Comments Feature

Goals

1. Display a list of comments when the user taps on an image in the feed.
2. Loading the comments can fail, so you must handle the UI states accordingly.
    - Show a loading spinner while loading the comments.
    - If it fails to load: Show an error message.
    - If it loads successfully: Show all loaded comments in the order they were returned by the remote API
3. The loading should start automatically when the user navigates to the screen.
    - The user should also be able to reload the comments manually (Pull-to-refresh)
4. At all times, the user should have a back button to return to the feed screen.
    - Cancel any running comments API requests when the user navigates back.
5. The comments screen layout should match the UI specs.
    - Present the comment date using relative date formatting, e.g., "1 day ago."
6. The comments screen title should be localized in all languages supported in the project
7. The comments screen should support Light and Dark Mode
8. Write tests to validate your implementaton, including unit, integration, and snapshot tests (aim to write the test first!).

API Specs

Payload Contract

```
GET /image/{image-id}/comments

2xx Response

{
  "items": [
    {
      "id": "a UUID",
      "message": "a message",
      "created_at": "2020-05-20T11:24:59+0000",
      "author": {
        "username": "a username"
      }
      },
```

```
      {
      "id": "another UUID",
      "message": "another message",
      "created_at": "2020-05-19T14:23:53+0000",
      "author": {
        "username": "another username"
      }
      },
      ...
  ]
}
```

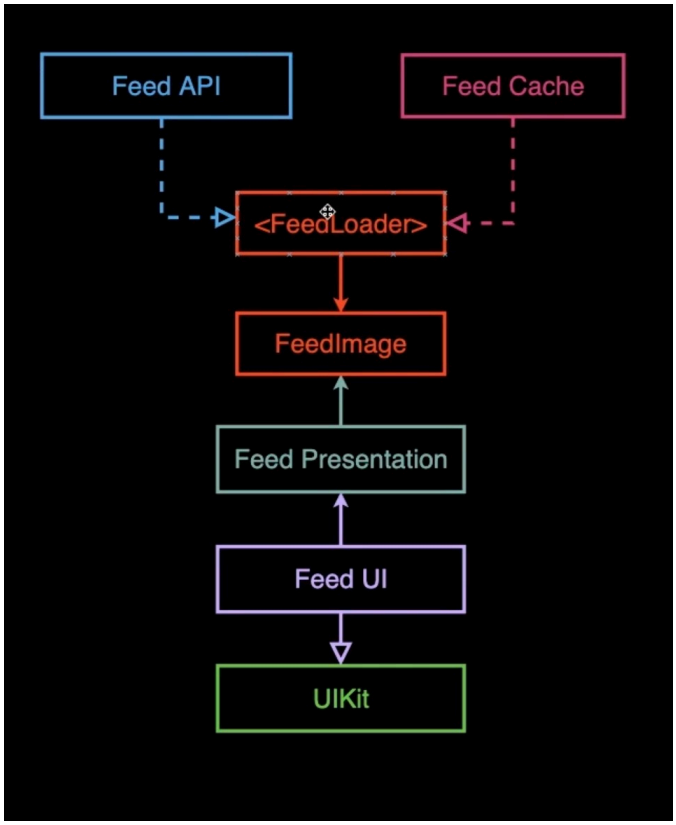
Feed Image Comment

Property	Type
`id`	`UUID`
`message`	`String`
`created_at`	`Date` (ISO8601 String)
`author`	`CommentAuthorObject`

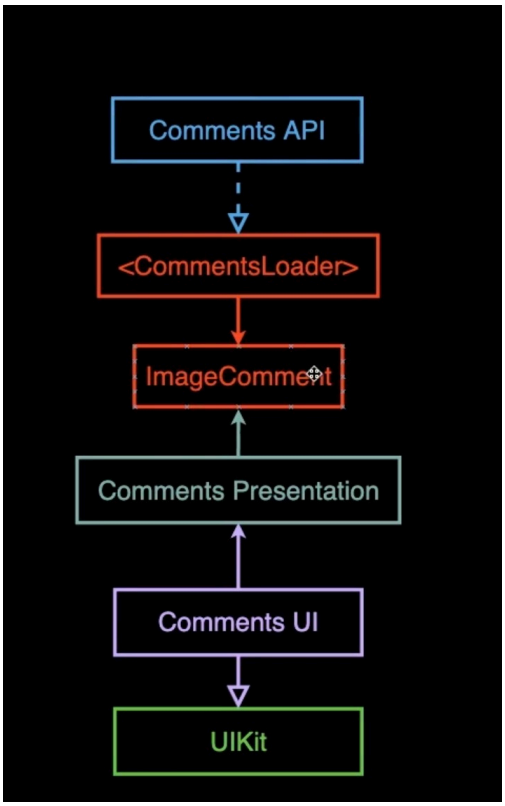
Feed Image Comment Author

Property	Type
`username`	`String`

Architecture for the APP

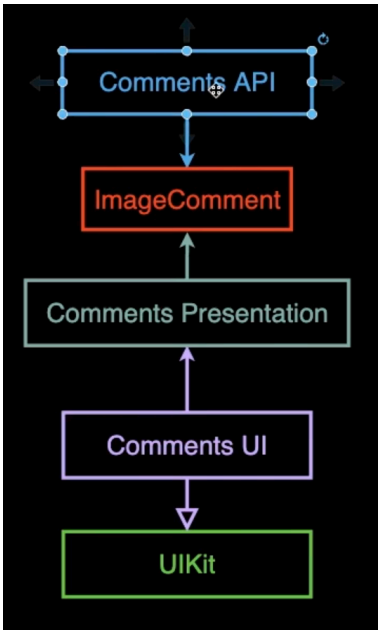


First Approach to the architecture



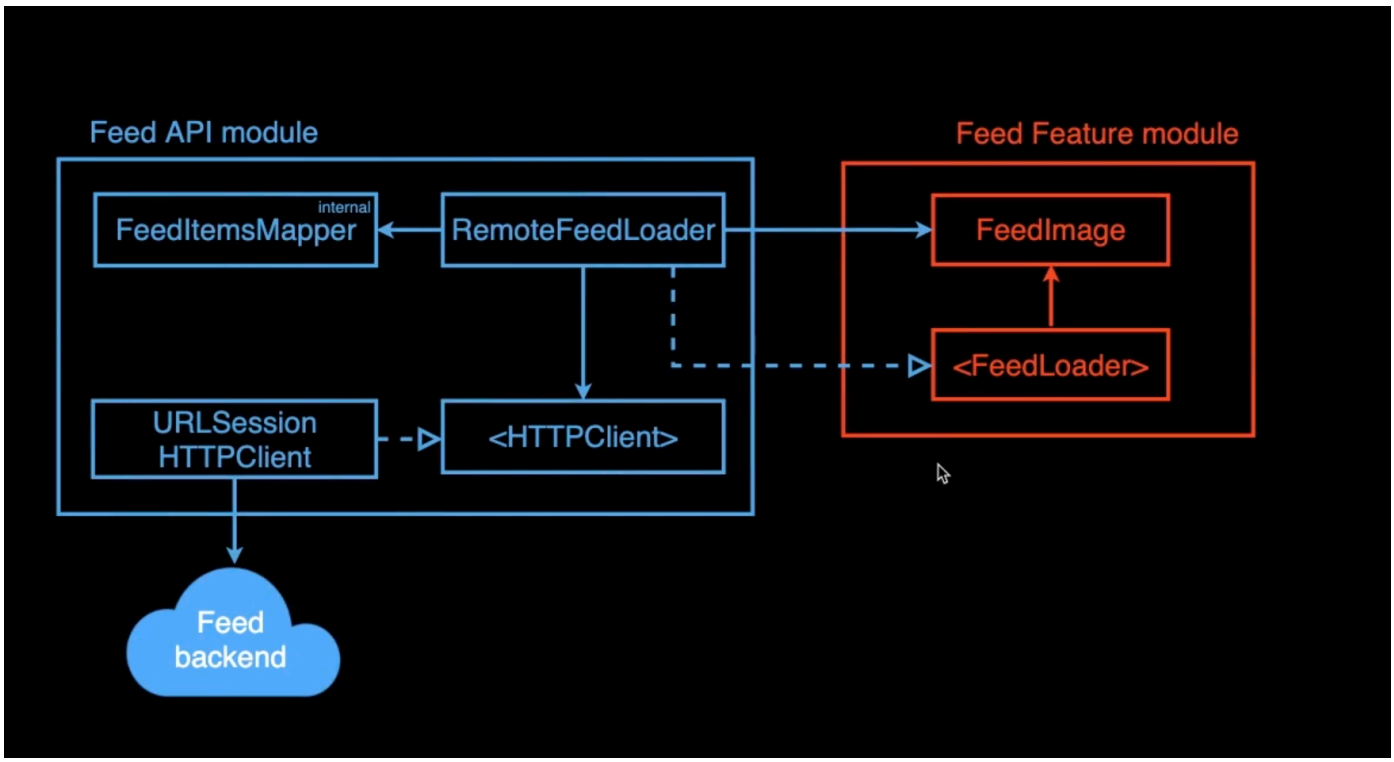
In Reality we dont need to have the **Comments Loader Protocol** since we wont be caching, usually protocols/interfaces are good for when we have more than one use of it. Its always good to simplify things when possible.

Final Diagram

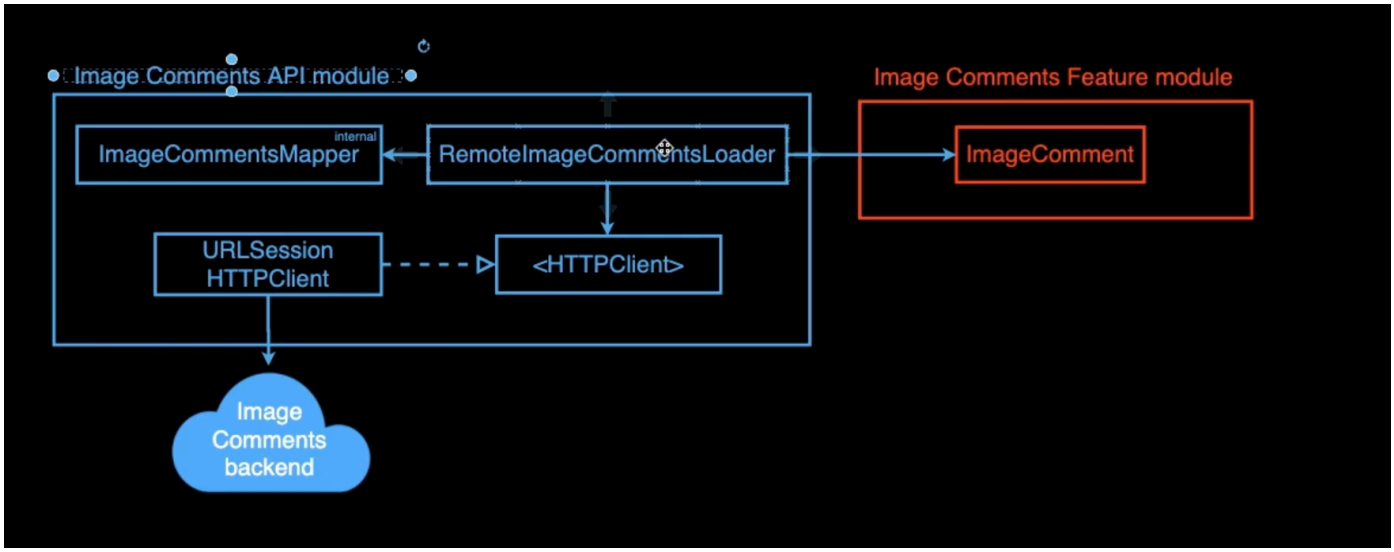


How can we create new features without duplicating code/ duplicating HTTPClients?

