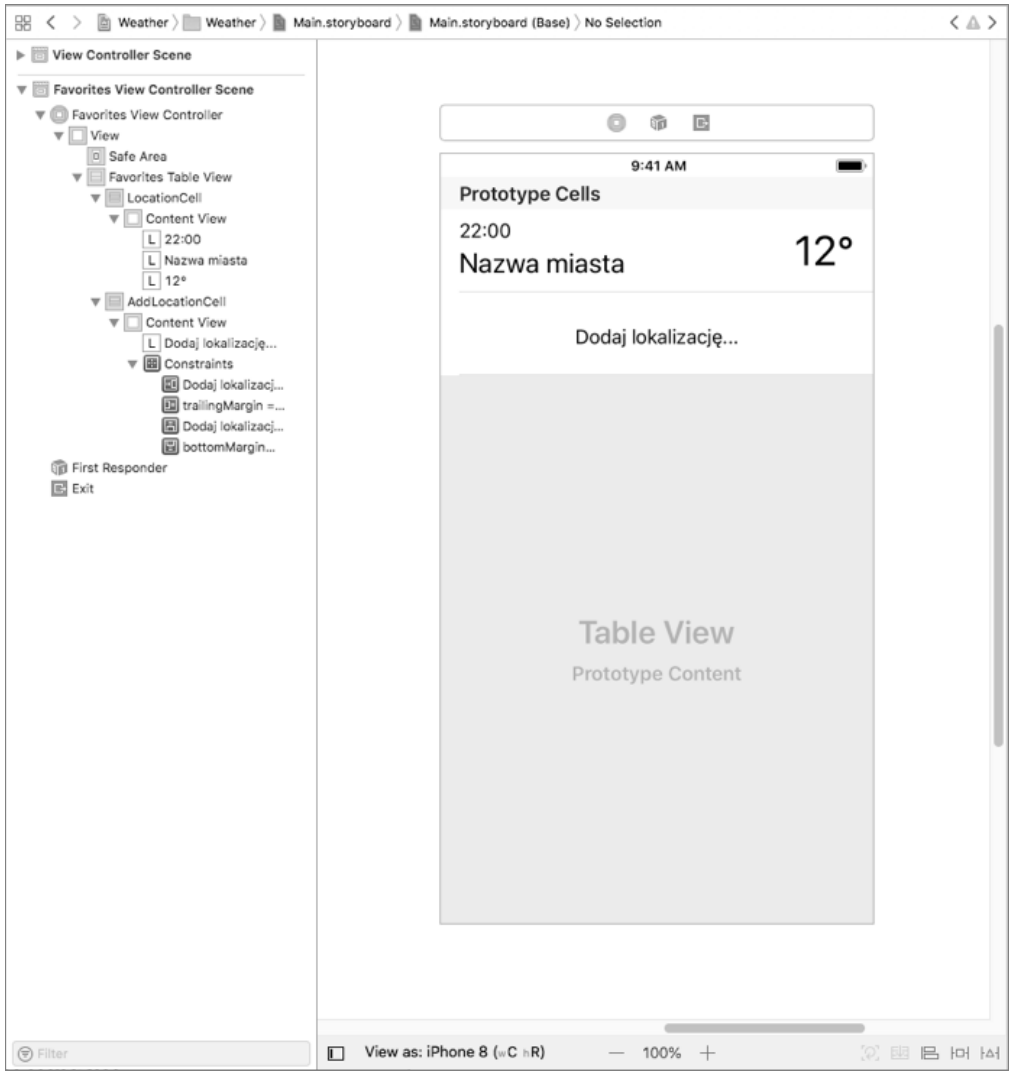
Najpierw skoncentrujesz się na ekranie ulubionych lokalizacji. To jest jedyny ekran, który może zostać wyświetlony bezpośrednio z poziomu ekranu głównego.

Ekran ulubionych lokalizacji

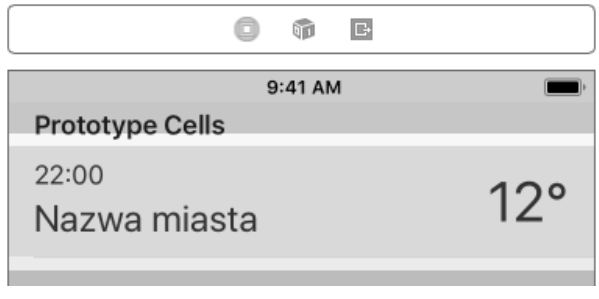
Na tym ekranie zostaną wyświetlone wszystkie ulubione lokalizacje użytkownika. Trzeba będzie zdefiniować kolekcję pozwalającą na pokazanie tych lokalizacji. Lista lokalizacji musi być zachowana po zakończeniu działania aplikacji. Ponadto użytkownik powinien mieć możliwość dodawania i usuwania elementów z listy. Element znajdujący się na początku listy będzie wskazywał lokalizację, dla której ekran główny wyświetla prognozę pogody. Ostateczną postać interfejsu użytkownika koniecznego do zaimplementowania możesz zobaczyć na rysunku 7.10.

Oto krótkie wyjaśnienie kroków, które trzeba wykonać, aby przygotować ten ekran. Pracę trzeba zacząć od dodania nowego kontrolera widoku do pliku Storyboard. Odszukaj go w **bibliotece obiektów**, a następnie przeciągnij i upuść obok istniejącego. Powinieneś już dość swobodnie poruszać się po interfejsie Xcode, ponieważ dokładnie omówiłem go w rozdziale 2. Jeżeli tak nie jest, poświęć nieco czasu na poznanie środowiska IDE i jego najważniejszych elementów. Potrzebny jest widok tabeli (`UITableView`) wypełniający cały ekran. Następnie musisz utworzyć dwa odmienne typy `UICellView`. Pierwszy jest używany do wyświetlenia listy wszystkich ulubionych lokalizacji, natomiast drugi będzie wykorzystany w charakterze przycisku wyświetlającego ekran, na którym można dodać kolejne lokalizacje do listy ulubionych.

Pierwszy egzemplarz `UICellView` powinien być już utworzony w `UITableView`. Potrzebujemy trzech etykiet — pierwsza wyświetla bieżącą godzinę w danej lokalizacji, druga nazwę lokalizacji, zaś trzecia bieżącą temperaturę w tej lokalizacji. Gotową komórkę pokazałem na rysunku 7.11.



Rysunek 7.10. Gotowy ekran ulubionych lokalizacji utworzony w pliku Storyboard

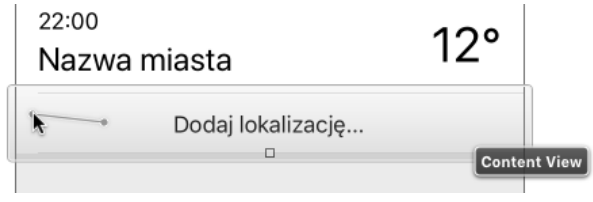


Rysunek 7.11. Ukończona pierwsza komórka na ekranie ulubionych lokalizacji

Druga komórka jest bardzo prosta. Zacznij od dodania nowej kontrolki `UICellView` do tabeli. Następnie w komórce umieść jedną etykietę, która powinna być w niej wyśrodkowana. Ten efekt można osiągnąć za pomocą ograniczeń — czyli zdefiniowanych reguł, których celem jest ułatwienie ułożenia komponentów (egzemplarzy `UIView`) na ekranach o różnych wielkościach.

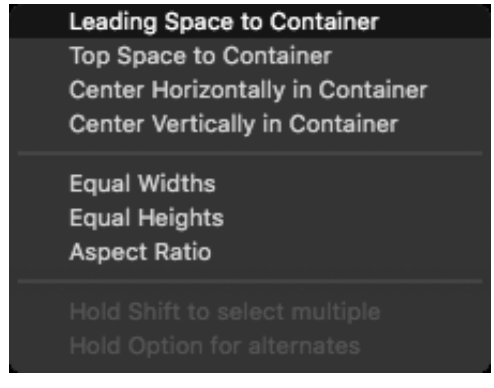
Ograniczenia

Mamy kilka sposobów na definiowanie ograniczeń w module *Interface Builder* aplikacji Xcode. Jeden z nich polega na użyciu myszy i klawisza *Ctrl* do przeciągnięcia myszą z jednego widoku do drugiego. Tym samym masz pewną kontrolę i możesz dodawać ograniczenia pojedynczo. Przykład takiego podejścia pokazałem na rysunku 7.12.



Rysunek 7.12. Definiowanie ograniczenia przez przeciągnięcie myszą między widokami

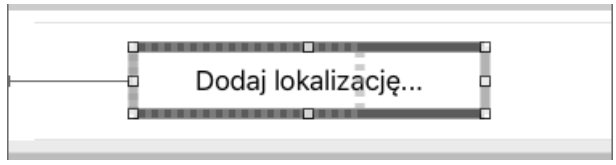
W ten sposób można zdefiniować ograniczenie *Leading Space*; należy je wybrać z panelu wyświetlonego po zwolnieniu przycisku myszy (zobacz rysunek 7.13).



