RepliFlix - My Replica of Netflix iOS App using Swift 5, UIKit, and Xcode.

Functionalities/Demo:
Having navigation bar scrolls with general table views
Generating lists of trending and upcoming movies and tv shows
Enabling to either play or download when the user clicks on certain movies or shows
Navigating through different tabs
Searching certain movies or shows when the user inputs certain keywords
Redirecting the user directly to the official trailers by clicking on the animated titles

Developing Techniques and Skills Needed:
Xcode 13.4.1 iOS 15.5+
SF Symbols Explorer
Swift 5,
MVVM(Model-View-Viewmodel)

Prerequisites Setups:
Set Xcode simulator to Dark Theme by Developer-> Dark Appearance.
Delete Main and Storyboard Key in Info Configuration
Search `main` in Find > Text > Containing:
Delete **`UIKit Main Storyboard File Base Name`** to get rid of Storyboard

Developing Notes:
In `SceneDelegate`:
We are going to be assigning the view controller to be the root view controller of our application.
Change `ViewController` into `MainTabBarViewController`
Use `view.backgroundColor = .systemYellow` to test if everything works okay so far.

4 main navigation bars in UI in Cocoa Touch Class Creation:
Create a new group for all controllers, name the folder of new group as `Controllers`
In `Controllers`, create a new file in Cocoa Touch Class, and name it `HomeViewController` as Home navigation bar, set the background color into red.
Create another file in Cocoa Touch Class, name it `UpcomingViewController`, as Upcoming navigation bar, set the background color into green.
Create another file in Cocoa Touch Class, name it `SearchViewController`, as Search navigation bar, set the background color into systemPink.
Create another file in Cocoa Touch Class, name it `DownloadsViewController`, as Downloads of movies and TV shows, set the background color into blue.

4 main navigation bars initialization:
In `MainTabBarViewController`:
Do `let vc1/2/3/4 = UINavigationController(rootViewController: HomeViewController/UpcomingController/SearchController/DownloadsViewController())` using vectors to initialize the 4 navigation bars.

Animation creation:
In `MainTabBarViewController:`
Do `setViewControllers([vc1,vc2,vc3,vc4],animated: true)`

Build RepliFlix in Simulator to test if color backgrounds switch through tabs bars correctly.

Selecting icon images:

Search “house” in SF Symbols Explorer, then in `MainTabBarViewController`, set “house” symbol as our HomeViewController tab bar by doing:

```
vc1.tabBarItem.image = UIImage(systemName: “house”).
```

Set “play.circle” symbol as our UpcomingViewController tab bar by doing:

```
vc2.tabBarItem.image = UIImage(systemName: “play.circle”).
```

Set “magnifyingglass” symbol as our SearchViewController tab bar by doing:

```
vc3.tabBarItem.image = UIImage(systemName: “magnifyingglass”).
```

Set “arrow.down.to.line” symbol as our DownloadsViewController tab bar by doing:

```
vc4.tabBarItem.image = UIImage(systemName: “arrow.down.to.line”).
```

Build RepliFlix in Simulator to test if the tab bars images appear correctly.

Setting up the titles for 4 navigation bars:

Do `vc1/2/3/4.title = “Home/Coming Soon/Top Search/Downloads”`

Change all ViewControllers’ backgrounds into defaulted system themed color:

```
view.backgroundColor = .systemBackground
```

In `MainTabBarViewController,` add `tabBar.tintColor = .label` to add tint on bars.

HomeFeedTable prototype setup:

In `HomeViewController`, use the anonymous closure pattern to initialize our table to be a UI tableView setup:

```
: let table = UITableView()...
```

Set up our register to be a reusable identifier for further cell customization:

```
table.register(UITableViewCell.self, forCellResuseIdentifier: “cell”).
```

To add data, create `tableView` function, dequeue the cell with text content “Hello World”:

```
...tableView.dequeueReusableCell(withIdentifier: "cell", for:
indexPath)...cell.textLabel?.text = "Hello world"
```

A vertical list of “Hello World” should be outputting. Done setting up tableView.

Heights setup(Headers & Cells):

Leverage the delete methods to assign height as 200px to the header and between adjacent cells: `func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat { return 200}`

Add 4 new groups named Managers, Resources, Models and Views.

Add `CollectionViewTableViewCell` in the Cocoa class in the Views group.