Starting with the architecture for the already existing FeedModule Feature:



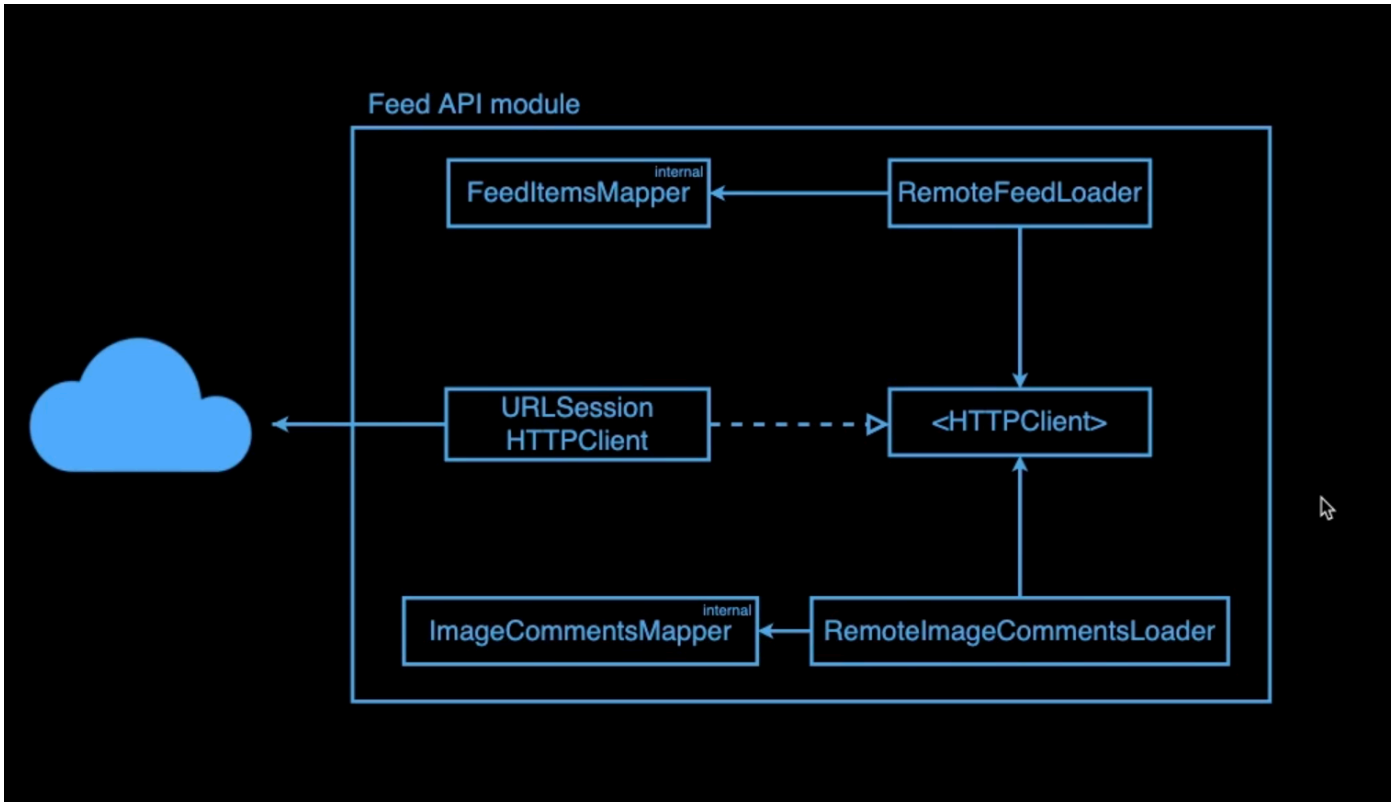
We can follow the same design to approach the Comments section:



Upsides: These both modules are decoupled. Since we dont need multiple strategies, we dont need the **protocol**.

Downsides: There is **two** of everything.

How can we do to avoid duplication?. --> Simplest thing we can do is to keep them in the same module, so the can therefore share the same types:

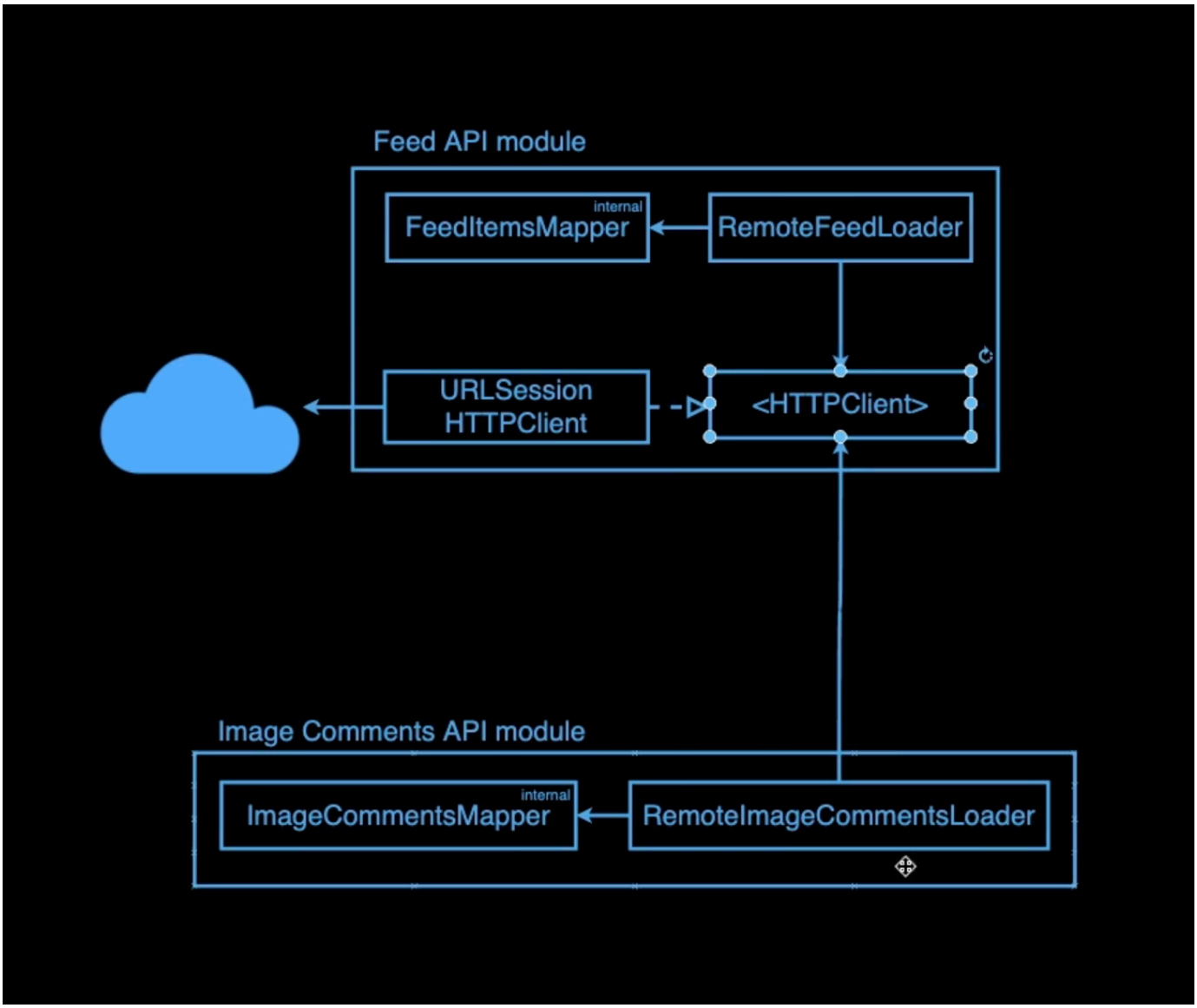


But then we lose the modularity that we want. Nonetheless its a good starting point.

- First we implement the `RemoteImageCommentsLoader`

Implementing `ImageCommentsMapper` and `RemoteImageCommentsLoader` is great, but we are still depending on the same `<HttpClient>`, what if we wanted to reuse the `Image Comments API` in some other place?

Well, we could move it into another module:



But it would still depend on the `<HTTPClient>`.

