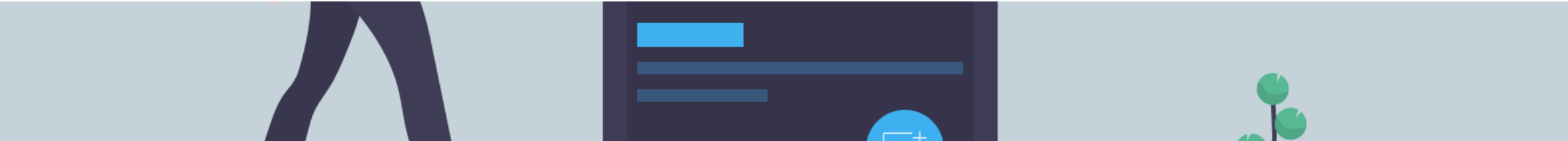




15 JUNE 2020 / [SWIFTUI](#)

Infinite List Scroll with SwiftUI and Combine





Infinite scrolling is a UX pattern which loads content continuously when users scroll down the screen. An example of an infinite list that you probably already know is Twitter or Instagram feed. In this article, let's implement an endless list of GitHub repositories using the SwiftUI and Combine frameworks, and the MVVM iOS app architecture pattern.

Firing Paginated HTTP Request

We begin by adding a network request that fetches repositories using [GitHub REST API](#):

In [Modern Networking in Swift](#) you can learn how to implement a networking layer from scratch.

```
enum GithubAPI {
    static let pageSize = 10

    static func searchRepos(query: String, page: Int) -> AnyPublisher<[Repository], Error> {
        let url = URL(string: "https://api.github.com/search/repositories?q=\(query)&sort=stars&per
        return URLSession.shared
            .dataTaskPublisher(for: url) // 1.
            .tryMap { try JSONDecoder().decode(GithubSearchResult<Repository>.self, from: $0.data) }
```

```

        .receive(on: DispatchQueue.main) // 3.
        .eraseToAnyPublisher()
    }
}

```

Here are the takeaways:

1. Create a publisher that wraps a URL session data task.
2. Decode the response as `GithubSearchResult`. This is an intermediate type created for the purpose of parsing JSON.
3. Receive response on the main thread.

The models are implemented as follows:

```

struct GithubSearchResult<T: Codable>: Codable {
    let items: [T]
}

struct Repository: Codable, Identifiable, Equatable {
    let id: Int
    let name: String
    let description: String?
    let stargazers_count: Int
}

```

Implementing Static List

After fetching GitHub repositories from the network, we display them in a static list:

```

struct RepositoriesList: View {
    // 1.
    let repos: [Repository]
    let isLoading: Bool
    let onScrolledAtBottom: () -> Void

    // 2.
    var body: some View {
        List {
            reposList
            if isLoading {
                loadingIndicator
            }
        }
    }

    private var reposList: some View { ... }

    private var loadingIndicator: some View { ... }
}

```

1. The list accepts an array of repositories to show, a callback that notifies when the list is scrolled to the bottom, and an `isLoading` flag, that indicates whether a loading animation needs to be shown.
2. The body contains a list and a loading indicator below it.

Let's take a closer look at `reposList`:

```

struct RepositoriesList: View {
    ...

```

```

private var reposList: some View {
    ForEach(repos) { repo in

        // 1.
        RepositoryRow(repo: repo).onAppear {
            // 2.
            if self.repos.last == repo {
                self.onScrolledAtBottom()
            }
        }
    }
}
...
}

```

1. `RepositoryRow` represents a list entry.
2. We call `onScrolledAtBottom()` when the last repository appears on the screen.

Here is how `RepositoryRow` is implemented:

```

struct RepositoryRow: View {
    let repo: Repository

    var body: some View {
        VStack {
            Text(repo.name).font(.title)
            Text("★ \"(repo.stargazers_count)\"")
            repo.description.map(Text.init)?.font(.body)
        }
        .frame(idealWidth: .infinity, maxWidth: .infinity, alignment: .center)
    }
}

```

Implementing the View Model

We'll use the MVVM pattern to organize the components of our infinite list. The view model will load data from the network, and compose a `state` object. The view, in, its turn, will bind to the state updates.

Here is an in-depth overview of the [modern state of the MVVM pattern](#).

```
// 1.
class RepositoriesViewModel: ObservableObject {
    @Published private(set) var state = State()
    private var subscriptions = Set<AnyCancellable>()

    // 2.
    func fetchNextPageIfPossible() {
        guard state.canLoadNextPage else { return }

        GithubAPI.searchRepos(query: "swift", page: state.page)
            .sink(receiveCompletion: onReceive,
                  receiveValue: onReceive)
            .store(in: &subscriptions)
    }

    ...

    // 3.
    struct State {
        var repos: [Repository] = []
        var page: Int = 1
        var canLoadNextPage = true
    }
}
```

1. To support data binding, the view model must conform to the `ObservableObject` protocol, and provide at least one `@Published` property. Whenever such a variable is updated, SwiftUI re-renders the bound view automatically.
2. The `fetchNextPageIfPossible()` method searches GitHub repositories using the 'swift' query. It checks that the next page is available before requesting it.
3. The state contains all the information to render a view.