Collection View Table Views Cells Management:

In `CollectionViewTableViewCell,` add the identifier, which is the same one as we used previously for registering the normal set: `static let identifier = "CollectionViewTableViewCell".`

Then initialize the cell style and reusable identifier:

```
override init...
super.init(style: style, reuseIdentifier: reuseIdentifier)
```

And implement the NSCoder as well.

Also, we need to specify the header to appear separately from cell blocks: `let table = UITableView(frame: .zero, style: .grouped)` with 40 px height.

In `CollectionViewTableViewCell,` we need to define a new connection view cell using the same anonymous closure pattern:

```
 private let collectionView: UICollectionView = {... }.
```

Cells Layouts arrangement:

In `CollectionViewTableViewCell:`

Set scroll directions are horizontal to navigate through categories:

```
layout.scrollDirection = .horizontal
```

Add an arbitrary cell name in the content view collection, and conform to those two protocols that allows us to display pictures and data inside the collection view.

We want our collection view to be the entire body of the cell, so we need to do:

```
override func layoutSubviews() {
        super.layoutSubviews()
        collectionView.frame = contentView.bounds }
```

Customize the size of each cell to get them independently aligned and horizontally scrollable:

```
layout.itemSize =  CGSize(width: 140, height: 200)
```

Adding & Modifying Headers to our TableView:

In `HomeViewController,` do `homeFeedTable.tableHeaderView = UIView(frame: CGRect(x: 0, y:0, width: view.bounds.width, height: 450))`

Then we can modify the header for each section by first creating a new file under the View folder in Cocoa class and name it as `HeroHeaderUIView`, which is in UIView.

Starting off by initializing frame and required initializer:

```
 override init(frame: CGRect){
        super.init(frame: frame)
    }
    required init?(coder: NSCoder){
        fatalError()
    }
```

Then we need a image that contains the header:

```
 private let heroImageView: UIImageView = {
```

Enable ImageView functionality:

```
let imageView = UIImageView()
```

We also want to avoid overflow:

```
imageView.clipsToBounds = true
```

Adding Poster Images with effects with GCColor:

Add a new image set and name the image in Assets, and in `HeroHeaderUIView`, add the image with its name: `imageView.image = UIImage(named: "image-name").` And in `HomeViewController,` we need to remove the arbitrary UIImageView setup and replace it with our hero image: `let headerView = HeroHeaderUIView(frame: CGRect(x:0, y:0, width: view.bounds.width, height: 450))`

To add fading-out effect to the header poster image, we need to utilize the gradient, we want to set the color to be clear by CGColor:

```
gradientLayer.colors = [
        UIColor.clear.cgColor,
        UIColor.systemBackground.cgColor]
```

Then we need to add this sub layer to our UIView and give the gradient a frame:

```
gradientLayer.frame = bounds
layer.addSublayer(gradientLayer)
```

Now we have the transparent-fading-out header poster.

Add two buttons for Play & Download:

Play button: set title, set color, set width, return:

```
private let playButton: UIButton = {
        let button = UIButton()
        button.setTitle("Play", for: .normal)
        button.layer.borderColor = UIColor.systemBackground.cgColor
        button.layer.borderWidth = 1
        return button
    }()
```

Also we need to set some constraints to align button perfectly:

`button.translatesAutoresizingMaskIntoConstraints = false`

Initialize the constraints as false so we can use it elsewhere further, so we need a function `applyConstraints` to setup playButton constraints:

Leading anchor left-right: `let playButtonConstraints = playButton.leadingAnchor.constraint(equalTo: leadingAnchor, constant: 90)`, to move the Play button upward, set the bottom anchor to negative value: `playButton.bottomAnchor.constraint(equalTo: bottomAnchor, constant: -50).` To modify width: `playButton.widthAnchor.constraint(equalToConstant: 100).` Activate the playButton constraints: `NSLayoutConstraint.activate(playButtonConstraints.`

Similarly layer setup for Download button. Besides, for aesthetic purposes, let's give the two buttons rounded corners with radius of 5: `button.layer.cornerRadius = 5`

Adding Netflix Logo:

Start off by defining a new configuration function: `private func configureNavbar(){`

Add Netflix logo PNG transparent image into Assets, to keep the image color original, we need: `image = image?.withRenderingMode(.alwaysOriginal)`

So the logo retains its red original color and dims when clicked.