Rysunek 7.13. Opcje dostępne podczas definiowania ograniczenia za pomocą myszy

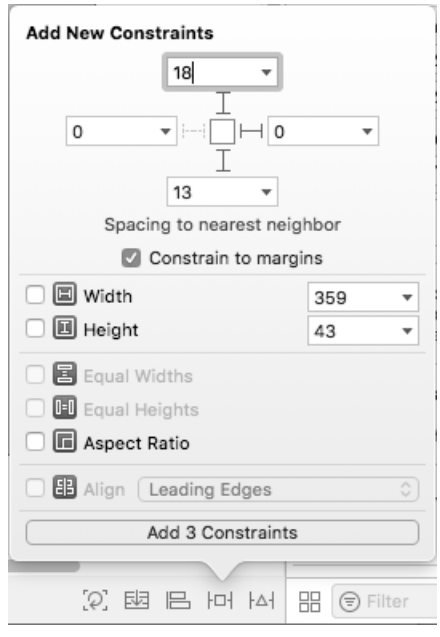
Na rysunku 7.10 wcześniej w tym rozdziale widać zdefiniowane wszystkie ograniczenia, choć musisz je dodać ręcznie. Nie panikuj, jeśli widzisz pewne czerwone linie w widoku (zobacz rysunek 7.14) — oznaczają one brakujące ograniczenia. Po dodaniu wszystkich czerwone linie znikną.

Inną możliwością w zakresie definiowania ograniczeń jest użycie przydatnego menu wyświetlanego po kliknięciu ikony znajdującej się na dole okna Xcode.



Rysunek 7.14. Czerwone linie oznaczające brakujące ograniczenia

Najpierw zaznacz widok, a następnie kliknij przycisk . Na ekranie powinieneś zobaczyć okno pokazane na rysunku 7.15.



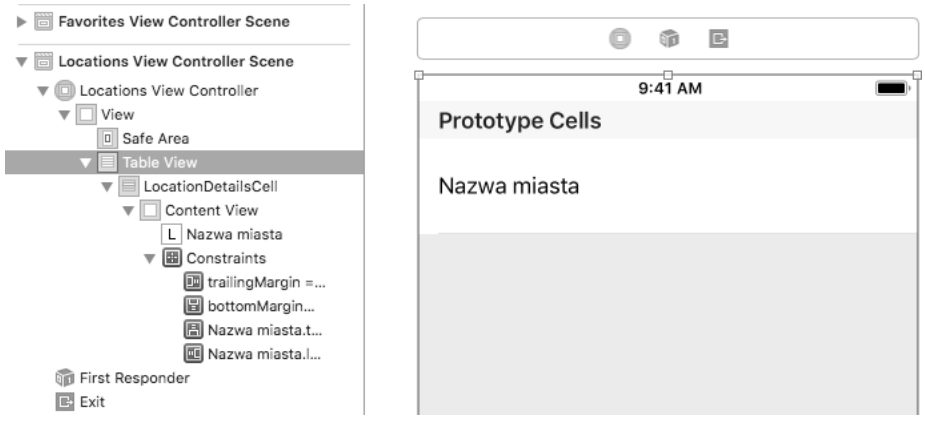
Rysunek 7.15. Okno pozwalające na zdefiniowanie ograniczeń dla widoku

Czerwone linie w tym oknie pokazują ograniczenia definiowane względem widoku nadrzędnego. Liczby w polach określają odległość od poszczególnych krawędzi. Kliknięcie przycisku na dole okna, tutaj *Add 3 Constraints*, spowoduje dodanie wszystkich zdefiniowanych ograniczeń. Kolejny ekran pozwalający na wybór lokalizacji dodawanej do listy ulubionych jest bardzo podobny do utworzonego w tym punkcie.

Ekran wyboru lokalizacji

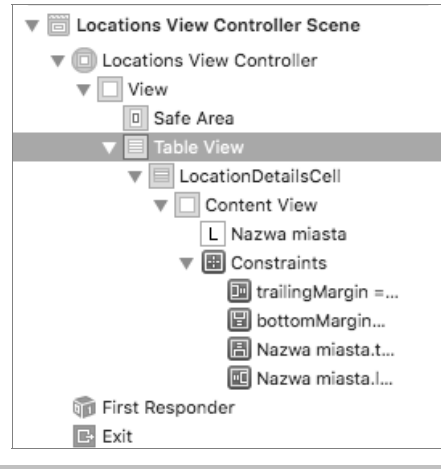
Aplikacja Weather powinna umożliwić użytkownikowi wybór lokalizacji i sprawdzić, czy dla niej można pobrać prognozę pogody. Aby zachować atrakcyjność omawianego tutaj przykładu, na początku udostępnisz użytkownikowi predefiniowaną listę kilkunastu miast. Dalej w książce poznasz sposób na rozbudowę tej funkcjonalności i wczytywanie wielu innych lokalizacji. Jednak w tym momencie skoncentrujesz się na prostszej wersji aplikacji.

Potrzebna jest kolekcja lokalizacji przeznaczonych do wyświetlenia. Mile widziana jest również możliwość wyszukiwania miast na liście dostępnych. To powinno przypomnieć Ci przykład z poprzedniego rozdziału, w którym został utworzony widok tabeli wraz z funkcją wyszukiwania. Tę funkcjonalność odtworzysz teraz w aplikacji Weather. Na rysunku 7.16 pokazałem przykładowy interfejs użytkownika; jeśli chcesz, możesz przygotować znacznie bardziej rozbudowany.



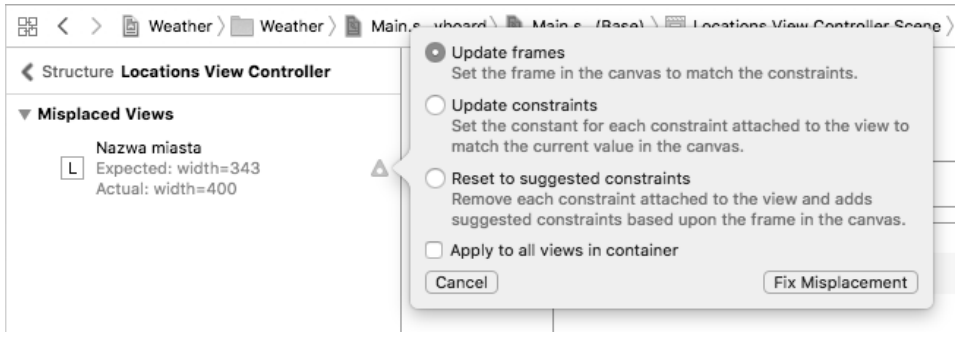
Rysunek 7.16. Interfejs użytkownika ekranu wyboru lokalizacji

Na scenie znajduje się kontrolka `UITableView` wypełniająca cały ekran. Zdefiniowałem też pewne ograniczenia. Czasami po zdefiniowaniu ograniczeń możesz zobaczyć żółtą ikonę ostrzeżenia w panelu Document Outline (po prawej stronie obszaru roboczego w pliku Storyboard). Przykład takiej ikony dla drzewa układu *Locations View Controller* pokazałem na rysunku 7.17.



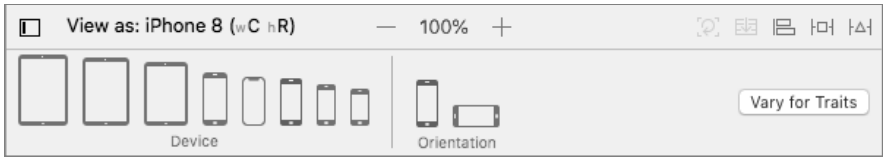
Rysunek 7.17. Ikona ostrzeżenia w drzewie układu Locations View Controller

Nie panikuj! Kliknij tę ikonę, a następnie podpowiedzi Xcode wykorzystaj do rozwiązania problemu i uaktualnienia wszystkich ograniczeń, tak jak pokazałem na rysunku 7.18.



Rysunek 7.18. Podpowiedzi Xcode dotyczące poprawienia ograniczeń

Modułu Interface Builder w Xcode można użyć do sprawdzenia wyglądu interfejsu użytkownika na ekranach o różnych wielkościach (zobacz rysunek 7.19). To jest dobry krok pozwalający na potwierdzenie poprawności zdefiniowanych ograniczeń. Ponadto możesz sprawdzić wygląd aplikacji na różnych ekranach bez konieczności jej rzeczywistego uruchamiania w poszczególnych symulatorach lub urządzeniach.



Rysunek 7.19. Funkcja Xcode pozwalająca sprawdzić wygląd interfejsu użytkownika na ekranach o różnych wielkościach

Wystarczy wybrać odpowiednią wielkość ekranu za pomocą menu wyświetlanego na dole ekranu modułu Interface Builder w oknie Xcode.

> Zawsze należy dokładnie sprawdzić wygląd i sposób działania aplikacji w zarówno symulatorze, jak i rzeczywistym urządzeniu. Dzięki temu można wychwycić ewentualne niedociągnięcia, które umknęły w trakcie opracowywania aplikacji.

Wersja aplikacji ma znaczenie podczas kompilowania wersji przeznaczonej do wydania, która następnie przechodzi proces kontroli jakości. Po kompilacji zwykle warto sprawdzić wersję aplikacji uruchomionej (testowanej) w urządzeniu. Dlatego też dodanie etykiety wyświetlającej wersję aplikacji jest eleganckim rozwiązaniem, które dodatkowo ułatwia proces testowania.

Przystąpisz teraz do zdefiniowania modelu spełniającego wymagania aplikacji. Dalej w książce usprawnisz go i przystosujesz do działania z zewnętrznymi danymi pochodzącymi z serwera dostarczającego dane prognozy pogody.