First idea:

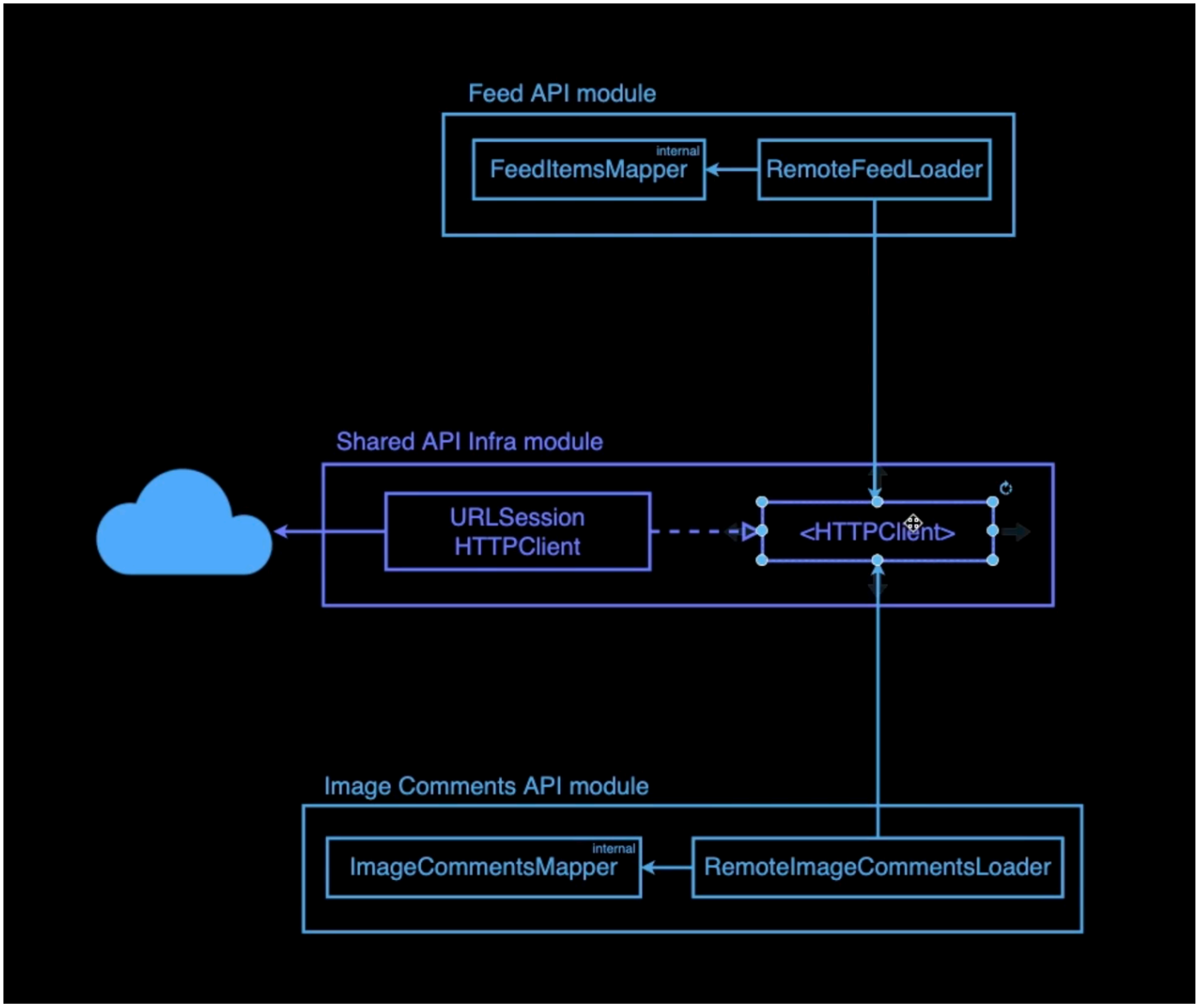
We create **two** HTTP client protocols:

This is actually a great idea, since the `ImageComments API` could want to post comments in the future, whereas the `Feed API` module doesn't, therefore it would make sense to have two different `HTTPClients` that work for every use case. Otherwise, if we used the same `HTTPClient` for both use cases, we would be breaking the **interface segregation principle** (clients should not depend on methods they do not need), since we would have to add methods to post data (comments) but the `Feed API` would not implement them.

But, we still share the same Infrastructure implementation, although neither of the API modules knows about this. This is what's called **dependency inversion principle**, where our Shared API Infra module doesnt depend on any implementation, but on abstractions, in this case `<HTTPClient>`.

Second idea:

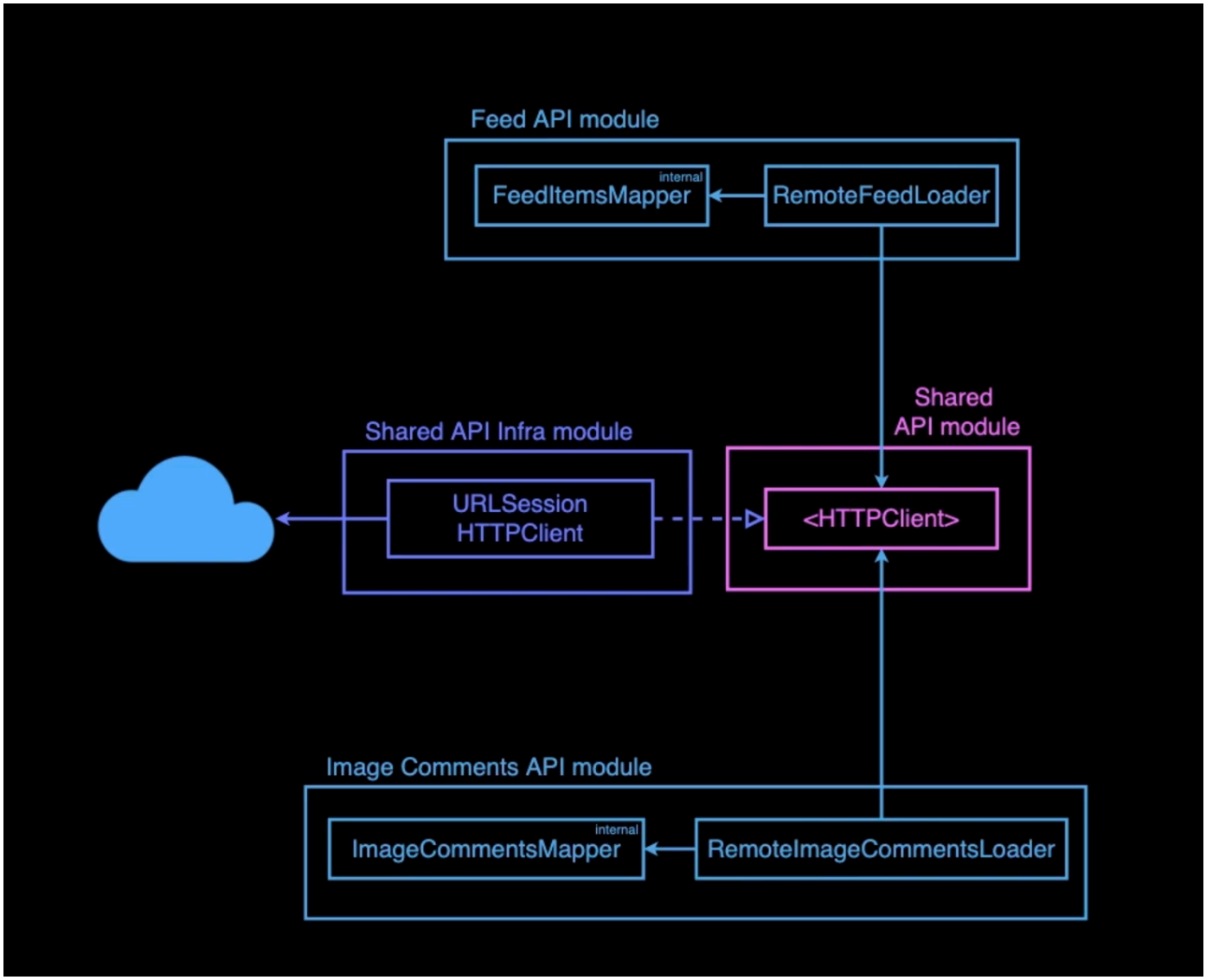
We might not want to duplicate the client, we may want to share it:



So what we would do is move the `<HTTPClient>` to the infrastructure module. This way they would share the client, but not depend on eachother, only on the shared infra module. The problem is that now those API modules depend on infrastructure details like the frameworks `(URLSession/HTTPCLient)`, which means we could not freely move the `Feed API` or the Image `Comments API` modules to another project without dragging the Shared API infra module, which would be a problem.

Third idea:

Create a new module:



We could create a module comprised of pure interfaces. This way, while both the API modules would depend on the shared API module, they would not depend on any infrastructure implementation details. But the downside of this is that we end up having too many components in our architecture which we have to mantain. We have great modularity, but really annoying maintenance (constant redeploy-recompile)

In our case, since we will only have one module comprising everything we will go ahead with this last implementation, using folders to separate the modules. But we will have to have discipline to do this. For this we will create a Shared API folder in the root directory of our **FeedFramework** project folder, where we will place our `<HTTPClient>` protocol, representing the aforementioned diagram. We also create the Shared API Infra folder that will contain the implementation, to which we move the `URLSessionHTTPClient` class.

We do the same for the other components, creating a **Image Comments API** folder, and moving the corresponding files there. We apply the same logic for all the tests.

Duplication in the Loaders

We fixed our duplication in the `<HTTPClient>` but we still have a lot of duplication in our `RemoteFeedLoader` and our `RemoteIImageCommentsLoader` which look exactly the same with minor variations in the mapping.

Duplicating the code is not all bad though, because without duplicating the code and analyzing what are the commonalities and similarities, we probably will end up arriving to the wrong abstractions because without comparison it's hard to see what is the same and what is not.

The way to solve this is creating a generic **RemoteLoader** and then inject the mapping for every case (which is what is different).

Once we've made the Generic **RemoteLoader** , a very cunning thing we can do is to typealias our **RemoteImageCommentsLoader** with the specific type ([**ImageComment**]). We can do this because typealiasing doesnt break any clients. We can then make an extension to this typealias and add a convenience init:

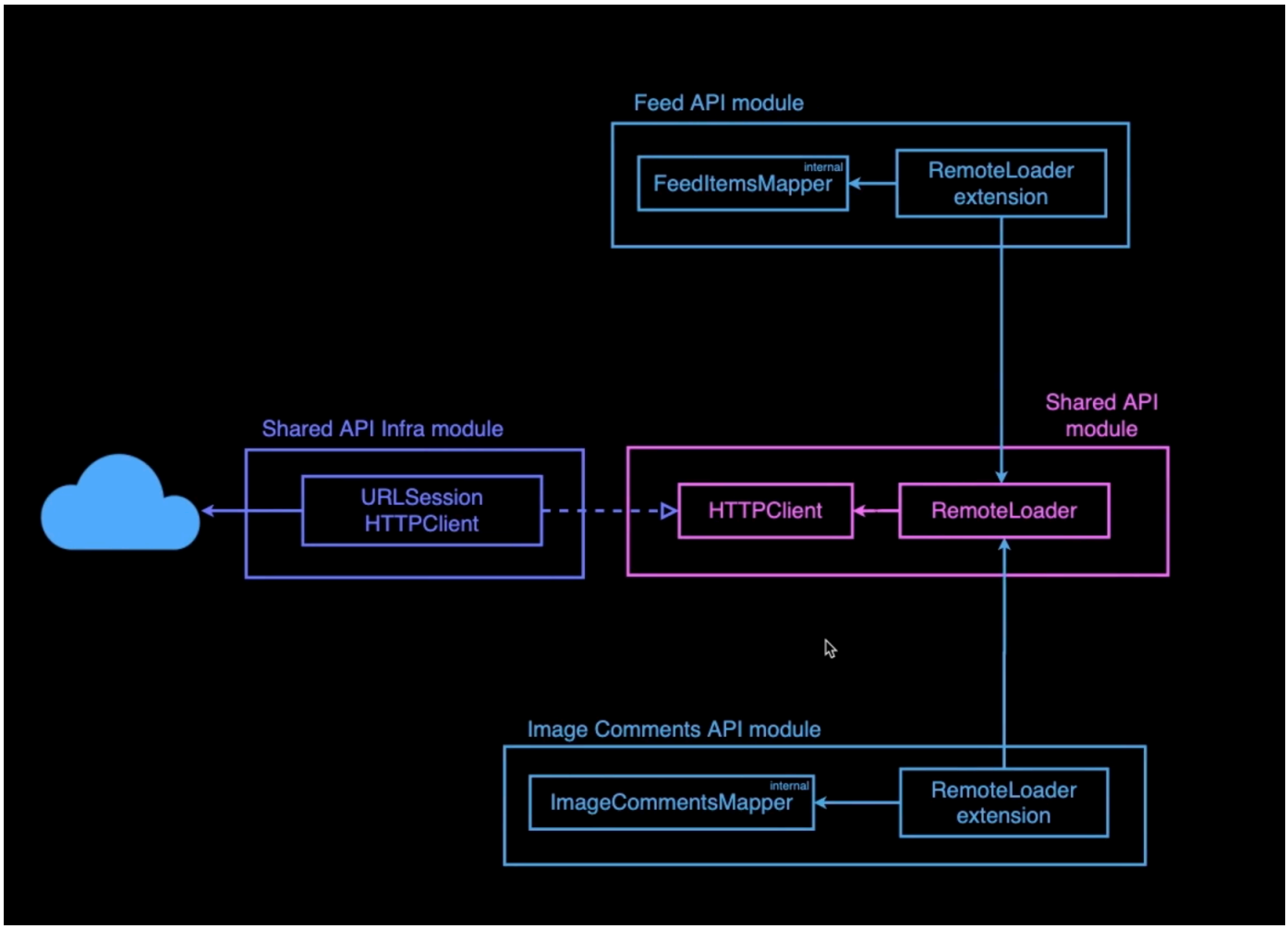
```
public typealias RemoteImageCommentsLoader = RemoteLoader<[ImageComment]>

public extension RemoteImageCommentsLoader {
    convenience init(url: URL, client: HTTPClient) {
        self.init(url: url, client: client, mapper: ImageCommentsMapper.map)
    }
}
```

This way we dont break any existing uses of the `RemoteImageCommentsLoader` and even the tests still work.

In the same fashion we repeat this procedure for the **RemoteFeedLoader**, which takes in a [**FeedImage**] as specific type:

```
public typealias RemoteFeedLoader = RemoteLoader<[FeedImage]>

public extension RemoteFeedLoader {
    convenience init(url: URL, client: HTTPClient) {
        self.init(url: url, client: client, mapper: FeedItemsMapper.map)
    }
}
```

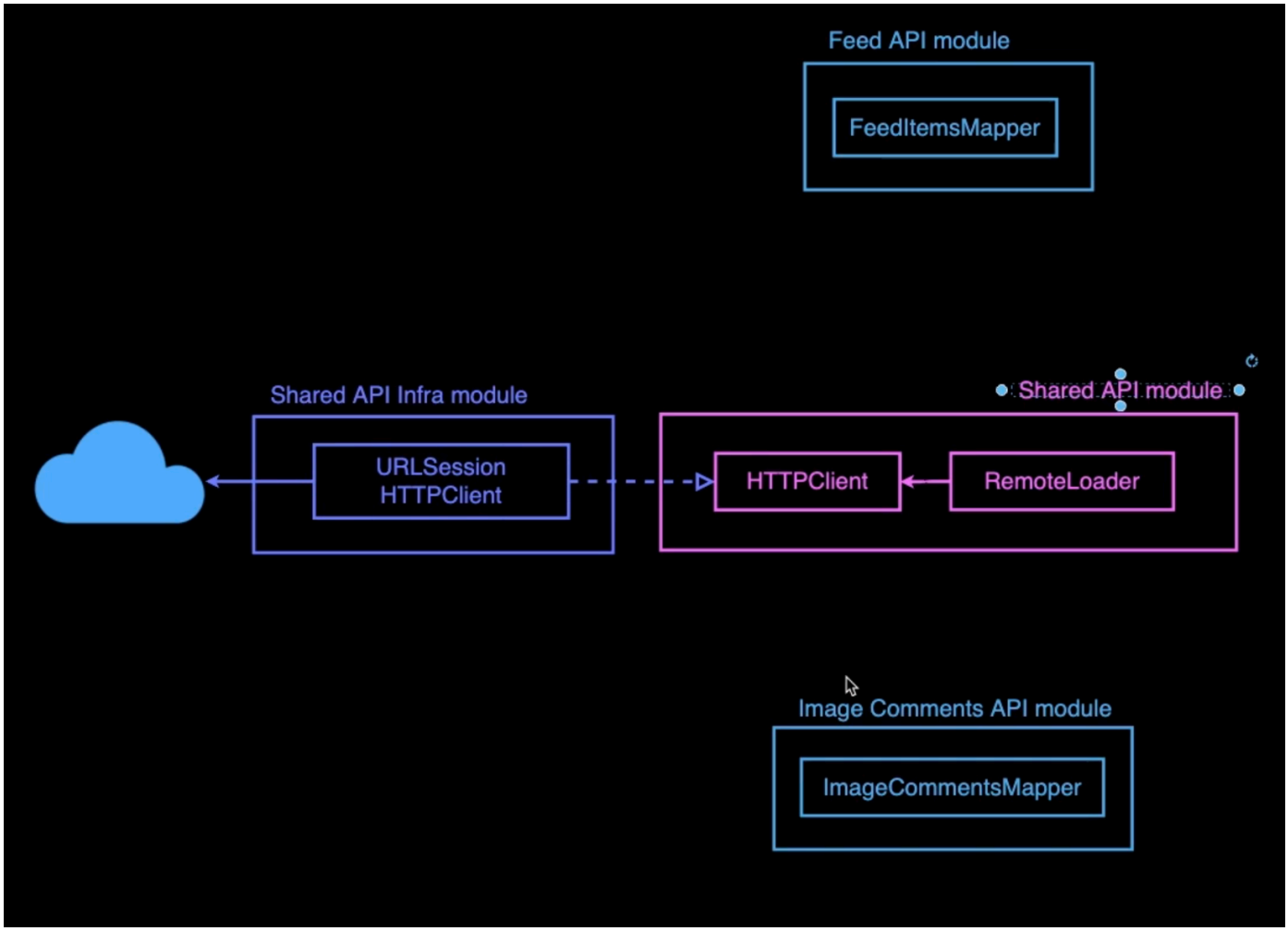
The new diagram is:



The **RemoteLoader** lives in the Shared API module along with the **HTTPClient** interface and both the Feed API and the Image Comments API modules depend on the Shared API module.

They only depend on this module because of the **RemoteLoader** extensions , and these extensions **compose** the generic loader with the mapper, which means that **Composition details** can be moved to the composition root and then we eliminate the dependency on the shared module.

Finally, we can see that the Feed API Module and the Image Comments Module don't depend on any shared infrastructure:



they are all standalone modules.

We eliminated dependency of the API Infrastructure, we are "rejecting" the dependency. (Dependency Injection means that we are injecting the dependencies, the Dependency rejection means that we are changing our system so that we depend on higher abstractions)

Now, we can go one step further, using Combine we can make use of its universal abstractions to further diminish the need for our own. As such we can get rid of the **RemoteLoader** and we can compose the loader with Combine.

```
public extension HTTPClient {
    typealias Publisher = AnyPublisher<(Data, HTTPURLResponse), Error>

    func getPublisher(url: URL) -> Publisher {
        var task: HTTPClientTask?

        return Deferred {
            Future { completion in
                task = self.get(from: url, completion: completion)
```

```
            }
        }
        .handleEvents(receiveCancel: { task?.cancel() })
        .eraseToAnyPublisher()
    }
}
```

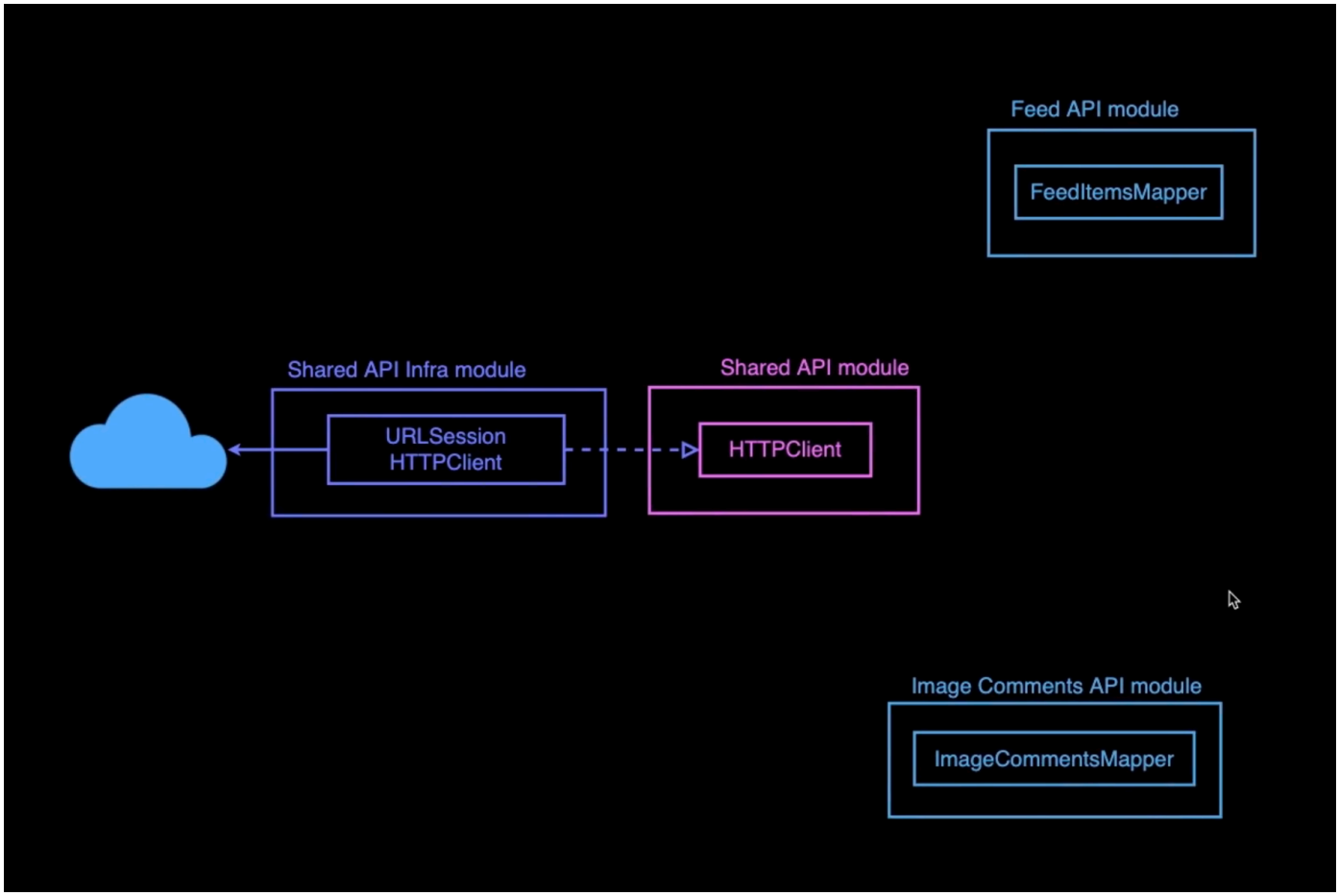
And then we change our `makeRemoteFeedLoaderWithLocalFallback() -> FeedLoader.Publisher` to:

```
  private func makeRemoteFeedLoaderWithLocalFallback() -> FeedLoader.Publisher {
        let remoteURL = URL(string: "http://api-url.com")!

        return httpClient
            .getPublisher(url: remoteURL)
            .tryMap(FeedItemsMapper.map)
            .caching(to: localFeedLoader)
            .fallback(to: localFeedLoader.loadPublisher)
    }
```