Adding Right Bar Play Button and User Profile Button:

We need an array: `navigationItem.rightBarButtonItems = [`

RightBar play button: `UIBarButtonItem(image: UIImage(systemName: "person"),style: .done, target: self, action: nil),`

User Profile button: `UIBarButtonItem(image: UIImage(systemName: "play.rectangle"), style: .done, target: self, action: nil)`

We want the navigation bar to be pushed upwards and disappear as the user scrolls up to prevent it from hiding the contents down below. To make the navigation bar stick on the top, we need to implement such an algorithm: in the given template `func scrollViewDidScroll(_ scrollView: UIScrollView)`, use `navigationController?.navigationBar.transform = .init(translationX: 0, y: min(0, -offset))`, `-offset` indicates we are pushing the navigation bar upward.

Align Header Sections Titles:

Starting off by creating a array of strings: `let sectionTitles: [String] = ["Trending Movies", "Popular", "Trending Tv", "Upcoming Movies", "Top rated"]`

Also, we want these section titles to have proper fonts, frames and bounds, so we need to add a new method: `func tableView(_ tableView: UITableView, willDisplayHeaderView view: UIView , forSection section: Int ){...}`

```
header.textLabel?.frame = CGRect(x: header.bounds.origin.x + 20, y: header.bounds.origin.y, width: 100, height: header.bounds.height)
```

We need to change the section title color as well:

```
header.textLabel?.textColor = .white
```

Sending URL Requests and Parsing JSON response:

https://www.themoviedb.org/?language=en-US

We need to request an API v3 auth Key from the website to identify and authenticate our application. Then create a new Swift file in Manager named APICaller.

For simplicity, create a structure to copy and paste the API Key we just requested:

```
struct Constants {
    static let API_KEY  = "API_KEY"}
```

Create shared API instance: `class APICaller{`

```
    static let shared = APICaller()
```

Now we have our data, we need to convert our data into JSON objects to serialize it, without using any 3rd party network layering: `let results = JSONSerialization.jsonObject(with: data, options: .fragmentsAllowed)`

Hare, we passed the API Documentations data above and allowed fragments.

Then we need to add `getTrendingMovies` function to test if our API works well.

```
    private func getTrendingMovies(){
        APICaller.shared.getTrendingMovies {_ in }}}
```

Built from the console, then we have App connection to the database established successfully.

Now copy the movie JSON objects attributes into a structure named Movie:

```
struct Movie {id: Int media_type: String? original_name: String? original_title: String? poster_path: String? overview: String? vote_count: Int. release_date: String? vote_average: Double
```

Also conform the protocols with `Codable` to get the arrays of objects set up.

Using Extensions and Fetch and Retrieve API Datas from Database:

Here we need to use Xcode extensions to capitalize only the first letter for the section titles:

So in `Resources`, create a new file named `Extensions`, then create an function `func capitalizeFirstLetter() -> String {return self.prefix(1).uppercased()`

`+ self.lowercased().dropFirst()` to capitalize the first letter, drop off the capitalized first letter while lowering case the rest letters fo avoid duplicate by `dropFirst()`.
Similarly as above, we need to get trending TVs by creating function: `func getTrendingTvs(completion: @escaping (Result<[Tv], Error>)-> Void)` with a completion callback handler, we use `JSONSerialization` here instead to fetch data.
*To fetch data successfully, we must ensure to resume the task at the end of every API.
Thus, we retrieved all upcoming movies and trending movies from the database.
Repeating the same processes above, we thereby also fetched and retrieved data of Popular Movies and Top Rated Movies as `getPopular()` and `getTopRated()`

Refactoring Models:

To avoid duplicates, in `Models` group, merge `Movies` and `Tv` into `Title`, and also merge methods `TrendingMovieResponse()` and `TrendingTvResponse()` into `TrendingTitleResponse()`.

Creating Custom CollectionViewCell:

In Views group, create a new file under Cocoa Touch Class and name it as `TitleCollectionViewCell` to handle everything inside the CollectionViewCell.
Start a frame initizalier `override init(frame: CGRect) {super.init(frame: frame)` and `required init?(coder: NSCoder){fatalError()` to avoid fatal error.
Then use anonymous closure pattern to create UIImageView method for poster image:

```
  private let posterImageView: UIImageView = {
        let imageView = UIImageView()
        imageView.contentMode = .scaleAspectFill
        return imageView
```

We need an asynchronous image downloader with cache support as a UIImageView category, so from https://github.com/SDWebImage/SDWebImage.git , we add the SDWebImage package in GitHub target to our UIImage objects directly. Then in `TitleCollectionViewCell` add configuration as:

```
public func configure(with model: String) {
        guard let url = URL(string: model) else {return}
        posterImageView.sd_setImage(with: url, completed: nil)
```