Model

Aplikacja potrzebuje modelu przechowującego wszystkie informacje o ekranie głównym. Ten model powinien zawierać dane o lokalizacji, aktualnej pogodzie i dokładnej prognozie pogody dla każdej godziny aż do końca dnia. Ponadto powinien zawierać prognozę pogody dla kilku kolejnych dni. Jak widzisz, będzie to bardzo prosty model. Definiujące go klasy muszą spełniać wymienione wymagania. Zamiast klasy istnieje możliwość użycia struktury wraz z różnymi polami, wybór należy do Ciebie. Na tym polega piękno tworzenia oprogramowania — wiele różnych klas, struktur i abstrakcji może prowadzić do otrzymania dokładnie tego samego wyniku.

```
public struct Location {
    var name: String
}
public class Forecast {
    var date:Date
    var weather:String = "brak"
    var temperature = 100
    public init(date:Date, weather: String, temperature: Int) {
        self.date = date
        self.weather = weather
        self.temperature = temperature
    }
}
public class DailyForecast : Forecast {
    var isWholeDay = false
    var minTemp = -100
    var maxTemp = 100
}
public class LocationForecast {
    var location:Location?
    var weather:String?
    var forecastForToday:[Forecast]?
    var forecastForNextDays:[DailyForecast]?
    // Utworzenie przykładowych danych i ich wyświetlenie w interfejsie użytkownika.
    static func getTestData() -> LocationForecast {
        let aMinute = 60
        let location = Location(name: "Nowy Jork")
        let forecast = LocationForecast()
        forecast.location = location
        forecast.weather = "słonecznie"
        // Dzisiaj.
        let today = Date().midnight
        var detailedForecast:[Forecast] = []
        for i in 0...23 {
            detailedForecast.append(Forecast(
              date: today.addingTimeInterval(TimeInterval(
                60 * aMinute * i)), weather: "słonecznie",temperature: 25))
        }
        forecast.forecastForToday = detailedForecast
        let tomorrow = DailyForecast(date: today.tomorrow,
                                     weather: "słonecznie",
                                     temperature: 25)
```

```
            tomorrow.isWholeDay = true
            tomorrow.minTemp = 23
            tomorrow.maxTemp = 27
            let afterTomorrow = DailyForecast(date: tomorrow.date.tomorrow,
                                              weather: "częściowe zachmurzenie",
                                              temperature: 25)
            afterTomorrow.isWholeDay = true
            afterTomorrow.minTemp = 24
            afterTomorrow.maxTemp = 28
            forecast.forecastForNextDays = [tomorrow, afterTomorrow]
            return forecast
        }
    }
```

W projekcie musisz utworzyć plik o nazwie *LocationForecast.swift* i zdefiniować w nim modele.

> Jeżeli zdecydujesz się na użycie innych pól lub nazw struktur bądź klas, powinieneś odpowiednio uaktualnić kod przedstawiony dalej w tym rozdziale.

Utworzona została metoda o nazwie `getTestData()` zwracająca dane testowe, które będą użyte na ekranie głównym aplikacji do sprawdzenia, czy wszystko działa zgodnie z oczekiwaniami. Można teraz uruchomić aplikację i przekonać się, że ekran główny jest prawie pusty — zawiera jedynie dane domyślne. Aby dodać pewne dane do `UICollectionView` lub `UITableView`, konieczne jest zdefiniowanie właściwości `datasource` i przypisanie jej wartości w postaci klasy specjalnej implementującej określony interfejs.

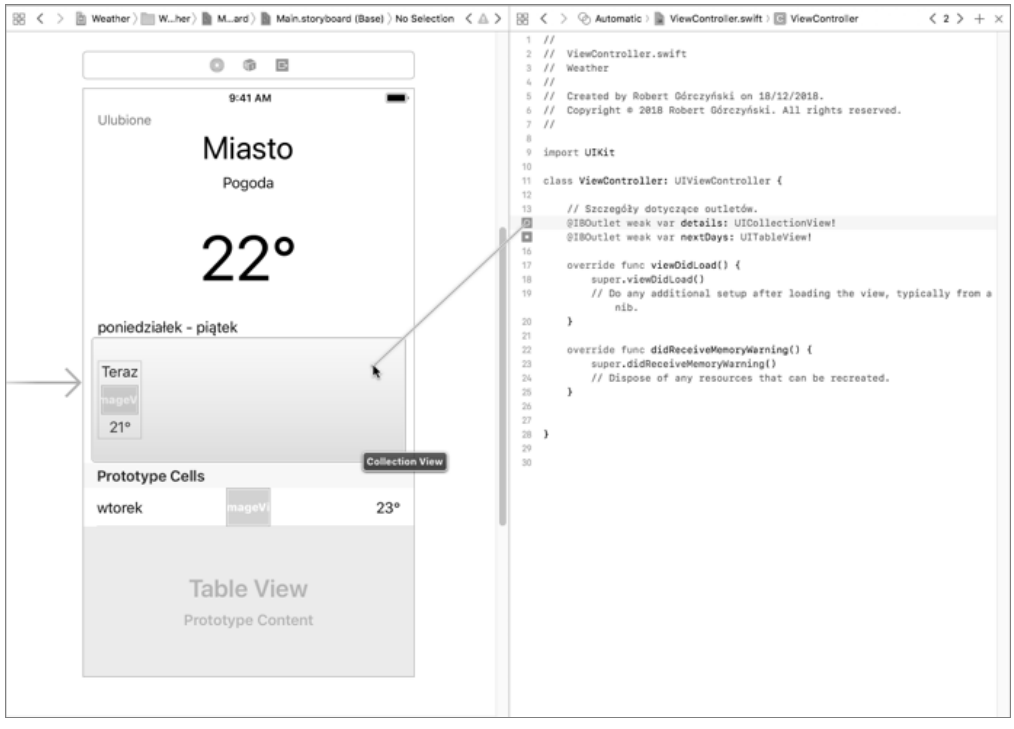
Dane modelu będą przechowywane w klasie kontrolera widoku, co oznacza, że powinna ona implementować interfejsy `UICollectionViewDataSource` i `UITableViewDataSource`. Trzeba zacząć od utworzenia w pliku *ViewController.swift* dwóch outletów pomagających w kontrolowaniu niezbędnych kolekcji.

```
// Szczegóły dotyczące outletów.
@IBOutlet weak var details: UICollectionView!
@IBOutlet weak var nextDays: UITableView!
```

Następnym krokiem jest połączenie elementów interfejsu użytkownika z nowymi outleltami. Na rysunku 7.20 pokazałem, jak to można zrobić (takie operacje już przeprowadzałeś we wcześniejszych rozdziałach).

Teraz trzeba utworzyć klasy komórek i połączyć wszystkie elementy komórek dla obu klas. Spójrz na kolejny fragment kodu.

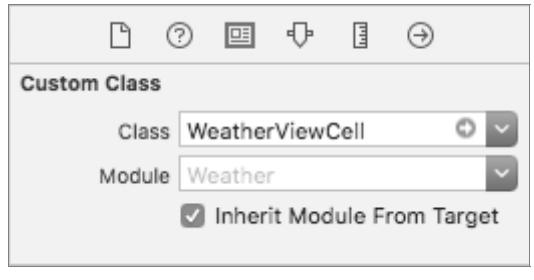
```
class DailyForecastViewCell: UITableViewCell {
    @IBOutlet weak var day: UILabel!
    @IBOutlet weak var icon: UIImageView!
    @IBOutlet weak var temperature: UILabel!
}
class WeatherViewCell: UICollectionViewCell {
    @IBOutlet weak var time: UILabel!
```



Rysunek 7.20. Połączenie outletu z elementem interfejsu użytkownika

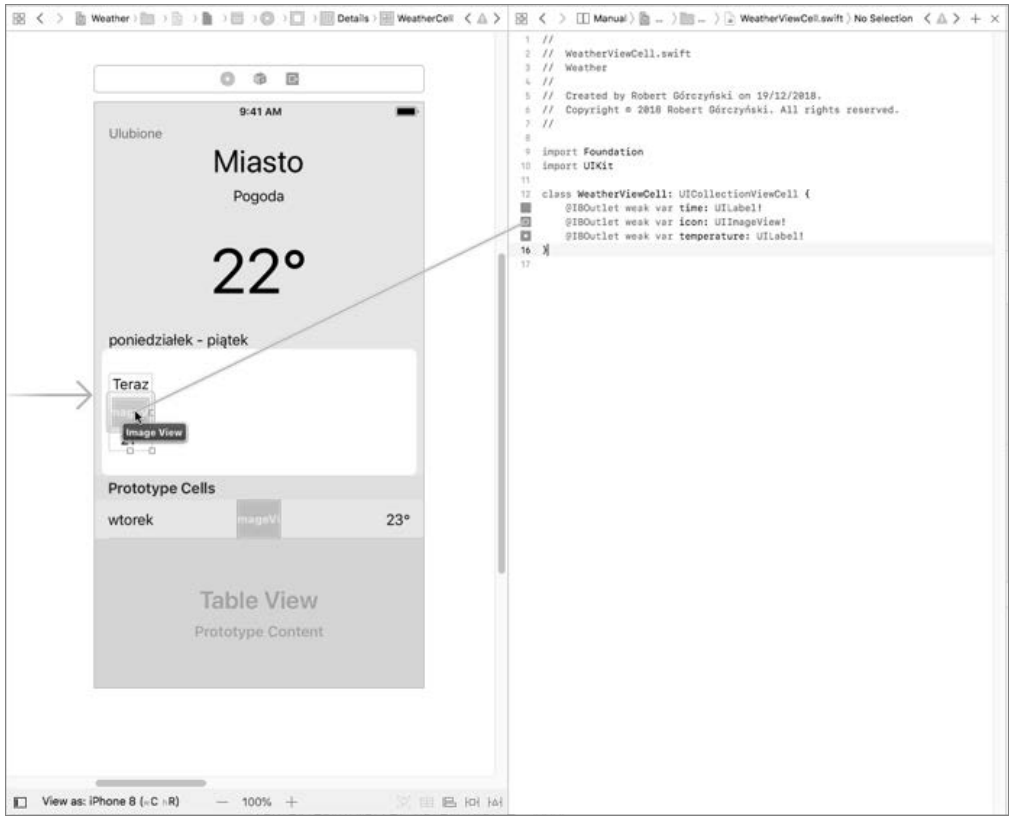
```
    @IBOutlet weak var icon: UIImageView!
    @IBOutlet weak var temperature: UILabel!
}
```

