
URLImage



[URLImage](#) is a SwiftUI view that displays an image downloaded from provided URL. [URLImage](#) manages downloading remote image and caching it locally, both in memory and on disk, for you.

Using [URLImage](#) is dead simple:

```
1 URLImage(url: url) { image in
2     image
3     .resizable()
4     .aspectRatio(contentMode: .fit)
5 }
```

Take a look at some examples in the demo app.

Table of Contents

- Features
- Installation
- Usage
 - View Customization
 - Options
 - Image Information
 - Zoom In
- Cache
 - Store Use cases
- Advanced
 - Start Loading
 - Make Your Own URLImage
 - Fetching an Image
 - Download an Image in iOS 14 Widget
- Migration Notes v2 to v3
- Common Issues
- Reporting a Bug
- Requesting a Feature
- Contributing

Features

- SwiftUI image view for remote images;
- Local image cache;
- Fully customizable including placeholder, progress indication, error, and the image view;
- Control over various download aspects for better performance.

Installation

`URLImage` can be installed using Swift Package Manager.

1. In Xcode open **File/Swift Packages/Add Package Dependency...** menu.
2. Copy and paste the package URL:

```
1 https://github.com/dmytro-anokhin/url-image
```

For more details refer to Adding Package Dependencies to Your App documentation.

Usage

You can create `URLImage` with URL and a `ViewBuilder` to display downloaded image.

```
1 import URLImage // Import the package module
2
3 let url: URL = //...
4
5 URLImage(url) { image in
6     image
7     .resizable()
8     .aspectRatio(contentMode: .fit)
9 }
```

Note: first argument of the `URLImage` initialiser is of `URL` type, if you have a `String` you must first create a `URL` object.

View Customization

`URLImage` view manages and transitions between 4 download states:

- Empty state, when download has not started yet, or there is nothing to display;
- In Progress state to indicate download process;
- Failure state in case there is an error;

-
- Content to display the image.

Each of this states has a separate view. You can customize one or more using `ViewBuilder` arguments.

```
1 UIImage(item.imageUrl) {
2     // This view is displayed before download starts
3     EmptyView()
4 } inProgress: { progress in
5     // Display progress
6     Text("Loading...")
7 } failure: { error, retry in
8     // Display error and retry button
9     VStack {
10         Text(error.localizedDescription)
11         Button("Retry", action: retry)
12     }
13 } content: { image in
14     // Downloaded image
15     image
16         .resizable()
17         .aspectRatio(contentMode: .fit)
18 }
```

Options

`UIImage` allows to control certain aspects using `UIImageOptions` structure. Things like when-ever to download image or use cached, when to start and cancel download, how to configure network request, what is the maximum pixel size, etc.

`UIImageOptions` is the environment value and can be set using `\.urlImageOptions` key path.

```
1 UIImage(url) { image in
2     image
3         .resizable()
4         .aspectRatio(contentMode: .fit)
5 }
6 .environment(\.urlImageOptions, UIImageOptions(
7     maxPixelSize: CGSize(width: 600.0, height: 600.0)
8 ))
```

Setting `UIImageOptions` in the environment value allows to set options for a whole or a part of your views hierarchy.

```
1 @main
2 struct MyApp: App {
```

```

3
4     var body: some Scene {
5         WindowGroup {
6             ContentView()
7                 .environment(\.urlImageOptions, URLImageOptions(
8                     maxPixelSize: CGSize(width: 600.0, height: 600.0)
9                 ))
10        }
11    }
12 }
```

Image Information

You can use `ImageInfo` structure if you need information about an image, like actual size, or access the underlying `CGImage` object. `ImageInfo` is an argument of `content` view builder closure.

```

1  UIImage(item.imageUrl) { image, info in
2      if info.size.width < 1024.0 {
3          image
4              .resizable()
5              .aspectRatio(contentMode: .fit)
6      } else {
7          image
8              .resizable()
9              .aspectRatio(contentMode: .fill)
10     }
11 }
```

Zoom In

If you want to add ability to scale the image consider checking `AdvancedScrollView` package.

```

1  import AdvancedScrollView
2  import UIImage
3
4  UIImage(url) { image in
5      AdvancedScrollView(magnificationRange: 1.0...4.0) { _ in
6          image
7      }
8  }
```

Cache

`UIImage` can also cache images to lower network bandwidth or for offline use.

By default, `URLImage` uses protocol cache policy, i.e. Cache-Control HTTP header and `URLCache`. This corresponds to how images work on web and requires network connection.

Alternatively, if you want to view images offline, you must configure the file store. When configured, `URLImage` will no longer use protocol cache policy, and instead follow `URLImageOptions.FetchPolicy` setting.

```
1 import URLImage
2 import URLImageStore
3
4 @main
5 struct MyApp: App {
6
7     var body: some Scene {
8
9         let urlImageService = URLImageService(fileStore:
10             URLImageFileStore(),
11                                             inMemoryStore:
12             URLImageInMemoryStore())
13
14         return WindowGroup {
15             FeedListView()
16             .environment(\.urlImageService, urlImageService)
17         }
18     }
19 }
```

Make sure to include `URLImageStore` library under “Frameworks, Libraries, and Embedded Content” of your target settings.

Store Use Cases

You may ask when to use protocol or custom cache. `URLImage` designed to serve two use cases:

Use protocol cache policy when an app can only work connected to the internet. Ecommerce apps, such as shopping, travel, event reservation apps, etc., work like this. Following protocol cache policy you can be sure that images are cached in a way that your CDN defines, can still be accessed quickly, and don't take unnecessary space on user devices.