Passing Data to the CollectionView:

We want every single section to handle its own cells, so to fetch API datas properly, we need a enumeration with all sections as distinct cases:

```
enum Sections: Int {
    case TrendingMovies = 0
    case TrendingTv = 1
    case Popular = 2
    case Upcoming = 3
    case TopRated = 4
```

To initialize our enumerators as raw values, in `HomeViewController`, we need a switch method to handle each title case:

```
switch indexPath.section{
        case Sections.TrendingMovies.rawValue:
```

```
        case Sections.TrendingTv.rawValue:
        case Sections.Popular.rawValue:
        case Sections.Upcoming.rawValue:
        case Sections.TopRated.rawValue:
```

Configure API Caller for each section title with providing them cases of success and errors:

```
            APICaller.shared.getTrendingMovies { result in
              switch result {
              case .success(let titles):
                  cell.configure(with: titles)
              case .failure(let error):
                  print(error.localizedDescription)
```

Then in our configuration function, we need to use: `self.titles = titles`

```
        DispatchQueue.main.async { [weak self] in
            self?.collectionView.reloadData()
```

method, since we have retrieved the section titles from the HomeViewController, so update and reload the titles array.

So now we passed data into CollectionView and got images for each section.

<u>Viewing poster images inside CollectionViewCell:</u>

Search TMDB API at https://developers.themoviedb.org/3/getting-started/introduction

Then in our configuration function, pass API images model:

```
 guard let url = URL(string:
"https://image.tmdb.org/t/p/w500/\(model)") else {
            return
        posterImageView.sd_setImage(with: url, completed: nil)
```

Where `image.tmdb.org/t/p/w500` is the API posters images model to pass.

<u>Creating Upcoming TableView inside Upcoming Tab:</u>

In UpcomingViewController, create a anonymous closure pattern function as Upcoming Table:

```
 private let upcomingTable: UITableView = {} ()
```

Set a normal register to itself:

```
table.register(UITableViewCell.self, forCellReuseIdentifier: "cell")
```

We need to make sure the data reloaded into upcoming table inside async function and be executed in main thread by adding: `DispatchQueue.main.async {`

```
                    self?.upcomingTable.reloadData()}
```

And to avoid the frequently occurred “switch must be exhaustive” error, we must append a failure case after every success to localized description:

```
   case .failure(let error):
                print(error.localizedDescription)
```

Remember to fetch upcoming movies data with `fetchUpcoming()`

viewing poster images inside CollectionViewCell.PNG

Creating custom TableViewCell from the upcoming table:

We need to create a UI Image so that it holds the poster for the title retrieved from the server.

So in `TitleTableViewCell` under Cocoa Touch class,

```
private let titlesPosterUIImageView: UIImageView = {
        let imageView =  UIImageView()
        imageView.contentMode = .scaleAspectFill
        return imageView
```

Where we need: `label.translatesAutoresizingMaskIntoConstraints = false`

And `imageView.translatesAutoresizingMaskIntoConstraints = false`

When setting up UIButton, we also need to activate this:

```
button.translatesAutoresizingMaskIntoConstraints = false
```

Similar to `HomeViewController`, we need a method to apply constraints:

```
  private func applyConstraints(){
        let titlesPosterUIImageViewConstraints = [
            titlesPosterUIImageView.leadingAnchor.constraint(equalTo:
contentView.leadingAnchor),
            titlesPosterUIImageView.topAnchor.constraint(equalTo:
contentView.topAnchor, constant: 15),
            titlesPosterUIImageView.bottomAnchor.constraint(equalTo:
contentView.bottomAnchor, constant: -15),

titlesPosterUIImageView.widthAnchor.constraint(equalToConstant: 100)
        ]
```

And to activate the layout constraints above:

```
 NSLayoutConstraint.activate(titlesPosterUIImageViewConstraints)
```

We also add the upcoming play title button constraints by:

```
   let playTitleButtonConstraints = [
            playTitleButton.trailingAnchor.constraint(equalTo:
contentView.trailingAnchor, constant: -20),
            playTitleButton.centerYAnchor.constraint(equalTo:
contentView.centerYAnchor)
```

And activate the play title button constraints with:

```
NSLayoutConstraint.activate(playTitleButtonConstraints)
```

play title button added in upcoming.PNG

Also modify the sizes, the colors and other alignments of the play buttons:

```
let image = UIImage(systemName: "play.circle", withConfiguration:
UIImage.SymbolConfiguration(pointSize: 30))
        button.setImage(image, for: .normal)
```

Then use `imageView.clipsToBounds = true` in `titlesPosterUIImageView` to prevent each poster from overflowing the container and decrease the padding slightly.