This way, we Compose our use-cases with the infrastructure in a **functional** way, we dont ***inject*** dependencies, we ***compose*** them. We create ***composable functions*** and we compose it with the infrastructure, this is the *infamous* functional composition sandwich:

```
[ side-effect ] - httpClient.getPublisher(url: remoteURL)
-pure function  - .tryMap(FeedItemsMapper.map)
[ side-effect]  - .caching(to: localFeedLoader)
```

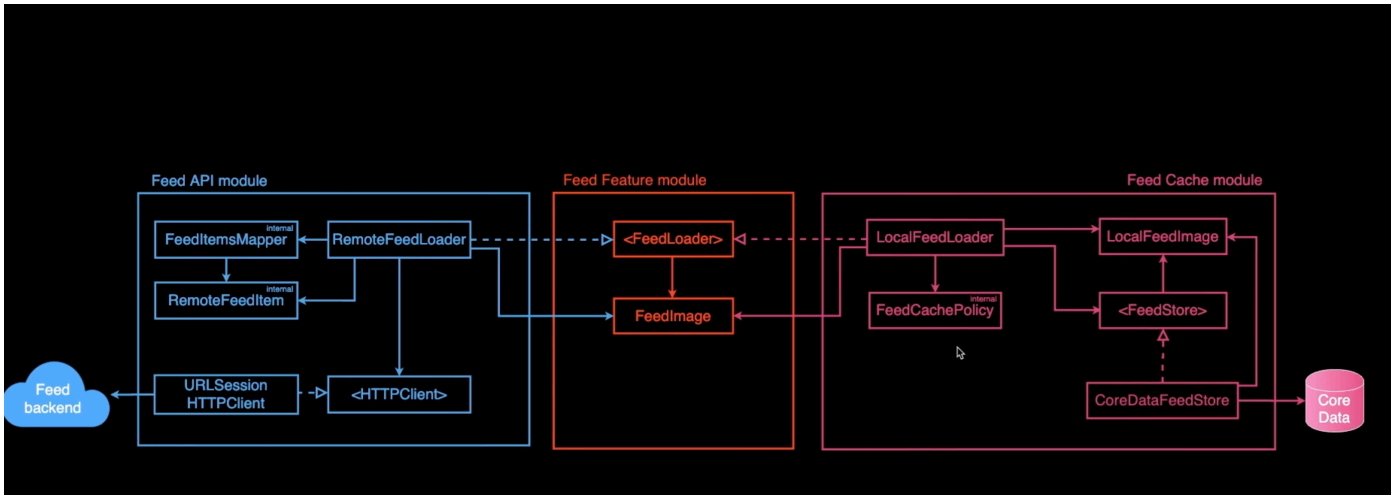
This way, our final design ends up looking like:



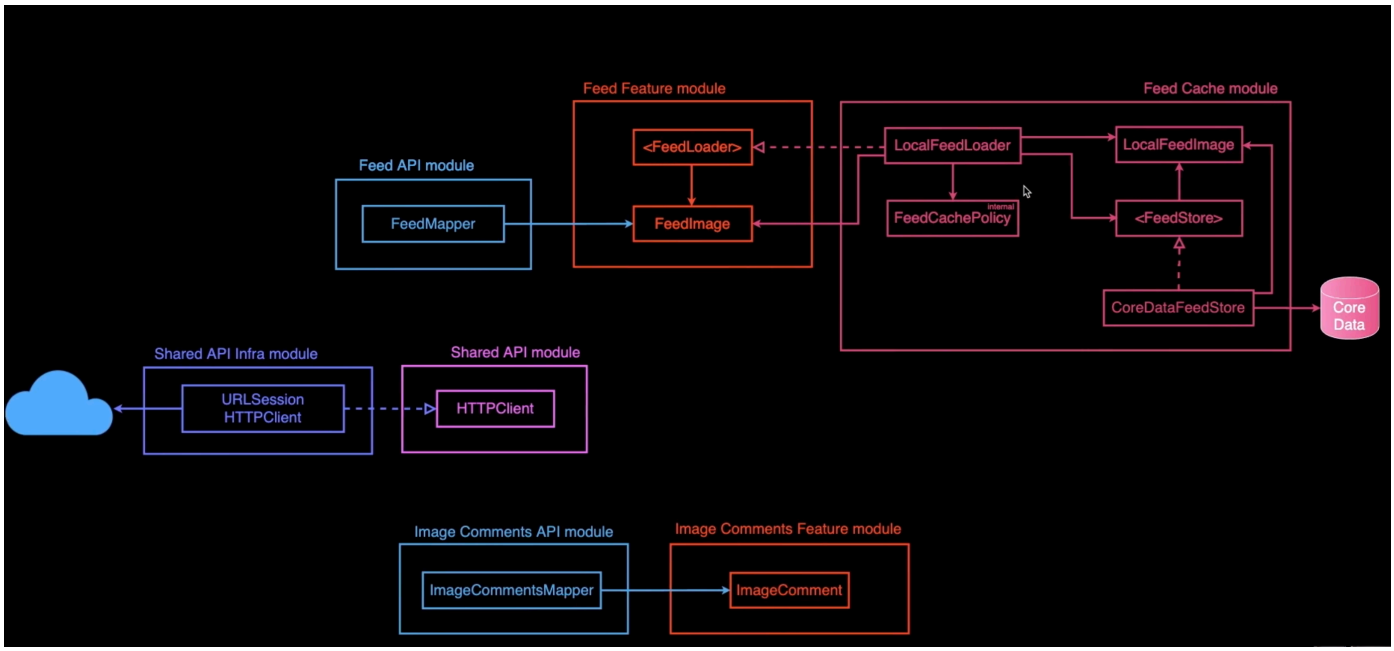
We dont need a loader, everything is composed with abstractions.

Old vs New Design

Old Design



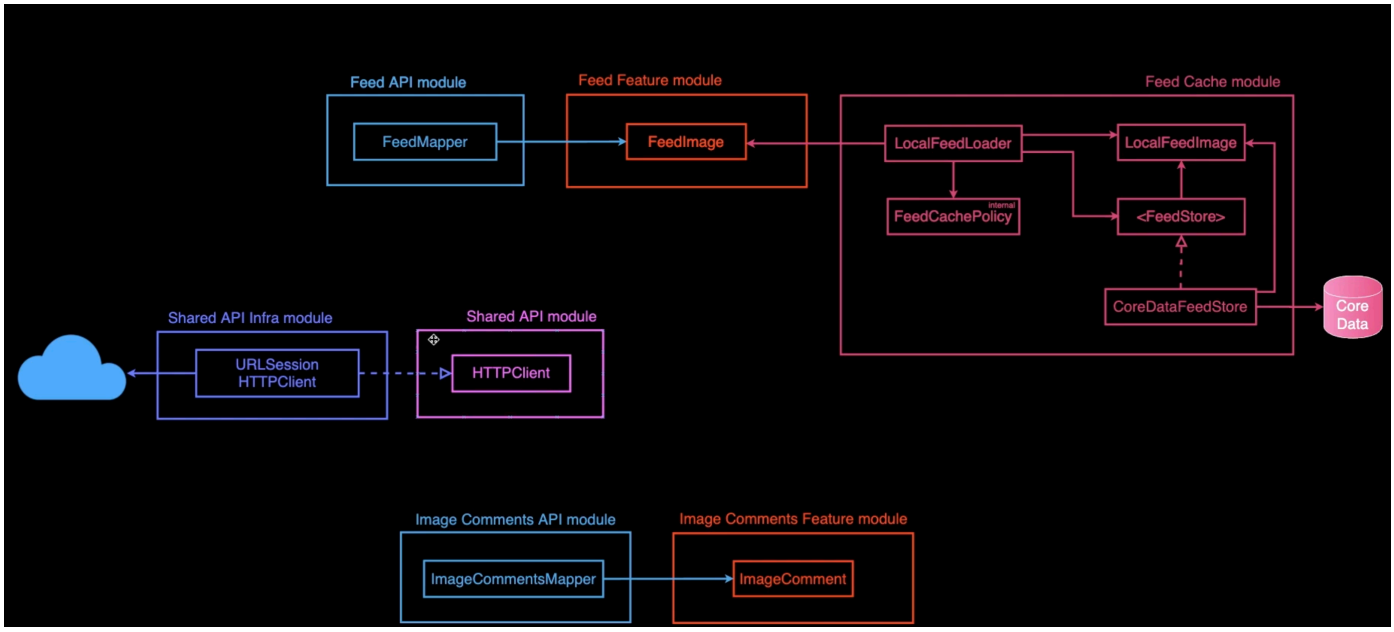
New Design



We dont have a **Loader** anymore, we just have the **FeedItemsMapper**, for the feed-use-case, everything composed in the Composition Root.

Since we dont have a component from the **FeedAPI** module conforming to the **FeedLoader** protocol (like the **RemoteFeedLoader**), we have only the **LocalFeedLoader** conforming to the **FeedLoader** (only one implementation), we don't even need the because we dont need a strategy anymore, so we can remove it.