The two missing pieces are the overloaded `onReceive()` methods which handle the API response:

```
class RepositoriesViewModel: ObservableObject {
    ...
    private func onReceive(_ completion: Subscribers.Completion<Error>) {
        switch completion {
        case .finished:
            break
        case .failure:
            state.canLoadNextPage = false
        }
    }

    private func onReceive(_ batch: [Repository]) {
        state.repos += batch
        state.page += 1
        state.canLoadNextPage = batch.count == GithubAPI.pageSize
    }
    ...
}
```

```
}
```

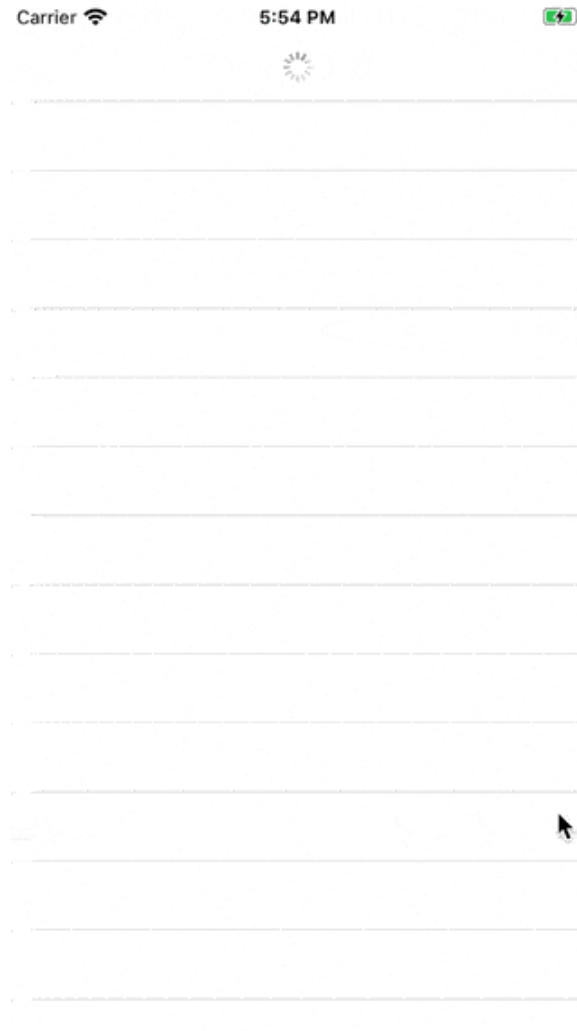
Implementing Infinite Scroll

Finally, let's connect the view model to the static list. We'll use the [container view pattern](#), and this is where the pagination logic will sit. The purpose of the container view is to provide the data and pagination behavior to `RepositoriesList`:

```
struct RepositoriesListContainer: View {
  @ObservedObject var viewModel: RepositoriesViewModel

  var body: some View {
    RepositoriesList(
      repos: viewModel.state.repos,
      isLoading: viewModel.state.canLoadNextPage,
      onScrolledAtBottom: viewModel.fetchNextPageIfPossible
    )
    .onAppear(perform: viewModel.fetchNextPageIfPossible)
  }
}
```

The final result looks next:



Source Code

You can find [the final project here](#). It is published under the "Unlicense", which allows you to do whatever you want with it.

Further Reading

- [Presentational and Container Components](#)
 - [Modern MVVM iOS App Architecture with Combine and SwiftUI](#)
 - [View Communication Patterns in SwiftUI](#)
 - [Asynchronous Programming with Futures and Promises](#)
-

Thanks for reading!

If you enjoyed this post, be sure to [follow me on Twitter](#) to not miss any new content.



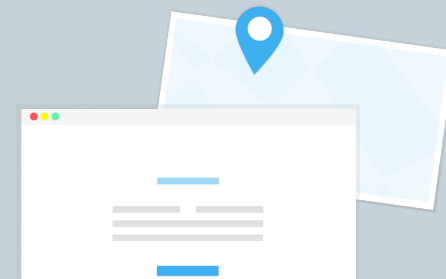
Vadim Bulavin

Creator of Yet Another Swift Blog. Lead iOS Engineer at [EPAM](#). Coding for fun since 2008, for food since 2012.

Follow

— Yet Another Swift Blog —

SwiftUI



SwiftUI Previews at Scale

Testing SwiftUI Views

Function Builders in Swift and SwiftUI

See all 12 posts →

SWIFT

Swift Pointers Overview: Unsafe, Buffer, Raw and Managed Pointers

Learn Swift pointers: what they are, when to use, and what you can get from them.



VADIM BULAVIN

6 MIN READ

Yet Another Swift Blog by Vadim Bulavin © 2020. Some images are copyright Icons8 LLC © 2018.

Latest publications · Twitter · vadybulavin@gmail.com