Creating Top Search TableView inside TopSearch tab:

Start off with:
```
override func viewDidLoad() {
        super.viewDidLoad()
        title = "Search"
        navigationController?.navigationBar.prefersLargeTitles = true
  navigationController?.navigationItem.largeTitleDisplayMode = .always
```
To create a bold solidary white title of “Search”.

Then to create a table view for Search section before the user entering a query:

Pass data with:
```
discoverTable.delegate = self
discoverTable.dataSource = self
```
Similar to `getUpcoming` and `getToprated`, we need a configured data fetching method:
```
  private func fetchDiscoverMovies() {
        APICaller.shared.getDiscoverMovies{[weak self] result in
            switch result {
            case .success(let titles):
                self?.titles = titles
                DispatchQueue.main.async {
                    self?.discoverTable.reloadData()
                }

            case .failure(let error):
                print(error.localizedDescription) }}}
    override func viewDidLayoutSubviews() {
        super.viewDidLayoutSubviews()
        discoverTable.frame = view.bounds }
```
DispatchQueue here still to override the main thread.

Creating SearchResultsViewController to display search results:

To read the database server once the user queries completed, we need to create a new file under `Controller` named `SearchResultsViewController.`

Create an anonymous closure pattern method to hold the search results controller:
```
private let searchController: UISearchController
```
Prompt searching request in the controller:
```
controller.searchBar.placeholder = "Search for a movie or a tv show"
```
Since we set the controller background color as systemGreen, when entering a user query in the placeholder, the current green background will pop out instead of a search result.

Use `layout.minimumInteritemSpacing = 0` to set the minimum intermittent spacing.

Possible error: SD_IMAGE BAD INSTRUCTION

Implement `SearchResultsViewController` to fix the error.

We need to adjust the simulator to iPhone 13 Pro for the screen to better fit searching blocks.

Querying database for individual movie:

To make the searching functionality works, go to Search & Query For Details in the Movie Database API, copy the url https://api.themoviedb.org/3/search/movie?api_key={api_key}&query=Jack+Reacher To fetch the querying searching data.

Then we need: `guard let query = query.addingPercentEncoding(withAllowedCharacters: .urlHostAllowed) else {return}` to properly format the url.

To get the search data updated itself, we need:

```
searchController.searchResultsUpdater = self
```

And to prevent Xcode from throwing errors, conform the protocols by adding extension:

```
extension SearchViewController: UISearchResultsUpdating
```

Along with prompting a non-empty and at least count of 3 (2 characters+) user query and also to minimize the cells:

```
!query.trimmingCharacters(in: .whitespaces).isEmpty,
query.trimmingCharacters(in: .whitespaces).count >= 3
```

We don't need a `weak self` in the switch here but only success and failure.

Configure the searching query to be a working array:

```
let title = titles[indexPath.row]
 cell.configure(with: title.poster_path ?? "")
```

Searching placeholder works.PNG

Using Youtube API:

Go to https://console.cloud.google.com.

On Dashboard, click API Services & API Overview -> Credentials -> Create a new project -> create credentials -> API Key created -> Enable APIs & Services -> YouTube Data API v3 -> enable YouTube Data API v3

In `APICaller` in Xcode, do `static let YoutubeAPI_KEY = API KEY`

Search Youtube Data API-> Content Search -> Show Code(right-hand-sided bar) -> HTTP

Copy https://youtube.googleapis.com/youtube/v3/search?key=[YOUR_API_KEY]

In Xcode: `static let YoutubeBaseURL = "https://youtube.googleapis.com/youtube/v3/search?"`

Then we matched successfully to get the response from the Youtube server.

Parsing YouTube API Response:

Noticeable error:

SD Image BAD INSTRUCTION error refer to: https://github.com/SDWebImage/SDWebImage/issues/3400

Failed to log metrics

In `HomeViewController,` do `APICaller.shared.getMovie(with: "RANDOM SEARCHING CONTENTS") { result in // }`

Then we got the best possible matches of the RANDOM SEARCHING CONTENTS.