Każda komórka prototypu powinna być połączona z odpowiadającą jej klasą. Takie połączenie można zdefiniować w panelu inspektora Identity, tak jak pokazałem na rysunku 7.21.



Rysunek 7.21. Panel inspektora Identity

Teraz można już przystąpić do połączenia komórek prototypu z odpowiednimi outletami, co pokazałem na rysunku 7.22.



Rysunek 7.22. Łączenie komórek z outletami

Można już przystąpić do usprawnienia klasy kontrolera widoku odpowiedzialnej za obsługę ekranu głównego.

```
class ViewController: UIViewController {
    var model:LocationForecast?
    // Szczegóły dotyczące outletów.
    @IBOutlet weak var details: UICollectionView!
    @IBOutlet weak var nextDays: UITableView!
    var forecast:[Forecast] = []
    var degreeSymbol = "°"

    let collectionViewFormatter = DateFormatter()
    let tableViewFormatter = DateFormatter()

    override func viewDidLoad() {
        super.viewDidLoad()
        // Wypełnienie modelu przykładowymi danymi.
        model = LocationForecast.getTestData()
        collectionViewFormatter.dateFormat = "H:mm"
        tableViewFormatter.dateFormat = "EEEE"
        // Klasa implementuje odpowiednie protokoły w rozszerzeniach.
```

```
        details.dataSource = self
        nextDays.dataSource = self
    }

    // MARK: metoda prywatna.
    fileprivate func getIcon(weather:String) -> UIImage? {
        return nil
    }
}
```

Przedstawiony fragment kodu definiuje ogólny sposób działania kontrolera widoku. Konieczne jest jeszcze dodanie rozszerzeń klasy odpowiedzialnych za wczytywanie danych do egzemplarzy `UICollectionView` i `UITableView`. Oto rozszerzenie dla pierwszego z wymienionych egzemplarzy.

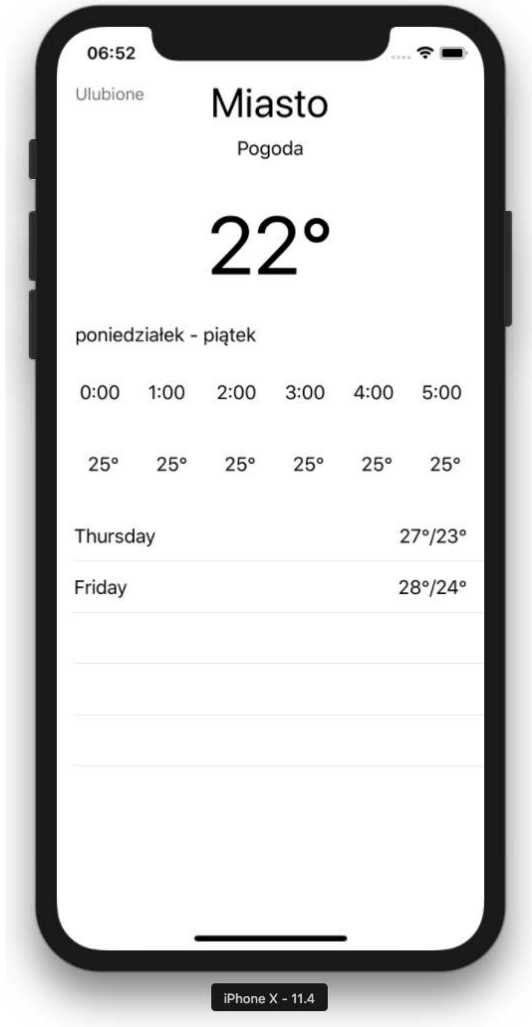
```
extension ViewController: UICollectionViewDataSource {
    public func collectionView(_ collectionView: UICollectionView,
      numberOfItemsInSection section: Int) -> Int {
        return model?.forecastForToday?.count ?? 0
    }
    public func collectionView(_ collectionView: UICollectionView,
      cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell:WeatherViewCell = collectionView
          .dequeueReusableCell(withReuseIdentifier:
          "WeatherCell", for: indexPath) as! WeatherViewCell
        let forecast:Forecast = (model?.forecastForToday?[indexPath.row])!
        cell.time.text = collectionViewFormatter.string(from: forecast.date)
        cell.icon.image = getIcon(weather: forecast.weather)
        cell.temperature.text = "\(forecast.temperature)\(self.degreeSymbol)"
        return cell
    }
}
```

To rozszerzenie zawiera implementację protokołu `UICollectionViewDataSource`. Z kolei w następnym fragmencie kodu przedstawiłem rozszerzenie zapewniające kontrolerowi widoku zgodność z protokołem `UITableViewDataSource`.

```
extension ViewController: UITableViewDataSource {
    public func tableView(_ tableView: UITableView, numberOfRowsInSection
      section: Int) -> Int {
        return model?.forecastForNextDays?.count ?? 0
    }
    public func tableView(_ tableView: UITableView, cellForRowAt
      indexPath: IndexPath) -> UITableViewCell {
        let cell:DailyForecastViewCell = tableView
          .dequeueReusableCell(withIdentifier: "FullDayWeatherCell",
          for: indexPath) as! DailyForecastViewCell
        let forecast:DailyForecast = (model?.forecastForNextDays? [indexPath.row])!
        cell.day.text = tableViewFormatter.string(from: forecast.date)
        cell.icon.image =  getIcon(weather: forecast.weather)
        cell.temperature.text = "\(forecast.maxTemp)
          \(self.degreeSymbol)/\(forecast.minTemp)\(self.degreeSymbol)"
        return cell
    }
}
```

Dzięki temu rozszerzeniu egzemplarza kontrolera widoku można użyć do dostarczenia danych dla elementów tabeli. W ten sposób otrzymamy prognozę pogody dla kilku kolejnych dni.

Jeżeli wszystko zostało przygotowane prawidłowo, po uruchomieniu aplikacji w symulatorze (tutaj to iPhone X) powinieneś otrzymać wynik pokazany na rysunku 7.23.



Rysunek 7.23. Omawiana aplikacja uruchomiona w symulatorze iPhone X

Nie wszystkie pola zostały połączone. Aby mieć możliwość podania nazwy miasta i innych szczegółów, konieczne jest zdefiniowanie kolejnych outletów. Możesz je utworzyć w kodzie, a następnie połączyć z odpowiednimi elementami interfejsu użytkownika.

Do projektu można dodać obrazy warunków atmosferycznych (sprawdź plik *Assets.xcassets* w materiałach przygotowanych dla książki), a następnie uaktualnić kod źródłowy, aby aplikacja zaczęła wyświetlać te ikony.

Potrzebna będzie metoda pomocnicza mapująca tekst na egzemplarze `UIImage`. Przedstawioną tutaj metodę należy dodać do klasy `LocationForecast`.

```
static func getImageFor(weather:String) -> UIImage {
    switch weather.lowercased() {
         case "słonecznie":
             return #imageLiteral(resourceName: "sunny")
         case "deszcz":
             fallthrough
         case "lekki deszcz":
             return #imageLiteral(resourceName: "rain")
         case "śnieg":
             return #imageLiteral(resourceName: "snow")
         case "zachmurzenie":
             return #imageLiteral(resourceName: "cloudy")
         case "częściowe zachmurzenie":
             return #imageLiteral(resourceName: "partly_cloudy")
         default:
             return #imageLiteral(resourceName: "sunny")
    }
}
```

Trzeba również uaktualnić metodę `getIcon(weather:String)` w klasie `ViewController`. W kolejnym fragmencie kodu przedstawiłem jej zmodyfikowaną wersję.