Configure `URLImageStore` for content that needs to be accessed offline or downloaded in background. This can be a reader app, you probably want to download articles before user opens them, maybe while the app is in the background. This content should stay for a considerably long period of time.

Advanced

Start Loading

`URLImage` starts loading when the image view is rendered. In some cases (like with `List`) you may want to start loading when view appears and cancel when it disappears. You can customize this using `URLImageOptions.LoadOptions` options. You can combine multiple to achieve behaviour that fits your UI best.

```
1 List(/* ... */) {
2     // ...
3 }
4     .environment(\.urlImageOptions, URLImageOptions(loadOptions: [ .
        loadOnAppear, .cancelOnDisappear ]))
```

Note: versions prior to 3.1 start loading on appearance and cancel when view disappears. Version 3.1 starts loading when the view renders. This is because `onAppear` and `onDisappear` callbacks are quite unpredictable without context.

Make Your Own URLImage

Alternatively you can make your own `URLImage` to customize appearance and behaviour for your needs.

```
1 struct MyURLImage: View {
2
3     @ObservedObject private var remoteImage: RemoteImage
4
5     init(service: URLImageService, url: URL) {
6         remoteImage = service.makeRemoteImage(url: url, identifier: nil
7         , options: URLImageOptions())
8     }
9
10    var body: some View {
11        ZStack {
12            switch remoteImage.loadingState {
13                case .success(let value):
14                    value.image
15
16                default:
17                    EmptyView()
18            }
19        }
20        .onAppear {
21            remoteImage.load()
22        }
23    }
24 }
```

```
22     }
23 }
```

You can access service environment value from enclosing view: `@Environment(\.urlImageService)`
`var service: URLImageService.`

Fetching an Image

You may want to download an image without a view. This is possible using the `RemoteImagePublisher` object. The `RemoteImagePublisher` can cache images for future use by the `URLImage` view.

Download an image as `CGImage` and ignore any errors:

```
1 cancellable = URLImageService.shared.remoteImagePublisher(url)
2   .tryMap { $0.cgImage }
3   .catch { _ in
4       Just(nil)
5   }
6   .sink { image in
7       // image is CGImage or nil
8   }
```

Download multiple images as an array of `[CGImage?]`:

```
1 let publishers = urls.map { URLImageService.shared.remoteImagePublisher
2   ($0) }
3 cancellable = Publishers.MergeMany(publishers)
4   .tryMap { $0.cgImage }
5   .catch { _ in
6       Just(nil)
7   }
8   .collect()
9   .sink { images in
10       // images is [CGImage?]
11   }
```

When downloading image using the `RemoteImagePublisher` object all options apply as they do for the `URLImage` object. By default downloaded image will be cached on the disk. This can speed up displaying images on later stage of your app. Also, this is currently the only supported way to display images in iOS 14 widgets.

Download an Image in iOS 14 Widget

Unfortunately views in WidgetKit can not run asynchronous operations: <https://developer.apple.com/forums/thread/>
The recommended way is to load your content, including images, in `TimelineProvider`.

You can still use `URLImage` for this. The idea is that you load image in `TimelineProvider` using the `RemoteImagePublisher` object, and display it in the `URLImage` view.

Migration Notes v2 to v3

- `URLImage` initialiser now omits an argument label for the first parameter, making `URLImage(url: url)` just `URLImage(url)`.
- `URLImage` initialiser now uses `ViewBuilder` attribute for closures that construct views.
- `URLImageOptions` now passed in the environment, instead of as an argument. Custom identifier can still be passed as an argument of `URLImage`.
- By default `URLImage` uses protocol cache policy and `URLCache`. This won't store images for offline usage. You can configure the file store as described in cache section.
- Swift Package Manager is now the only officially supported dependency manager.

Common Issues

Image reloads when view reloads

This is a common issue if you use `URLImage` alongside `TextField` or another control that updates a state that triggers view update. Because `URLImage` is asynchronous and initially empty, it will reset to empty state before displaying downloaded image. To avoid this, setup `URLImageInMemoryStore` somewhere in your `App`.

```
1 import SwiftUI
2 import URLImage
3 import URLImageStore
4
5 @main
6 struct MyApp: App {
7     var body: some Scene {
8         let urlImageService = URLImageService(fileStore: nil,
9             inMemoryStore: URLImageInMemoryStore())
10
11         return WindowGroup {
12             ContentView()
13             .environment(\.urlImageService, urlImageService)
14         }
15     }
16 }
```

Note: you can reset cached image using `removeImageWithURL`, `removeImageWithIdentifier`, or `removeAllImages` methods of `URLImageInMemoryStore`.

Image in navigation/toolbar displayed as single color rectangle

This is not a bug. Navigation/toolbar uses `.renderingMode(.template)` to display images as templates (renders all non-transparent pixels as the foreground color). The way to reset it is to specify `.renderingMode(.original)`:

```
1 UIImage(url) { image in
2     image.renderingMode(.original)
3 }
```

Reporting a Bug

Use GitHub issues to report a bug. Include this information when possible:

Summary and/or background; OS and what device you are using; Version of UIImage library; What you expected would happen; What actually happens; Additional information: Screenshots or video demonstrating a bug; Crash log; Sample code, try isolating it so it compiles without dependancies; Test data: if you use public resource provide URLs of the images.

Please make sure there is a reproducible scenario. Ideally provide a sample code. And if you submit a sample code - make sure it compiles ;)

Requesting a Feature

Use GitHub issues to request a feature.

Contributing

Contributions are welcome. Please create a GitHub issue before submitting a pull request to plan and discuss implementation.