In APICaller, pass completion success case by accessing the 1st element of items:

```
completion(.success(results.items[0]))
```

Pass completion failure case by: `completion(.failure(error))`

Handling selections of cells (Tapping on cells):

In `CollectionViewTableViewCell`, with:

```
  APICaller.shared.getMovie(with: titleName + "Trailer " ){ result in
            switch result {
            case .success(let videoElement):
                print(videoElement.id)
            case .failure(let error):
                print(error.localizedDescription)
```

Build and run, and if we click on a specific movie poster.

i.e. Minions: The Rise of Gru 2022

Output in console simultaneously:

```
IdVideoElement(kind: "youtube#video", videoId: "6DxjJzmYsXo")
```

If we copy the videoId and paste appending to https://www.youtube.com/watch?v=

Then we will be redirected to the movie's official trailer.

Creating TitlePreviewViewController:

Create a new UI Controller file named `TitlePreviewViewController` in General, then import WebKit with `private let webView: WKWebView = WKWebView()` to enable viewing the trailer videos in our RepliFlix UI.

Initialize the UI titles labels and return labels:

```
    private let titleLabel: UILabel = {
        let label = UILabel()
        return label }()
```

We set `label.numberOfLines = 0` to assign labels to have multiple lines in Swift.

Initialize Downloads button:

```
  private let downloadButton: UIButton = {
        let button = UIButton()
        button.translatesAutoresizingMaskIntoConstraints = false
        return button
```

Now for the controller, we need to add several subview attributes:

```
  view.addSubview(webView)
        view.addSubview(titleLabel)
        view.addSubview(overviewLabel)
        view.addSubview(downloadButton)
```

We also need to set up for webview constraints:

```
 func configureConstraints(){
        let webViewConstraints = [
            webView.topAnchor.constraint(equalTo: view.topAnchor),
            webView.leadingAnchor.constraint(equalTo:
view.leadingAnchor),
            webView.trailingAnchor.constraint(equalTo:
view.trailingAnchor)
```

And activate the web view constraints above with:

```
 NSLayoutConstraint.activate(webViewConstraints)
```
We also need some constraints for the title labels:
```
    let titleLabelConstraints = [
            titleLabel.topAnchor.constraint(equalTo:
webView.bottomAnchor, constant: 20),
            titleLabel.leadingAnchor.constraint(equalTo:
view.leadingAnchor, constant: 20),]
```
And activate it with: `NSLayoutConstraint.activate(webViewConstraints)`
we also need some constraints for the overview title labels:
```
let overviewLabelVConstraints = [
            overviewLabel.topAnchor.constraint(equalTo:
titleLabel.bottomAnchor, constant: 15),
            overviewLabel.leadingAnchor.constraint(equalTo:
view.leadingAnchor, constant: 20),]
```
And activate it with: `NSLayoutConstraint.activate(overviewLabelVConstraints)`
Create a new file named `TitlePreviewViewModel` in `ViewModel`
In `HomeViewController`, navigate to push the animated Title Preview View Model with:
```
navigationController?.pushViewController(TitlePreviewViewController(),
animated: true)
```
With some constraints, in `TitlePreviewViewController`, we define:
```
 private let webView: WKWebView = {
        let webView = WKWebView()
        webView.translatesAutoresizingMaskIntoConstraints = false
        return webView
```
We also need marginal constraints for the Download button:
```
 let downloadButtonConsraints = [
            downloadButton.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
            downloadButton.topAnchor.constraint(equalTo:
overviewLabel.bottomAnchor, constant: 25)
```
And activate it with: `NSLayoutConstraint.activate(downloadButtonConsraints)`
Customize the layer radius of the download button:
```
button.layer.cornerRadius = 8
button.layer.masksToBounds = true
```
We need to pass YouTube embedded data to web view with the following url:
```
   func configure(with model: TitlePreviewViewModel ){
        titleLabel.text = model.title
        overviewLabel.text = model.titleOverview
        guard let url = URL(string:
"https://www.youtube.com/embed/)\(model.youtubeView.id.videoId)")
else{return}
          webView.load(URLRequest(url: url))
```

Also in `CollectionViewController`, we need to create a protocol:

```
protocol CollectionViewTableViewCellDelegate: AnyObject{
    func collectionViewTableViewCellDidTapCell(_ cell:
CollectionViewTableViewCell, viewModel: TitlePreviewViewModel)}
```