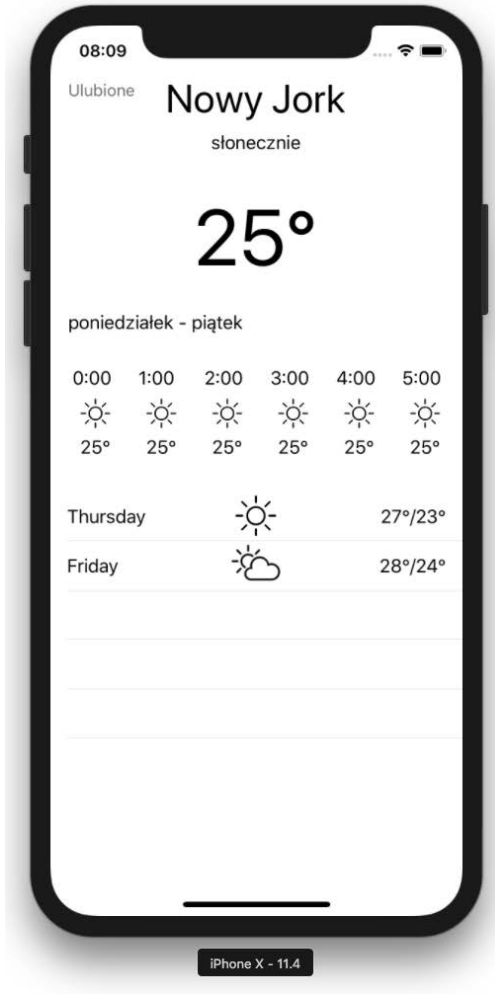
```
fileprivate func getIcon(weather:String) -> UIImage? {
    return LocationForecast.getImageFor(weather:weather)
}
```

Po wprowadzeniu tych drobnych zmian ekran główny aplikacji wygląda znacznie lepiej i dostarcza użytkownikowi informacje w znacznie prostszej postaci, co pokazałem na rysunku 7.24.

Aby zakończyć pracę nad tym ekranem, naciśnij przycisk *Ulubione* widoczny w lewym górnym rogu, a to powinno spowodować wyświetlenie innego ekranu. Musisz poznać sposób na zdefiniowanie w procedurze obsługi przycisku przejścia do innego ekranu, tym zajmiesz się za chwilę. Wcześniej trzeba przygotować ekran ulubionych lokalizacji. Najpierw skoncentrujesz się na jego opracowaniu, a dopiero później połączysz oba ekrany.

Jeżeli strzałkę widoczną w obszarze roboczym Storyboard przesuniesz w taki sposób, że wskaże inny kontroler widoku, stanie się on głównym kontrolerem widoku wyświetlanym po uruchomieniu aplikacji. Trzeba więc wskazać drugi kontroler widoku aplikacji (zobacz rysunek 7.25) jako ten, który będzie wyświetlany po jej uruchomieniu.

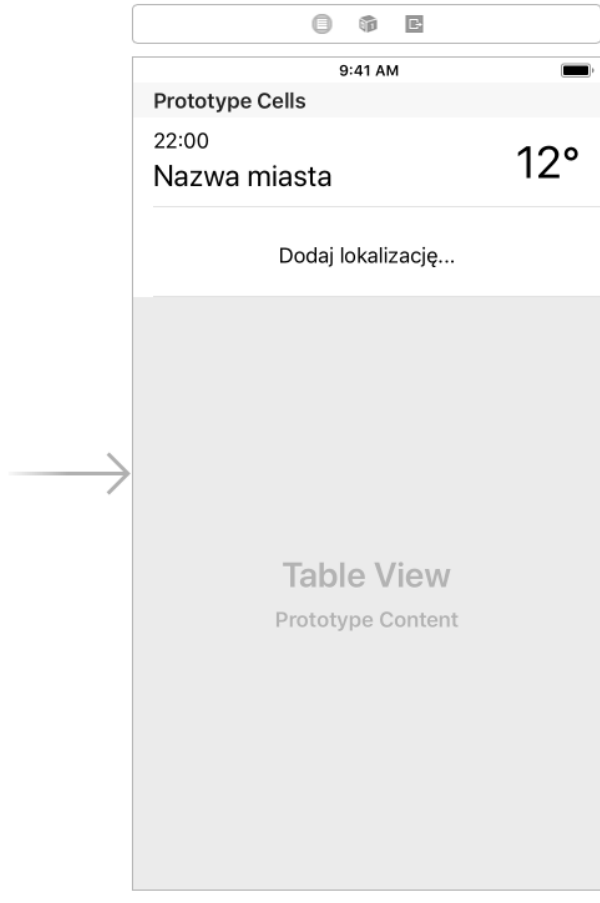
Tworzona w rozdziale aplikacja musi obsługiwać lokalizacje, ponieważ są one ważnym elementem jej modelu.



Rysunek 7.24. Ikony wyświetlane na ekranie głównym aplikacji

Lokalizacje

Aby zachować przejrzystość implementacji, poszczególne lokalizacje powinny być powiązane z miastem. Warto mieć nazwę miasta i kraj, w którym się ono znajduje, co pozwala na łatwą identyfikację lokalizacji. Ponieważ używane dane są początkowo filtrowane, można przyjąć założenie o braku jakichkolwiek powielonych danych. Dlatego też każde miasto będzie unikatowe w grupie danego kraju. Takie założenia są wystarczające dla obecnej wersji aplikacji *Weather*.



Rysunek 7.25. Drugi kontroler widoku jest teraz wyświetlany po uruchomieniu aplikacji

Model podobny do wymienionego opracowałeś już w poprzednim rozdziale — pozwalał na wyświetlanie krajów i leżących w nich miast. W budowanej tutaj aplikacji wykorzystasz ten model, choć nieco rozbudowany do potrzeb projektu.

Struktura lokalizacji powinna być związana z konkretnym miastem, a dla każdego miasta musi być zdefiniowana strefa czasowa (czasy zimowy lub letni nie mają tutaj znaczenia).

```
// Model kraju i miasta.
class Country {
    var name = "brak"
    var cities:[City] = []
    init(name:String) {
        self.name = name
    }
    init(name:String, cities:[City]) {
        self.name = name
        self.cities = cities
    }
```

```
}
public class City {
    var name: String
    init(name:String) {
        self.name = name
    }
    static var NewYork: City = {
        return City(name: "Nowy Jork")
    } ()
}

public struct Location {
    var city:City
    init(city: City) {
        self.city = city
    }
    var name: String {
        get {
                return self.city.name
        }
    }
}
```

Konieczne jest uaktualnienie ekranu głównego aplikacji, ponieważ używa on poprzedniej wersji struktury `Location`.

W pliku *LocationForecast.swift* znajduje się następujący wiersz kodu:

```
let location = Location(name: "Nowy Jork")
```

Należy go zastąpić nowym:

```
let location = Location(city: City.NewYork)
```

Mamy elegancki model pozwalający na przygotowanie listy wszystkich ulubionych lokalizacji. Teraz należy go połączyć z elementami interfejsu użytkownika. Takie zadanie wykonywałeś już wielokrotnie.

Trzeba zacząć od utworzenia nowego kontrolera widoku (plik *FavoritesViewController.swift*) i zdefiniowania zmiennej przechowującej odwołanie do widoku tabeli. Ta zmienna musi zostać połączona z rzeczywistym elementem interfejsu użytkownika w pliku Storyboard.

```
@IBOutlet weak var favoritesTableView: UITableView!
```

Potrzebna jest również kolekcja przechowująca wszystkie lokalizacje wyświetlone na ekranie.

```
var favorites:[Location] = []
```

Przechodzimy teraz do implementacji minimalnej liczby wymaganych metod, aby cokolwiek mogło zostać wyświetlone przez egzemplarz `UITableView` na ekranie.

```
public override func viewDidLoad() {
    super.viewDidLoad();
    formatter.dateFormat = "H:mm"
    loadFavorites()
    if favorites.count == 0 {
        // Domyślną lokalizacją jest Nowy Jork.
        var loc = Location.init(city: City.NewYork)
        // W przypadku strefy czasowej DST wartością loc.timeZone powinno być -4 * 3600.
        loc.timeZone = -5 * 3600
        favorites.append(loc)
    }
    favoritesTableView.dataSource = self
    favoritesTableView.delegate = self
}
```

Następnym krokiem jest implementacja przedstawionych tutaj interfejsów, czyli `UITable`
`↪ViewDataSource` i `UITableViewDelegate`. Należy je zaimplementować w tym samym pliku, ale w oddzielnych rozszerzeniach.

```
extension FavoritesViewController: UITableViewDataSource {
    public func tableView(_ tableView: UITableView,
      numberOfRowsInSection section: Int) -> Int {
        // Wartość 1 dla ostatniej komórki specjalnej.
        return favorites.count + 1
    }
    public func tableView(_ tableView: UITableView,
      cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let index = indexPath.row
        if index < favorites.count {
            let location = favorites[index]
            let cell:FavoriteViewCell = tableView
              .dequeueReusableCell(withIdentifier: "LocationCell",
              for: indexPath) as! FavoriteViewCell
            cell.city.text = location.name
            cell.temperature.text =  location.temperature +
            ↪LocationForecast.degreeSymbol
            let date = Date()
            formatter.timeZone = TimeZone(secondsFromGMT: location.timeZone)
            cell.time.text = formatter.string(from: date)
            return cell
        }
        // Ostatnia komórka jest statyczna.
        let cell:StaticViewCell = tableView
          .dequeueReusableCell(withIdentifier:
          "AddLocationCell", for: indexPath) as! StaticViewCell
        return cell
    }
}
```

Trzeba dostarczyć również implementację delegata, która zostanie usprawniona w przyszłości.

```
extension FavoritesViewController: UITableViewDelegate {
    public func tableView(_ tableView: UITableView,
      didSelectRowAt indexPath: IndexPath) {
```

```
            if indexPath.row == favorites.count {
                // TODO: otworzenie nowego kontrolera.
            } else {
                selectedItem = favorites[indexPath.row]
                // TODO: wybór lokalizacji i zapisanie wszystkich.
                saveFavorites(favorites: favorites)
            }
        }
    }
```