Removing the `<FeedLoader>` :



We get a much simpler design.

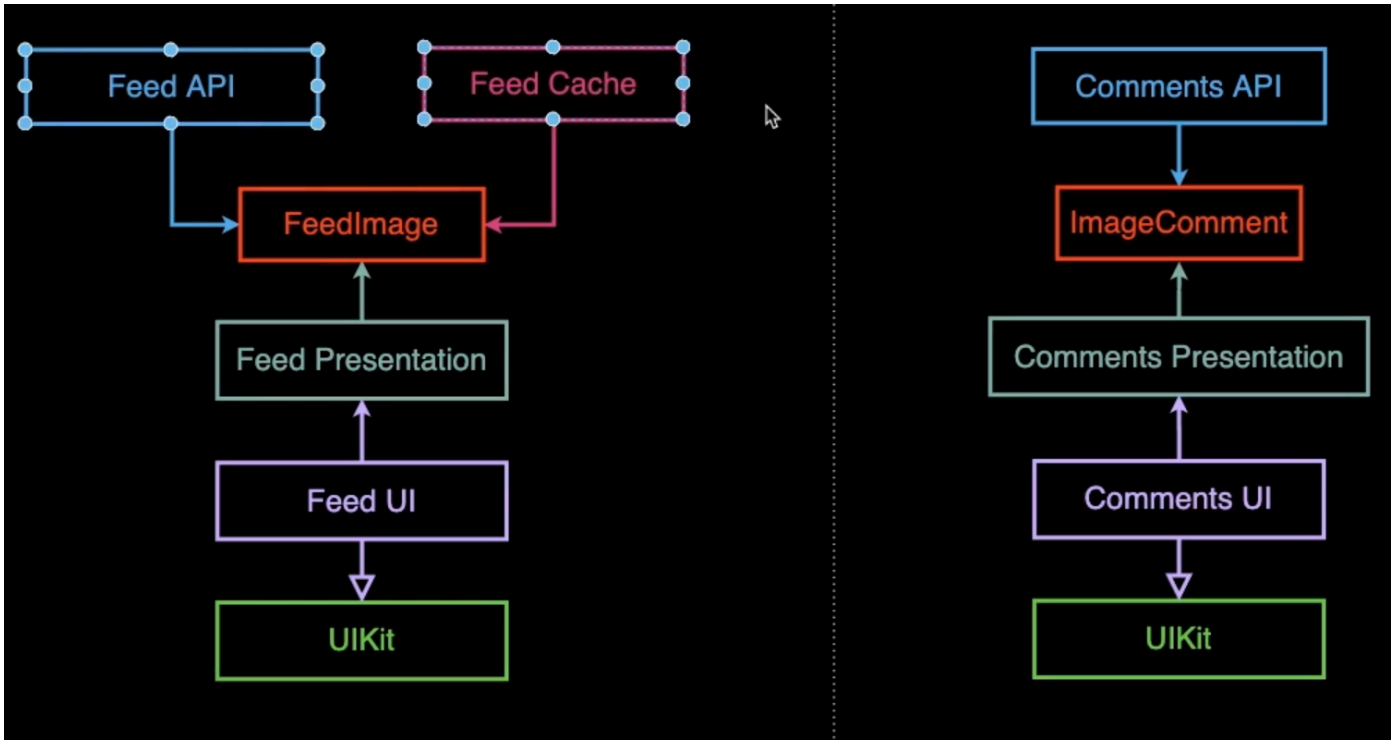
So, **Functional Style**: you dont ***inject*** dependencies, you ***compose*** them with the functional sandwich.

How to create Reusable presentation logic layer

The presentation we have for the main feed is the same that we will have for the comments, we have the spinner, then either the sad case or the happy case. This presentation is even the same for the image loading, although with a different visual representation (spinner vs shimmering).

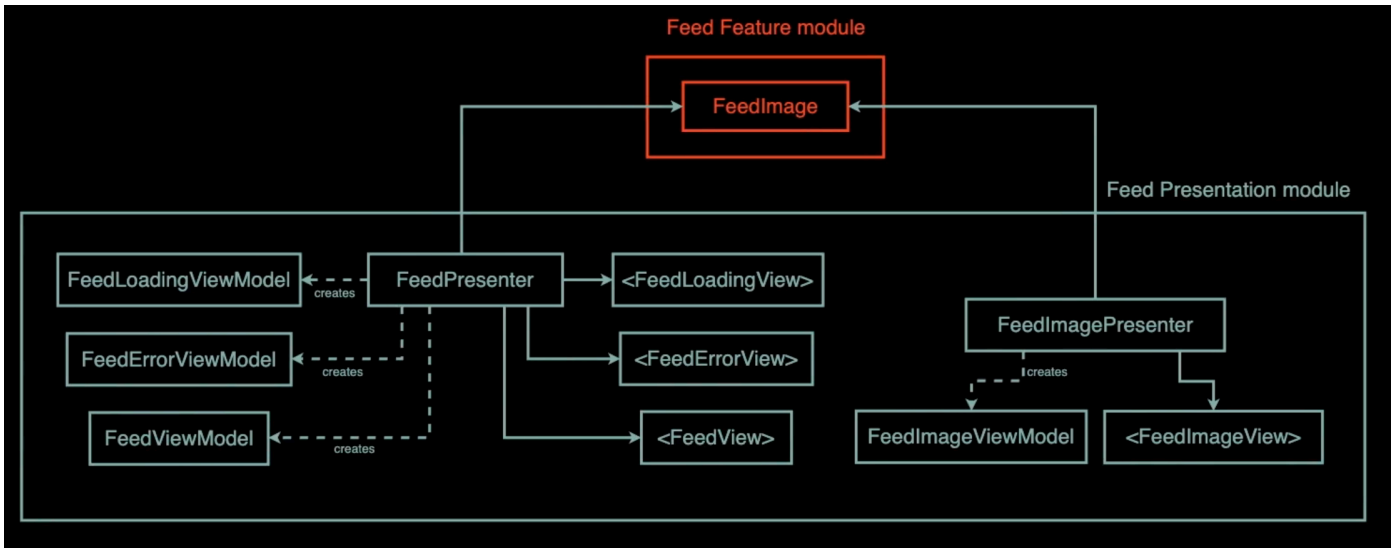
So we have to see how to reuse this code, by making some sort of abstraction, and not duplicate. This way everytime we need to load a resource we can use the esame presentation

Current Design



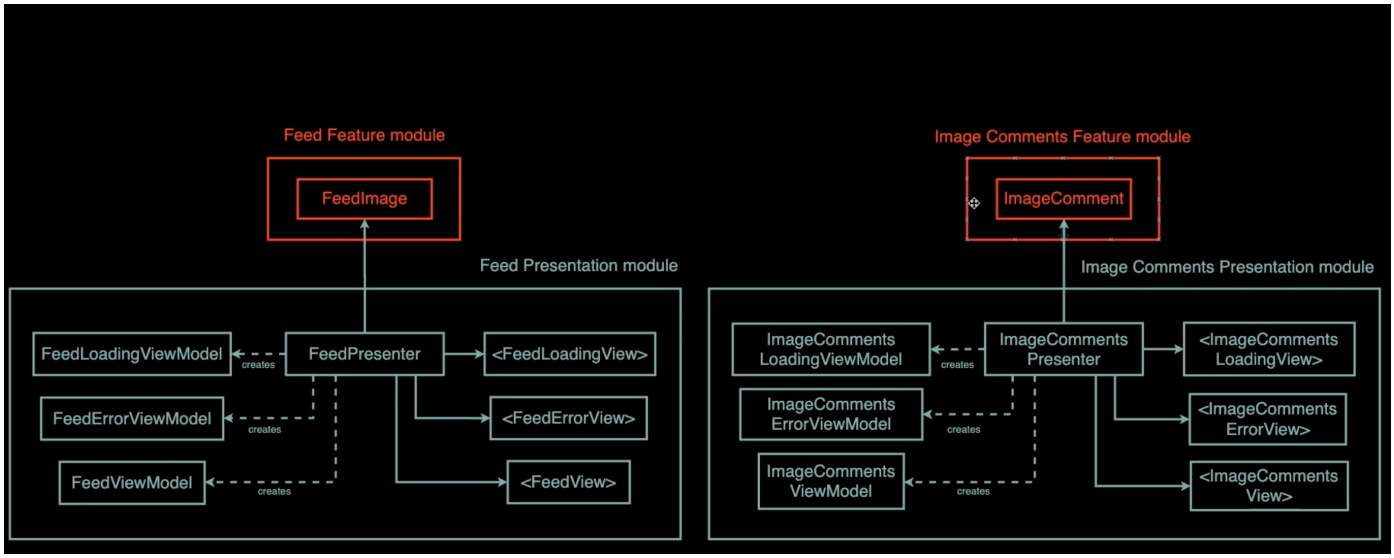
We can see that the features are decoupled from eachother, and we want to keep it that way, even though they are going share presentation logic and may share infrastructure for loading details.