We also need an optional weak delegate of the protocol:

```
weak var delegate: CollectionViewTableViewCellDelegate?
```

Then we need an extension in `HomeViewControlle`r to conform it:

```
extension HomeViewController: CollectionViewTableViewCellDelegate {
    func collectionViewTableViewCellDidTapCell(_ cell:
CollectionViewTableViewCell, viewModel: TitlePreviewViewModel) {
        let vc = TitlePreviewViewController()
        vc.configure(with: viewModel)
        navigationController?.pushViewController(vc, animated: true)}}
```

Build and run we then have **homeview.PNG**

To make the posters clickable, set the delegate to controller, and modify the extension in `HomeViewController` by placing the current contents into a weak self of dispatch queue:

```
DispatchQueue.main.async { [weak self] in
            let vc = TitlePreviewViewController()
            vc.configure(with: viewModel)
            self?.navigationController?.pushViewController(vc,
animated: true)
```

Now we successfully achieve that functionality of being redirected to a certain movie's trailer playing and downloading page when clicking on it:

I.e. Thor: Love and Thunder 2022

Before redirecting.PNG after redirecting.PNG

We also need to add `overviewLabel.trailingAnchor.constraint(equalTo: view.trailingAnchor)` to better modify the overview alignment.

After modifying overview alignment constraint.PNG

Error:

May encounter WKWebView ViewportSizing logs in SwiftUI warning with `[ViewportSizing] maximumViewportInset cannot be larger than frame`

To fix this:

Replace `let webView = WKWebView()` to `let webView = WKWebView(frame: CGRect(x: 0.0, y: 0.0, width: 0.1, height: 0.1))`

Reference:

https://stackoverflow.com/questions/73314364/wkwebview-viewportsizing-logs-in-swiftui

Refactoring TableViewHeader Hero title:

Fetch random trending movies with this function:

```
    private func configureHeroHeaderView() {
        APICaller.shared.getTrendingMovies { [weak self] result in
            switch result {
            case .success(let titles):
```

```
                self?.randomTrendingMovie =  titles.randomElement()
            case .failure(let error):
                print(error.localizedDescription)}}}
```

Also in order to generate the random trending element in Home View, in Hero View:

```
  private func configureHeroHeaderView(){
        APICaller.shared.getTrendingMovies { [weak self] result in
            switch result {
            case .success(let titles):
                let selectedTitle =  titles.randomElement()
                self?.randomTrendingMovie =  selectedTitle
                self?.headerView?.configure(with:
TitleViewModel(titleName: selectedTitle?.original_title ?? "",
posterURL: selectedTitle?.poster_path ?? ""))
            case .failure(let error):
                print(error.localizedDescription)}}
```

Now we successfully generated different random trending movies in Home View page, and each time we reopen the app, there are different trending movie being generated:

random element generated1.PNG random element generated2.PNG

Handling Tapping across all ViewControllers:

In `UpcomingViewController,` do:

```
  APICaller.shared.getMovie(with: titleName) {[weak self ] result in
            switch result {
            case .success(let videoElement):
                DispatchQueue.main.async {
                    let vc = TitlePreviewViewController()
                    vc.configure(with: TitlePreviewViewModel(title:
titleName, youtubeView: videoElement, titleOverview: title.overview ??
""))
                    self?.navigationController?.pushViewController(vc,
animated: true)}
            case .failure(let error):
             print(error.localizedDescription)
```

Then build and run, in the simulator, we will also be redirected to the trailer playing and downloading page of a specific movie if we click it on Upcoming, just like HomeView.

Upcoming Done.

In `SearchViewController,` copy and paste the same table view function.

TopSearch Done.

In the `SearchResultsViewController,` we need to create a new protocol:

```
protocol SearchResultsViewControllerDelegate: AnyObject {
    func SearchResultsViewControllerDidTapItem(_ viewModel:
TitlePreviewViewModel)}
```

And make it an accessible public weak var delegate:

```
public weak var delegate: SearchResultsViewControllerDelegate?
```

Now configure to conform the searching view results in `SearchViewController:`

searching bar searching results showing.PNG

We also want to be redirected to a downloading page when we click on the home view posters. So in `CollectionViewTableViewCell,` we need to add a downloading action:

```
func collectionView(_ collectionView: UICollectionView,
contextMenuConfigurationForItemAt indexPath: IndexPath, point:
CGPoint) -> UIContextMenuConfiguration? {
        let config = UIContextMenuConfiguration(
            identifier: nil,
            previewProvider: nil){ _ in
                let downloadAction = UIAction(title: "Download",
subtitle: nil, image: nil, identifier: nil,
                    discoverabilityTitle: nil, state: .off) { _ in
                    print("Download tapped")
                }
                return UIMenu(title: "", image: nil, identifier: nil,
options: .displayInline, children: [downloadAction])
            }
        return config }}
```