Operacje zapisu i wczytywania kolekcji ulubionych lokalizacji są obsługiwane przez dwie metody. Gdy kontroler widoku wyświetla dane, musi wczytać listę ulubionych lokalizacji. Jeżeli kolekcja jest pusta, będzie do niej dodany jeden element przedstawiający Nowy Jork. Zapis danych odbywa się po wybraniu elementu, ponieważ to spowoduje zamknięcie bieżącego widoku i uaktualnienie ekranu głównego aplikacji. Po zmianie sposobu działania aplikacji metodę zapisu można zmodyfikować. Oto kod metody odpowiedzialnej za zapisanie ulubionych lokalizacji.

```
// MARK: zapisywanie ulubionych lokalizacji.
func saveFavorites(favorites:[Location]) {
    let encoded = try? JSONEncoder().encode(favorites)
    let documentsDirectoryPathString = NSSearchPathForDirectoriesInDomains(
      .documentDirectory, .userDomainMask, true).first!
    let filePath = documentsDirectoryPathString + "/favorites.json"
    if !FileManager.default.fileExists(atPath: filePath) {
        FileManager.default.createFile(atPath: filePath, contents: encoded,
        ↪attributes: nil)
    } else {
        if let file = FileHandle(forWritingAtPath:filePath) {
            file.write(encoded!)
        }
    }
}
```

Metoda `saveFavorites()` konwertuje model na dane w formacie JSON (ang. *javascript object notation*), a następnie tworzy plik, w którym są one zapisywane.

> Podczas pracy z plikami w urządzeniu iOS trzeba używać klasy `FileManager` i jej egzemplarza domyślnego `FileManager.default`. Jeżeli masz plik do odczytania, zapisania lub uaktualnienia, skorzystaj z egzemplarza klasy `FileHandler`.

Spójrz teraz na kod metody `loadFavorites()`.

```
func loadFavorites() {
    let documentsDirectoryPathString = NSSearchPathForDirectoriesInDomains(
      .documentDirectory, .userDomainMask, true).first!
    let filePath = documentsDirectoryPathString + "/favorites.json"
    if FileManager.default.fileExists(atPath: filePath) {
        if let file = FileHandle(forReadingAtPath:filePath) {
            let data = file.readDataToEndOfFile()
            let favs = try? JSONDecoder().decode([Location].self, from: data)
```

```
                favorites = favs!
            }
        }
    }
```

Metoda `loadFavorites()` próbuje wczytać zawartość pliku modelu. Jeżeli wskazany plik istnieje, zostanie otworzony. Egzemplarz `JSONDecored` przeprowadza konwersję ciągu tekstowego na postać tablicy przechowującej lokalizacje. Naciśnięcie elementu *Dodaj lokalizację…* powinno spowodować wyświetlenie następnego ekranu pozwalającego użytkownikowi na wybranie nowej lokalizacji, która zostanie dodana do listy ulubionych. Podobne rozwiązanie zaimplementowałeś w poprzednim rozdziale podczas pracy nad funkcjonalnością wyszukiwania danych na liście. Implementację tej funkcji w tym projekcie pozostawiam Ci jako ćwiczenie. Skopiuj kod z poprzedniego rozdziału i dostosuj go do potrzeb tego projektu.

> Pełne działające rozwiązanie znajdziesz w materiałach przygotowanych dla książki, które możesz pobrać ze strony *ftp://ftp.helion.pl/przyklady/poswif.zip*.

W ten sposób zdefiniowałeś różne fragmenty aplikacji. Teraz dowiesz się, jak można je połączyć, aby powstała w pełni działająca aplikacja.

Kontrolery i przejścia

Każdy kontroler odpowiada za komponent graficzny wyświetlany na ekranie. Niektóre kontrolery odpowiadają za cały ekran, podczas gdy inne tylko za jego część. Tutaj skoncentrujesz się na tych pierwszych, ponieważ pracujesz nad aplikacją przeznaczoną dla urządzenia o małym ekranie. Jednak przedstawione tutaj rozwiązania można łatwo zastosować w bardziej skomplikowanych hierarchiach kontrolerów.

Koncepcją o znaczeniu kluczowym podczas przedstawiania nowych ekranów jest **przejście** określane w Xcode mianem **segue**. Przejście odbywa się bez żadnych zakłóceń od jednego kontrolera widoku do drugiego. Wykorzystując przejścia, można połączyć różne sceny w aplikacji, a nawet przekazywać informacje między kontrolerami widoku. Każde przejście może mieć zdefiniowane animacje odtwarzane w trakcie przechodzenia między scenami.

> Przejścia są ściśle związane z plikiem Storyboard.

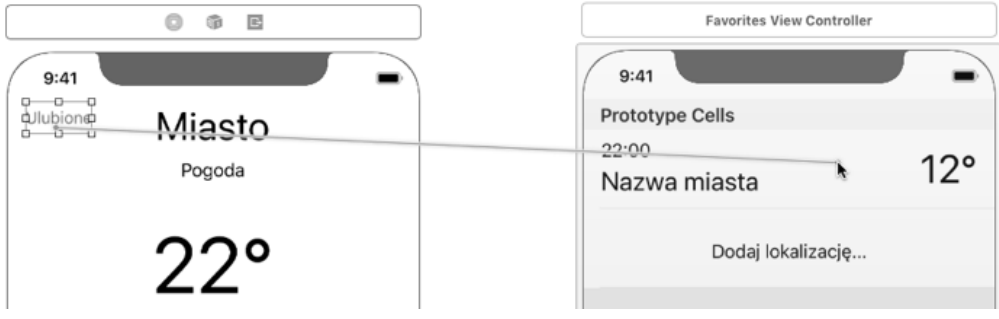
Przystąpisz teraz do utworzenia pierwszego przejścia między ekranem głównym aplikacji i widokiem zawierającym ulubione lokalizacje.

Pierwsze przejście

Dowiesz się teraz, jak tworzyć przejścia w pliku Storyboard i wywoływać je za pomocą kodu. Każde przejście to rodzaj związku między ekranami aplikacji i może być wywoływane na

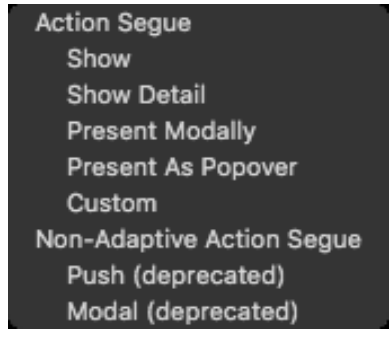
skutek działania podjętego przez użytkownika lub też w innych okolicznościach, np. o określonej godzinie lub w przypadku konkretnej akcji serwera.

1. Przejście najłatwiej można utworzyć przez przytrzymanie klawisza *Ctrl* podczas przeciągania myszą od jednego kontrolera widoku lub przycisku do innego kontrolera widoku, który powinien zostać wyświetlony na ekranie. W omawianym przypadku trzeba zacząć od ekranu głównego aplikacji. Naciśnij i przytrzymaj klawisz *Ctrl*, a następnie przeciągnij myszą od przycisku *Ulubione* do następnego kontrolera widoku (ulubionych lokalizacji). Operację pokazałem na rysunku 7.26.



Rysunek 7.26. Definiowanie w pliku Storyboard przejścia między kontrolerami widoku

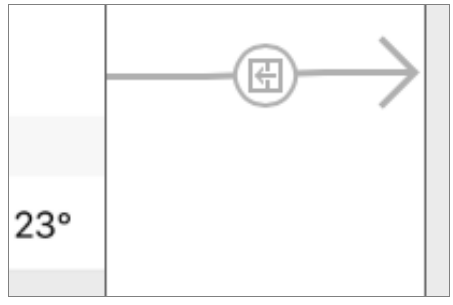
2. Po zwolnieniu przycisku myszy wyświetli się małe okno wraz z akcjami do wyboru (zobacz rysunek 7.27). Wybrana akcja definiuje sposób pojawienia się nowego kontrolera widoku na ekranie. W omawianym przypadku wybierz *Push (depreciated)*, w innych przypadkach masz do dyspozycji znacznie lepsze akcje.



Rysunek 7.27. Akcje dostępne podczas definiowania przejścia między kontrolerami widoku

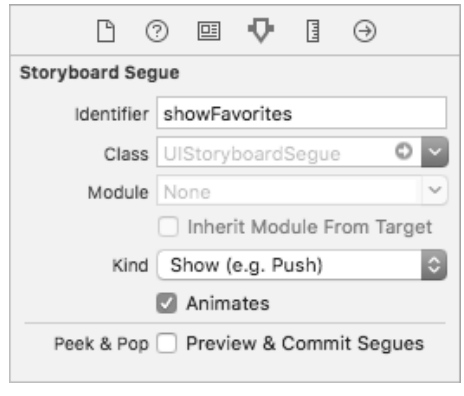
3. Po wybraniu akcji w pliku Storyboard zostanie zdefiniowane połączenie między dwoma kontrolerami widoku, co pokazałem na rysunku 7.28.

   To połączenie symbolizuje związek między dwoma ekranami. Po dodaniu kolejnego przejścia do innego kontrolera widoku otrzymasz następne połączenie między nimi.