Feed Presentation Logic



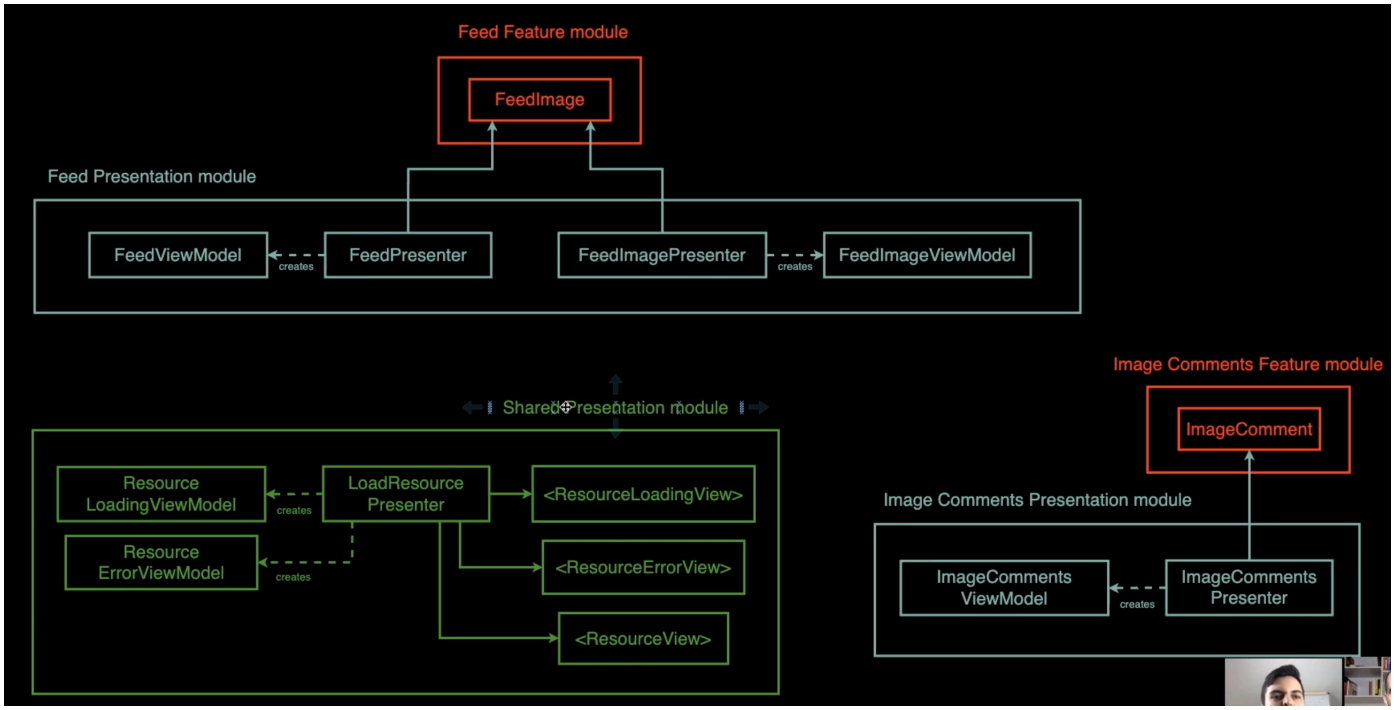
We have the **FeedPresenter** that we implemented in the presentation module, that creates `ViewModels` and passes them to the `<View>` protocols. So the **FeedPresenter** controls the loading of the resource, and the resource is the *feed* which is an array of images.

We can follow the same design for the ***ImageComments*** but we would end up with duplication



But, all the logic of the `LoadingViewModel`, `ErrorViewModel`, `*ViewModel` being passed to the `<View>` protocols is the same for both modules. So we dont't really need duplication.

We can refactor the architecture as follows:



We create a shared module with all that is similar, ***SharedPresentation*** module, and we inject what is different into the generic presenter. So this way we can abstract all the loading of the resources into a reusable share logic and inject what is different (the mappings).

We can see that our modules are decoupled from eachother, how is this possible? By using a Composition Root, this allows us to compose all the modules with whatever abstract dependencies they may have, in a single place, without needing them to be directly coupled by themselves.

Usually its very common to see projects where all the code is *coupled* with a **SharedModule**, with lots of arrows coming into the shared module. But we do not want this because any changes to the SharedModule would mean at minimum needing to recompile and redeploy, and at worse needing to modify the coupled modules. So the best thing is to have them all decoupled, and then use a **Composition Root**.

Generalization of the previously existing Presenter logic

Our **FeedPresenter** is comprised at the moment with the following protocol view dependencies:

```
    private let feedView: FeedView
    private let loadingView: FeedLoadingView
    private let errorView: FeedErrorView
```

and the following methods:

```
 public func didStartLoadingFeed() {
        errorView.display(.noError)
        loadingView.display(FeedLoadingViewModel(isLoading: true))
}
public func didFinishLoadingFeed(with feed: [FeedImage]) {
        feedView.display(FeedViewModel(feed: feed))
        loadingView.display(FeedLoadingViewModel(isLoading: false))
}
public func didFinishLoadingFeed(with error: Error) {
        errorView.display(.error(message: feedLoadError))
        loadingView.display(FeedLoadingViewModel(isLoading: false))
}
```

Doing an analysis we can see that what they do is: `data in → creates view models → data out to the UI` :

`Void` → `creates view models` → `sends to the UI`

```
 public func didStartLoadingFeed() {
        errorView.display(.noError)
        loadingView.display(FeedLoadingViewModel(isLoading: true))
}
```

`[FeedImage]` → `creates view models` → `sends to the UI`

```
public func didFinishLoadingFeed(with feed: [FeedImage]) {
        feedView.display(FeedViewModel(feed: feed))
        loadingView.display(FeedLoadingViewModel(isLoading: false))
}
```

`Error` → `creates view models` → `sends to the UI`

```
public func didFinishLoadingFeed(with error: Error) {
        errorView.display(.error(message: feedLoadError))
        loadingView.display(FeedLoadingViewModel(isLoading: false))
}
```

And, we would need our ImageComments to function in a similar fashion:

`[ImageComment]` → `creates view models` → `sends to the UI`

For images we would need:

`Data` → `creates UIImage` → `sends to the UI`

We come to the conclusion that basically what we need is :

`Resource` → `creates ResourceViewModel` → `sends to the UI`

Thereforethe goal is to create a generic **Presenter** with the above logic.

Procedure

We start by adding a new test file, copying the already existing tests for the already existing **FeedPresenter**, but for our new generic presenter **LoadResourcePresenter**. We analyze which tests make sense for a generic presenter and which dont. We also realize that both the ***resource*** as well as the ***view model*** to be used are to be also generic, for which we will need to inject custom ***mappers*** that map the `resource` → `resourceViewModel` .

(For the tests we need to use a type, since we cant test generics in a generic way, so we choose to use the **String** type, but any type will do, we use **String** because due to its verbosity its easier to read than another type like **Int**.)

Advice: Before you make it generic, make it concrete, otherwise if you try to change behaviour while trying to make it generic at the same time you will have a lot of compiler errors and problems that will confuse you

Once we've made the **ResourcePresenter**, we have to replace the original **FeedPresenter** with the generic one, and we need to inject the mapping.

After that we need to do the same and unify the **FeedImagePresenter** , which will have the same three states, *loading*, *error*, and *success(uiImage)*

We could even make the **FeedLoaderPresentationAdapter** generic since its very similar to the **FeedImageDataLoaderPresentationAdapter**:

1. they tell the presenter they start loading
2. if there is a failure it passes a failure to the presenter
3. if there is a success it passes data to the presenter

The idea is that we end up using the **FeedImagePresenter** in the same fashion as the **FeedPresenter**, only to inject mapping.

To be able to use the generic `LoadResourcePresenter`, we need to have the same cases: *loading, success(data), error(error)* that we did for the **FeedPresenter**,

To do this we will have to refactor the `FeedImageViewModel<Image>` into what is image specific and what is image-data-loading specific.

To do this we start with a test:

```
 func test_map_createsViewModel() {
        let image = uniqueImage()

        let viewModel = FeedImagePresenter<ViewSpy, AnyImage>.map(image)

        XCTAssertEqual(viewModel.description, image.description)
        XCTAssertEqual(viewModel.location, image.location)
}
```

We state that we want a `map(_ image: Image)-> FeedImageViewModel`, that takes in an image and maps it into a **FeedImageViewModel**.

From the original `FeedImageViewModel<Image>`:

```
public struct FeedImageViewModel<Image> {
    public let description: String?
    public let location: String?
    public let image: Image?
    public let isLoading: Bool
    public let shouldRetry: Bool

    public var hasLocation: Bool {
        return location != nil
    }
}
```

The only properties that will remain in the **FeedImageViewModel** will be `description` and `location` which are the true properties related to the FeedImage, the rest are either related to loading or to error, and will be moved to other ViewModels.

First of all we need to replace the **FeedImageDataLoaderPresentationAdapter** from the **FeedViewAdapter** with the generic **LoadResourcePresentationAdapter**.

We have to go from:

```
let adapter =
FeedImageDataLoaderPresentationAdapter<WeakRefVirtualProxy<FeedImageCellController>,
UIImage>(model: model, imageLoader: imageLoader)
```

to:

```
let adapter = LoadResourcePresentationAdapter<Data,
WeakRefVirtualProxy<FeedImageCellController>>(loader: () -> AnyPublisher<Resource,
Error>)
```

But as we can see, we have unmatching types, since the **FeedImageDataLoaderPresentationAdapter** takes in a model (which it uses to get the URL for the closure) and an `imageLoader: (URL) -> AnyPublisher<(Data, HTTPURLResponse), Error>`, but the **LoaderResourcePresentationAdapter** only takes in a `loader: () -> AnyPublisher<Resource, Error>` closure.

So what we can do in this situation is pass a custom closure that calls the `imageLoader(model.url)`, and thus we are adapting the `imageLoader` method that takes in a parameter into a closure that takes in no parameters. This is called ***partial application of functions*** and it allows us to adapt closures:

By doing this, we are able to adapt from a closure that takes in a parameter, like `imageLoader(url)`, by pre-passing the required parameter, and thus we wont need to pass the model, therefore having a signature that looks alike the previous adapter one.

This way the client/caller doesnt need to know about the model parameter, it stays in the composition. In essence what we have here is another Adapter, adapting inputs and outputs to our convenience.

The other thing that we see when analyzing the existing code, is that the existing **FeedImageDataLoaderPresentationAdapter** passes the model around through the presenter, so the presenter can send it to the UI, but we don't need to do that, we can pass the ViewModel directly to the **FeedImageCellController**, because the ViewModel is immutable, this is because we already have access to the **model**, which is immutable, thus the **FeedImageViewModel** will be immutable (because both the *description* and *location* don't change, they are `let`).

Therefore we can pass in the **ViewModel** to the **FeedImageCellController** at construction time. **(Things that dont change, can be passed in initialization time, and things that change over time you pass either through property injection or method injection)**

To do this we only need to use the `map` function from the `FeedImagePresenter` that we created earlier and do:

```
let view = FeedImageCellController(viewModel: FeedImagePresenter.map(model), delegate:
adapter)
```

This change allows us to modify the **FeedImageCellController**. Now we will keep a copy **FeedImageViewModel** that will be immutable, except for its UIImage which will came asynchronously from the network. This means we can move some code from the `func display(_ viewModel: FeedImageViewModel)` into the `func view(in tableView:)`, method because we will have that information at that time.

Next step is to modify the **FeedImageCellController** so that it conforms to the , and protocol, which means that we get the new `display(_ viewmodel: Resource)` , `display(_ viewModel: ResourceLoadingViewModel)` and `display(_ viewModel: ResourceErrorViewModel)` (where Resource is passed in as UIImage for this case).

This way, we get rid of our original `func display(_ viewModel: FeedImageViewModel)` and its code gets distributed into the three different viewModel states of loading, success and error.

Finally, we remove all the logic related to the **FeedImagePresenter** , in lieu of the new generic presenter.