So that in Home View, when we click on any poster and hold for couple second, there will appear a download option for us to be redirected to Downloads bar for the movie:

Long press download action.PNG

Add optional downloading path:

```
   private func downloadTitleAt(indexPath: IndexPath){
        print("Downloading \(titles[indexPath.row].original_title)")}}
```

Core Data:

Add core data to an existing project:

Create a new test project with Core Data selected.

Then in its `AppDelegate,` copy the whole CORE DATA STACK.

Paste the whole core data stack into the class in `AppDelegate` within the RepliFlix project.

Add `import CoreData` at the top.

Create a new Core Data file under RepliFlix path and name it as `RepliFlixModel.`

And change `let container = NSPersistentContainer(name: "RepliFlixModel")` in `AppDelegate.`

`Add Entity` in `RepliFlixModel,` and rename it as `TitleItem.`

Add all the attributes from `Title` to the entity.

To ensure `TitleItem` is accessible in `CollectionViewTableCell,` close Xcode and reopen it. Now the entity `TitleItem` should be fully accessible publicly.

To implement core data:

First create a new swift file under `Manager` and name it `DataPersistenceManager.`

So this class will be responsible for downloading the core data and talking to its API.

Add an instance/reference `guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else { return}` to the app delegate, with context: `let context = appDelegate.persistentContainer.viewContext`

Create a new item to pass all `Title` attributes into the item:

```
let item = TitleItem(context: context)
        item.original_title = model.original_title
        item.id = Int64(model.id)
        item.original_name = model.original_name
        item.overview = model.overview
        item.media_type = model.media_type
        item.poster_path = model.poster_path
        item.release_date = model.release_date
        item.vote_count = Int64(model.vote_count)
        item.vote_average = model.vote_average
```

Then manage to save the data with:

```
  do{try context.save()} catch {print(error.localizedDescription)}
```

Pass EMPTY to completion success case: `completion(.success(()))`

For the completion failure case, create a enum:

```
    enum DatabasError: Error{
      case failedToSaveData }
```

And `completion(.failure(DatabasError.failedToSaveData))`

Now we need to fetch the data from the server database:

```
   func fetchingTitleFromDataBase(completion: @escaping
(Result<[TitleItem], Error>) -> Void){
        guard let appDelegate = UIApplication.shared.delegate as?
AppDelegate else {
            return}
        let context = appDelegate.persistentContainer.viewContext
        let request: NSFetchRequest<TitleItem>
        request = TitleItem.fetchRequest()
             do {
            let titles = try context.fetch(request)
            completion(.success(titles))
        } catch {
            completion(.failure(DatabasError.failedToFetchData))
```

Now we have our data fetching request sent and done.

Fetch local storage for download in `DownloadsViewController:`

```
private func fetchLocalStorageForDownload(){
        DataPersistenceManager.shared.fetchingTitleFromDataBase {}
```

And also reload the data while fetching in:

```
 self?.downloadedTable.reloadData()
```

Also for the downloaded table view to show, we need to add `view.addSubview(downloadedTable)` to the `ViewDidLoad` in `DownloadViewController.`

Now we need to ask database manager to delete certain object:

```
context.delete(model)
```

We also need to implement a function in `DownloadsViewController` to delete data:

```
    func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
        switch editingStyle{
        case .delete:
           DataPersistenceManager.shared.deleteTitleWith(model:
titles[indexPath.row]) { [weak self] result in
                switch result{
                case .success():
                    print("Delete from the database")
                case .failure(let error):
                    print(error.localizedDescription)
                }
                self?.titles.remove(at: indexPath.row)
                tableView.deleteRows(at: [indexPath], with: .fade)
            }
        default:
            break; }
```

So we are able to delete the already downloaded movies in the Downloads bar.

Using Notification Center to update ViewControllers:

```
NotificationCenter.default.addObserver(forName:
NSNotification.Name("downloaded"), object: nil, queue: nil) { _ in
            self.fetchLocalStorageForDownload()}
```

So the system will be notified when a certain movie has finished downloading.