Rysunek 7.28. Zdefiniowane w pliku Storyboard przejście między kontrolerami widoku

W panelu narzędziowym po prawej stronie okna Xcode można zaznaczyć połączenie i dodać dla niego identyfikator, który następnie będzie używany w kodzie do zainicjalizowania przejścia między kontrolerami widoku. W omawianym projekcie utworzonemu przejściu nadaj identyfikator `showFavorites`, tak jak pokazałem na rysunku 7.29.



Rysunek 7.29. Właściwości przejścia wyświetlone w panelu narzędziowym

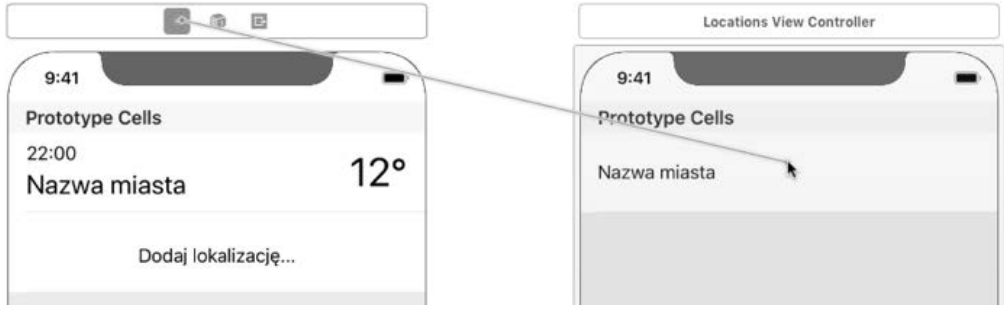
Utworzone przejście jest ściśle związane z przyciskiem i zostanie aktywowane po naciśnięciu danego przycisku (zobacz rysunek 7.30).

Rysunek 7.30. Przejście wymienione w ustawieniach przycisku

Innym sposobem na aktywowanie przejścia jest użycie kodu i dodanie akcji wywoływanej po wystąpieniu zdarzenia `Touch Up Inside`. W takim przypadku przejście można aktywować za pomocą następującego kodu:

```
    @IBAction func onFavoritesClicked(_ sender: Any) {
    performSegue(withIdentifier: "showFavoritesAlternative", sender: sender)
}
```

W pliku Storyboard można zdefiniować przejście o identyfikatorze `showFavoritesAlternative`. To ogólne przejście aktywowane z poziomu kodu można zdefiniować przez przytrzymanie klawisza *Ctrl* i przeciągnięcie myszą z jednego kontrolera widoku do innego, tak jak pokazałem na rysunku 7.31.



Rysunek 7.31. Definiowanie ogólnego przejścia w pliku Storyboard

Jest to przykład pokazujący połączenie dwóch kolejnych kontrolerów widoku. Istnieje możliwość zdefiniowania dowolnej liczby przejść. Będą one wyświetlane w pliku Storyboard i pomagają w wizualnym przedstawieniu skomplikowanych związków zachodzących między komponentami aplikacji.

Skoro poznałeś sposoby przejścia z jednej sceny do drugiej, teraz dowiesz się, jak przekazywać dane między kontrolerami widoku, dzięki czemu użytkownik uwierzy, że te sceny są powiązane. Każdy ekran zdefiniujesz jako szablon przedstawiający konkretne informacje. Te dane muszą być przekazywane, a kontroler widoku ma je obsłużyć.

Przekazywanie danych między scenami

Poszczególne ekrany podczas projektowania są definiowane jako zależne od pewnego modelu (danych). Ten model powinien być przekazywany podczas przejścia, a jego dane pokazane po wyświetleniu nowego kontrolera widoku. Dzięki temu użytkownik jest przekonany o połączeniu obu ekranów. Przykładem może być tutaj widok kolekcji i widok szczegółowy wyświetlający informacje dodatkowe. Wprawdzie te dwa ekrany mogą działać niezależnie, ale podczas przekazywania danych między nimi będą postrzegane jako jedna całość.

> Doskonale opracowane sceny (kontrolery widoku) mogą pojawiać się w wielu miejscach aplikacji.

W budowanej tutaj prostej aplikacji każdy ekran pełni określoną rolę i nie ma możliwości ich wielokrotnego używania w różnych miejscach. Jednak w doskonale zaprojektowanych aplikacjach należy korzystać z takiej możliwości.

Przejdź teraz do kontrolera widoku, który ma przekazać dane po rozpoczęciu przejścia. W tym kontrolerze należy nadpisać metodę `prepare()`, tak jak pokazałem w kolejnym fragmencie kodu.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
     if let id = segue.identifier {
        switch id {
            case "showFavorites":
                print("Przekazywanie danych.");
            default:
                break;
        }
    }
}
```

Ten kod ustala wykonywane przejście. Doskonale wiadomo, który ekran zostanie wyświetlony jako następny. Nowy kontroler widoku powinien zaakceptować otrzymane dane. Kontroler widoku zwykle ma właściwość (lub właściwości) publiczną, która powinna być zdefiniowana. Przedstawiona tutaj metoda ma dwa argumenty. Pierwszy to obiekt przejścia zawierający oba kontrolery widoku. Natomiast drugi to obiekt `sender` określający element, który zainicjalizował przejście. Oba wymienione argumenty są niezbędne do rozróżniania między różnymi logicznymi scenariuszami i pozwalają na opracowanie aplikacji działającej odmiennie w różnych sytuacjach. Spójrz na przykład kodu przekazującego pewne dane do kontrolera widoku ekranu ulubionych lokalizacji.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let id = segue.identifier {
        switch id {
            case "showFavorites":
                guard let favVC: FavoritesViewController =
                  segue.destination as? FavoritesViewController else {
                    return
                }
                favVC.receivedData = 42
            default:
                break;
        }
    }
}
```

Właściwość `receivedData` może być bez żadnych problemów użyta w metodzie `viewDidLoad()`. Takie rozwiązanie pozwala na przekazanie wielu informacji, które następnie zostaną użyte w innym kontrolerze widoku. Nie ma żadnego ograniczenia pod względem typu przekazywanych danych. Po aktywowaniu następnego kontrolera widoku będzie miał on dostęp do przekazanych mu danych.

Nieco innym problemem jest przekazanie danych w drugą stroną. Spróbuję to najpierw wyjaśnić. Gdy w potomnym kontrolerze widoku są przeprowadzane pewne akcje, warto mieć możliwość przekazania danych do nadrzędnego kontrolera widoku. Takie podejście pozwala na usprawnienie projektu modułu. Każdy kontroler widoku wykonuje swoje zadanie i przekazuje model po jego uaktualnieniu.

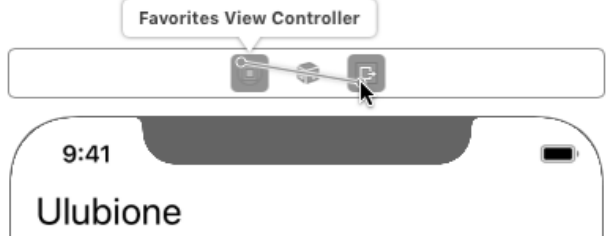
Przekazywanie danych w drugą stronę

Przekazywanie informacji do nadrzędnego kontrolera widoku jest nieco trudniejsze. Nie istnieje łatwy sposób na ustalenie, który kontroler widoku spowodował przejście do aktualnego kontrolera widoku. Dlatego też gdy chcesz zdefiniować przejście z powrotem do konkretnego kontrolera widoku, powinieneś utworzyć w nim metodę specjalną.

W omawianej aplikacji trzeba przekazać do ekranu głównego element wybrany na ekranie ulubionych lokalizacji. To wymaga dodania nowej metody obsługującej przejście w drugą stronę.

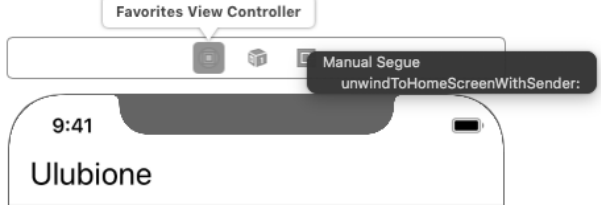
```
@IBAction func unwindToHomeScreen(sender: UIStoryboardSegue) {
    if let favoritesVC = sender.source as? FavoritesViewController {
        model = LocationForecast()
        model?.location = favoritesVC.selectedItem
    }
}
```

Ta metoda zostanie wywołana po wykryciu przejścia w drugą stronę. Konieczne jest przekazanie danych ze źródłowego kontrolera widoku do bieżącego. Aby utworzyć takie przejście, przytrzymaj klawisz *Ctrl* i przeciągnij myszą od kontrolera widoku do punktu wyjścia, tak jak pokazałem na rysunku 7.32.



Rysunek 7.32. Definiowanie przejścia w drugą stronę

Po zwolnieniu przycisku myszy wybierz tę metodę, która ma zostać wykonana (zobacz rysunek 7.33). Metody wymienione na liście należą do innych kontrolerów widoku.



Rysunek 7.33. Wybór metody do wykonania po zainicjalizowaniu przejścia w drugą stronę

W ten sposób dowiedziałeś się, jak zdefiniować przejście, które spowoduje zastąpienie wyświetlonego kontrolera widoku innym, i jak wrócić do konkretnego kontrolera widoku. Teraz przedstawię możliwości w zakresie tworzenia niestandardowych i animowanych przejść.

Definiowanie niestandardowego przejścia

Istnieje możliwość zdefiniowania niestandardowego przejścia, aby odtworzyć animację podczas aktywacji lub dezaktywacji. W tym podpunkcie zobaczysz, jak można przygotować takie przejście. Rozpocznij od utworzenia nowej klasy rozszerzającej `UIStoryboardSegue`.

```
class ZoomInSegue : UIStoryboardSegue {
    override func perform() {
        zoomIn()
    }

    func zoomIn() {
        let superView = self.source.view.superview
        let center = self.source.view.center
        self.destination.view.transform =
            CGAffineTransform.init(scaleX: 0.05, y: 0.05).rotated(by: 90 * .pi / 180)
        self.destination.view.center = center
        superView?.addSubview(self.destination.view)
        UIView.animate(withDuration: 0.5, delay: 0, options: .curveEaseIn, animations: {
            self.destination.view.transform = CGAffineTransform.identity
        }, completion: { success in self.source.present(
            self.destination, animated: false, completion: nil)
        })
    }
}
```

W tej klasie trzeba nadpisać metodę `perform()` aktywowaną po rozpoczęciu przejścia. Wprawdzie w omawianym przykładzie zdecydowałem się na użycie metody pomocniczej, ale cały kod można również umieścić w metodzie `perform()`. Znaczenie kluczowe ma utworzenie animacji za pomocą wywołania `UIView.animate()` i wyświetlenie nowego kontrolera widoku bez animacji domyślnej (będzie użyta zdefiniowana tutaj w kodzie). Przygotowana animacja interpoluje stan początkowy (mały odwrócony widok) i końcowy (widok na środku ekranu wypełniający go w całości). Najlepszym sposobem na tworzenie przyjemnych dla oka animacji jest eksperymentowanie z różnymi wartościami i przekształceniami.

W następnym punkcie pokrótce przedstawię propozycje dalszego usprawnienia aplikacji.

Dalsze usprawnienia aplikacji

Czy istnieje łatwiejszy sposób na zmianę lokalizacji? Aktualnie aplikacja wyświetla tylko jedną ulubioną lokalizację. Byłoby dobrze, gdyby użytkownik miał możliwość łatwego przechodzenia między nimi i sprawdzania prognozy pogody. Przewijanie poziome między prognozami pogody dla różnych lokalizacji wydaje się być intuicyjnym rozwiązaniem. Dzięki temu za pomocą łatwego gestu machnięcia użytkownik mógłby sprawdzić prognozę pogody w następnej ulubionej lokalizacji. Zmiana kolejności lokalizacji na liście ulubionych powinna zostać odzwierciedlona również na ekranie głównym.

Aby wprowadzić te usprawnienia, należy sprawdzić, czy aktualny projekt aplikacji (kod i interfejs użytkownika) pozwala na ich wprowadzenie. Oto wymagania dla uaktualnionej aplikacji.

- Ekran główny powinien zostać rozbudowany i używać listy widoków, a wszelkie interakcje mają być przekazywane delegatowi. Zastosowany kontroler jest naprawdę abstrakcyjny, ale wykorzystuje pojedynczy widok. Konieczne jest utworzenie ekranu głównego pozwalającego na łatwe przejście między lokalizacjami.
- Potrzebny jest kontroler specjalny pozwalający na przewijanie poziome. System iOS dostarcza doskonałe rozwiązanie w tym zakresie — klasę `UIScrollView`.
- Aplikacja powinna używać jedynie modeli, aby mogła wygenerować treść po ich uaktualnieniu. Obecnie zastosowany model wymaga zmian, ale można je wprowadzić podczas pracy nad zmodyfikowaną wersją aplikacji.

Wymienione usprawnienia aplikacji odzwierciedlą jej aktualny model i strukturę, a ponadto mogą być wprowadzone przez osobę, która opanowała dotychczasową wiedzę o frameworku iOS przedstawioną w tej książce. Prawdopodobnie używałeś systemu kontroli wersji Git podczas pracy nad aplikacją w rozdziale. Jeżeli nie, to jest doskonały moment na przekazanie kodu do repozytorium. Utwórz repozytorium lokalne i umieść w nim bieżącą wersję aplikacji. Następnie utwórz nową gałąź, np. o nazwie `side-scrolling`, w której będziesz mógł eksperymentować z kodem. W ten sposób możesz prowadzić eksperymenty bez obaw o działającą wersję aplikacji. Repozytorium Git zapewnia pewne bezpieczeństwo, ponieważ zawsze możesz powrócić do gałęzi `master`, utworzyć nową gałąź i rozpocząć eksperymenty od początku.

Obraz wart jest tysiąca słów. Aby upiększyć aplikację, można dodać do niej pewne zasoby graficzne, np. przedstawiające aktualne warunki atmosferyczne w wybranej lokalizacji. W ten sposób można znacznie łatwiej powiązać prognozę pogody z tym, co dzieje się na zewnątrz. Jeżeli chcesz zdobyć nowe umiejętności w zakresie programowania, spróbuj zaimplementować wybrane z wymienionych tutaj pomysłów.

Podsumowanie

Wiesz, jak utworzyć aplikację rozwiązującą rzeczywiste problemy. Rozpocząłeś od przygotowania projektu na papierze, a następnie przystąpiłeś do utworzenia interfejsu użytkownika w pliku Storyboard. Taka wersja aplikacji jest określana mianem **wczesnego prototypu**. Możesz eksperymentować z ekranami o różnej wielkości i zapewnić prawidłowe wyświetlanie na nich interfejsu użytkownika aplikacji. Poznałeś sposoby na dodanie interaktywności i połączenia ze sobą poszczególnych ekranów. Dokładnie omówione w rozdziale przejścia i przekazywanie danych między ekranami to techniki używane we wszystkich aplikacjach. Wykorzystałeś standardowe narzędzia oferowane przez iOS i przykładowe dane. Użyte dane symulują zachowanie aplikacji po otrzymaniu przez nią rzeczywistych danych.

W następnym rozdziale przedstawię temat nowoczesnego tworzenia oprogramowania na przykładzie projektów typu open source. Zobaczysz, jak łatwo można połączyć wiele projektów za pomocą CocoaPods, czyli jednego z najpopularniejszych menedżerów zależności dla języków Swift i Objective-C. Zobaczysz również, jak wygląda dołączanie innych frameworków do własnych projektów. Należy skończyć z wyważaniem otwartych drzwi — znacznie lepsze podejście polega na wyszukaniu projektów open source, które mogą sprawdzić się w budowanej aplikacji. Dzięki temu można zaoszczędzić wiele czasu podczas tworzenia i debugowania aplikacji.

Skorowidz

A

B

C

D

E

F

G

I

J

K

L

Ł

M

N

O

P

R

S

T

U

W

X

Z

Ż

Oto Swift: najlepszy język dla aplikacji na iOS!

Swift został zaprezentowany światu w 2014 roku — dziś jest uznanym standardem dla programistów platformy macOS i iOS. Równocześnie to jeden z najpopularniejszych języków programowania na świecie. Charakteryzuje się zwięzłą i przejrzystą składnią, jest łatwy do nauczenia się, wygodny i elastyczny. Programiści mawiają, że Swift podsuwa nowe sposoby rozwiązywania starych problemów. Jeśli chcesz pisać efektywne i bezpieczne, a przy tym eleganckie i przyjazne użytkownikom aplikacje dla maszyn z logo jabłuszka, po prostu musisz nauczyć się Swifta!

Jeśli jesteś początkującym programistą i postanowiłeś nauczyć się rzetelnego programowania aplikacji mobilnych dla iOS, to książka dla Ciebie. Znajdziesz w niej przystępne wprowadzenie do koncepcji programowania oraz podstawy języka Swift. Przećwiczysz tworzenie aplikacji mobilnych na platformie iOS. Dowiesz się, jak tworzyć interfejs użytkownika za pomocą plików typu storyboard w Xcode, a także jak pobierać i wyświetlać obrazy oraz zapisywać i wczytywać informacje w trakcie różnych sesji pracy z aplikacją. Nauczysz się korzystać z menedżera zależności CocoaPods i przekonasz się, jak bardzo jest użyteczny. Poznasz kilka przydatnych bibliotek open source do szybkiego tworzenia oprogramowania, dowiesz się też, jak opracowywać aplikacje pobierające informacje i zasoby z chmury.

Najważniejsze zagadnienia:

- składnia i elementy języka Swift oraz praca w środowisku Xcode
- struktura aplikacji mobilnej
- zastosowanie poszczególnych struktur danych w Swifcie
- tworzenie GUI i zapewnianie interaktywności aplikacji
- biblioteki dla open source Swifta

Emil Atanasov — od ponad dziesięciu lat programuje aplikacje dla urządzeń mobilnych, jest też doświadczonym konsultantem IT w tej dziedzinie. Obecnie prowadzi własną firmę, Appose Studio Inc., świadczącą usługi konsultingowe klientom z całego świata. Wcześniej pracował dla wielu amerykańskich i brytyjskich firm jako kierownik zespołu, menedżer projektu oraz programista aplikacji dla platform iOS i Android.

ISBN 978-83-283-5453-1
9 788328 354531
Cena: 59,00 zł

Packt