Next step is to finally implement the **ImageCommentsPresenter**, which in the same fashion as before, will have a `map(_ model: [ImageComment]) -> ImageCommentsViewModel` method.
ImageCommentsViewModel is just a wrapper for an array of **ImageCommentViewModel**.

```
public struct ImageCommentsViewModel {
    public let comments: [ImageCommentViewModel]
}

public struct ImageCommentViewModel {
    public let message: String
    public let date: String
    public let username: String
}
```

our map function is:

```
  public static func map(_ comments: [ImageComment]) -> ImageCommentsViewModel {
        ImageCommentsViewModel(comments: comments.map { comment in
            let formatter = RelativeDateTimeFormatter()
            let localizedDate = formatter.localizedString(for: comment.createdAt,
relativeTo: Date())

            return ImageCommentViewModel(message: comment.message, date: localizedDate,
username: comment.username)
        })
    }
```

But there is a problem with this map function, which is that it is not taking into account any of the locale data for the date, which makes our tests brittle.

So we modify our map function to take into account the current date, locale, and calendar parameters to properly create the viewModel.

```
public static func map(_ comments: [ImageComment],
                           currentDate: Date = Date(),
                           calendar: Calendar = .current,
                           locale: Locale = .current) -> ImageCommentsViewModel {

        ImageCommentsViewModel(comments: comments.map { comment in
            let formatter = RelativeDateTimeFormatter()
            formatter.calendar = calendar
            formatter.locale = locale

            let localizedDate = formatter.localizedString(for: comment.createdAt,
relativeTo: currentDate)

            return ImageCommentViewModel(message: comment.message, date: localizedDate,
username: comment.username)
        })
}
```

This way we can properly test different locales and what date string results they return.

This is all that that was required for the **ImageCommentsPresenter** layer. (literally a map and a title, the rest is reused.)

We now have a **very clear way** to add new resource presenters, and adding new API mappers without duplicating code nor coupling modules:

- Everytime there is a new feature, we just create a new API mapper that maps from **domain model** to **view model**, and then we compose with the composable logic in the UICompositionRoot.

We are now ready to implement the UILayer for the comments section. As an additional note, it would be possible to create an adapter between the presenter and the view layer so that both could be changed without incurring in trouble for the dependant layer , but for this case it's not neccesary.

Truth is, that we dont even need the Specific Presenter, because at this point it's just a namespace for the map function, and we can inject directly the init for the viewmodels instead of the map functions.

Image Comments UI Layer

In the previous sections we implemented:

- [x] API (Live 001)
- [x] Presentation (Live 002)

Now we will implement:

- [ ] UI (Live 003)
    - [ ] Reusing UI components (without breaking modularity)
    - [ ] Creating UI elements programatically
    - [ ] Diffable Data Sources
    - [ ] Dynamic Fonts (aka Dynamic Type)
    - [ ] Snapshot testing

In the following Live we will implement:

- [ ] Composition (Live 004)

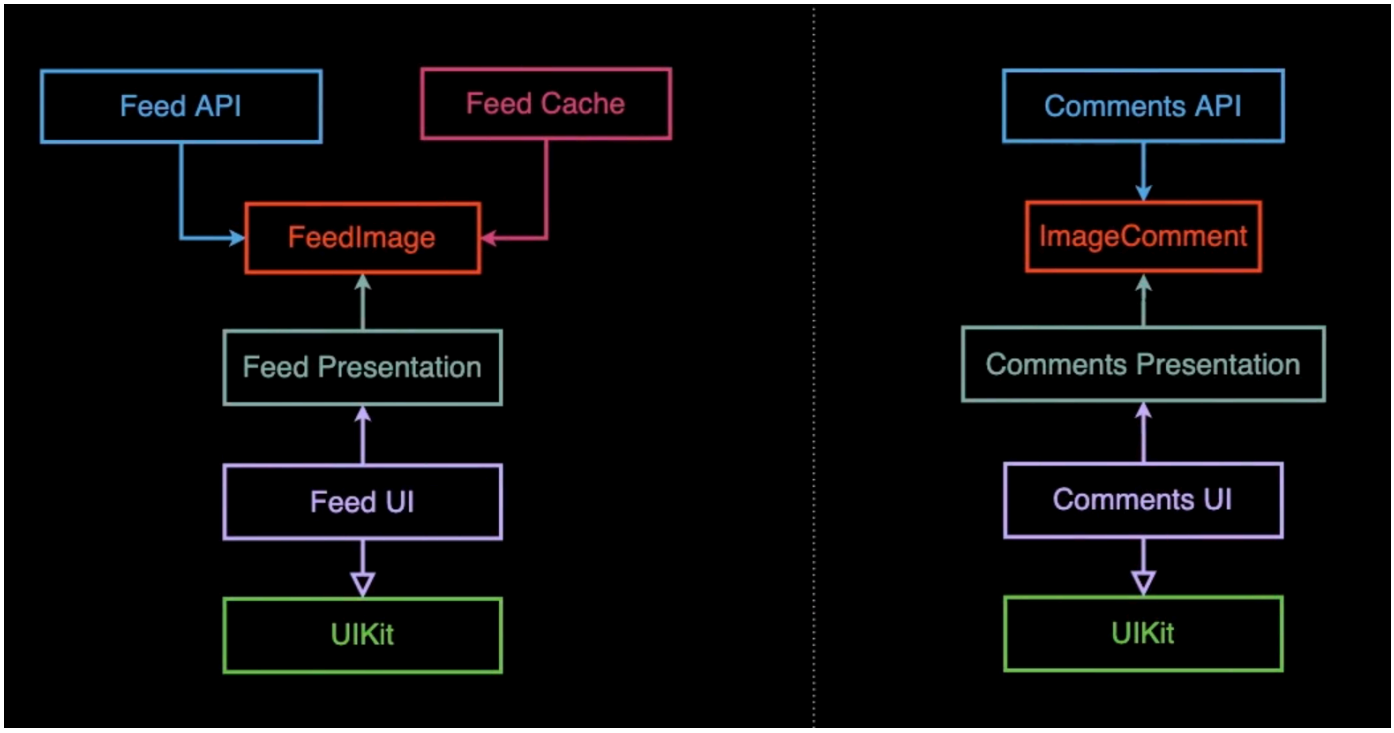
First Analysis

At first sight we can say that views are generally the same, the only thing that is going to change for our UI is the cell configuration, and the loading/error view is exactly the same for both the feed and the comments section.

So we are going to reuse as much as we can the same UI from our existing feed, this way we avoid code duplication and we guarantee a smooth user experience.

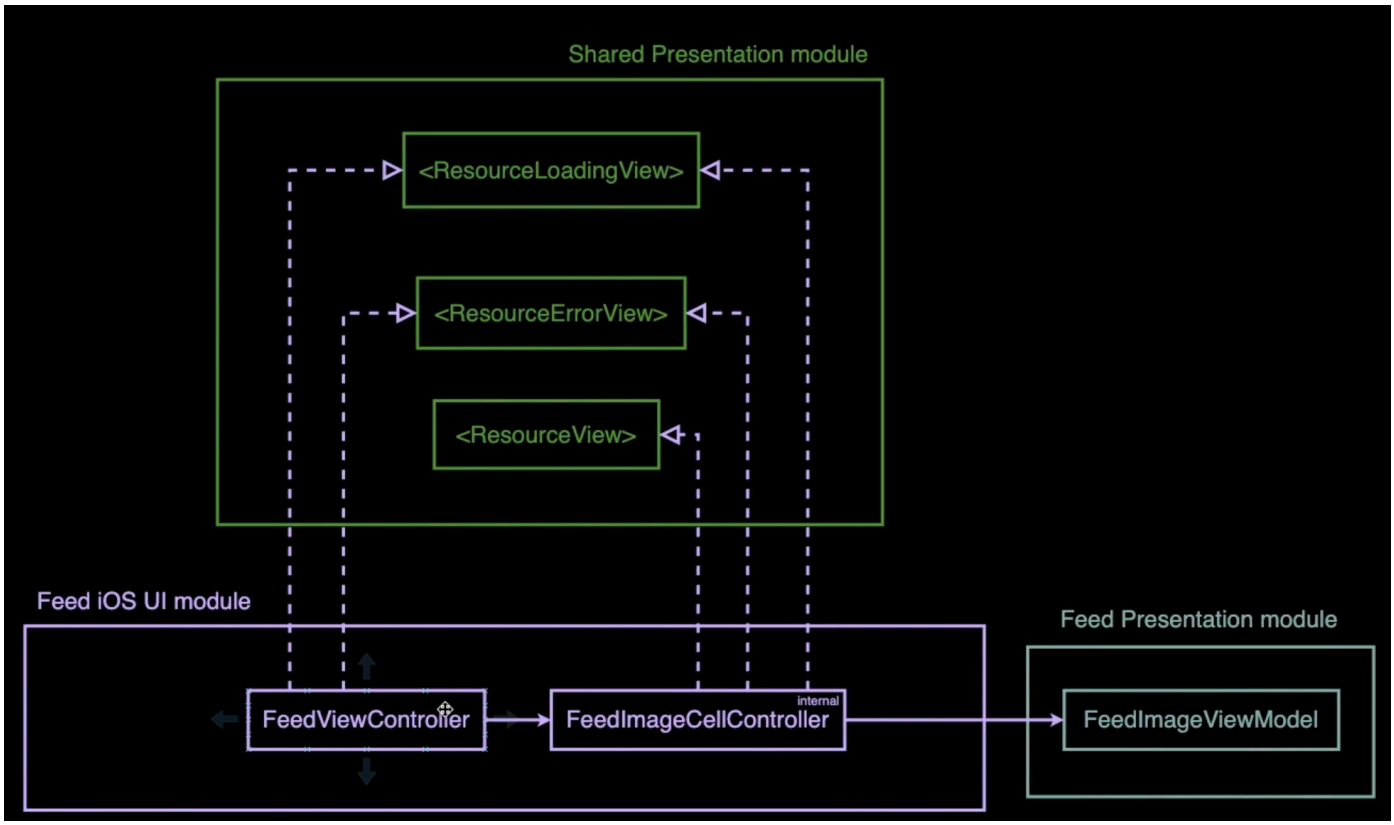
We are going to genericize our existing ui and just inject the different changes when needed, which in this case is the cell.

As usual, we start by having a look at our diagram:



We have already implemented the **Feed UI** , now its time to implement the **Comments UI**

Lets take a look at our **Feed User Interface Design**:



So, we have a **FeedViewController** that implements some shared Presentation Interfaces/ protocols and it renders a collection of **FeedImageCellControllers**, and each **FeedImageCellController** renders a Cell configured for a specific **FeedImageViewModel**, so every CellController is specific to one cell, and the **FeedViewController** coordinates the collection of **CellControllers** , we can see this here:

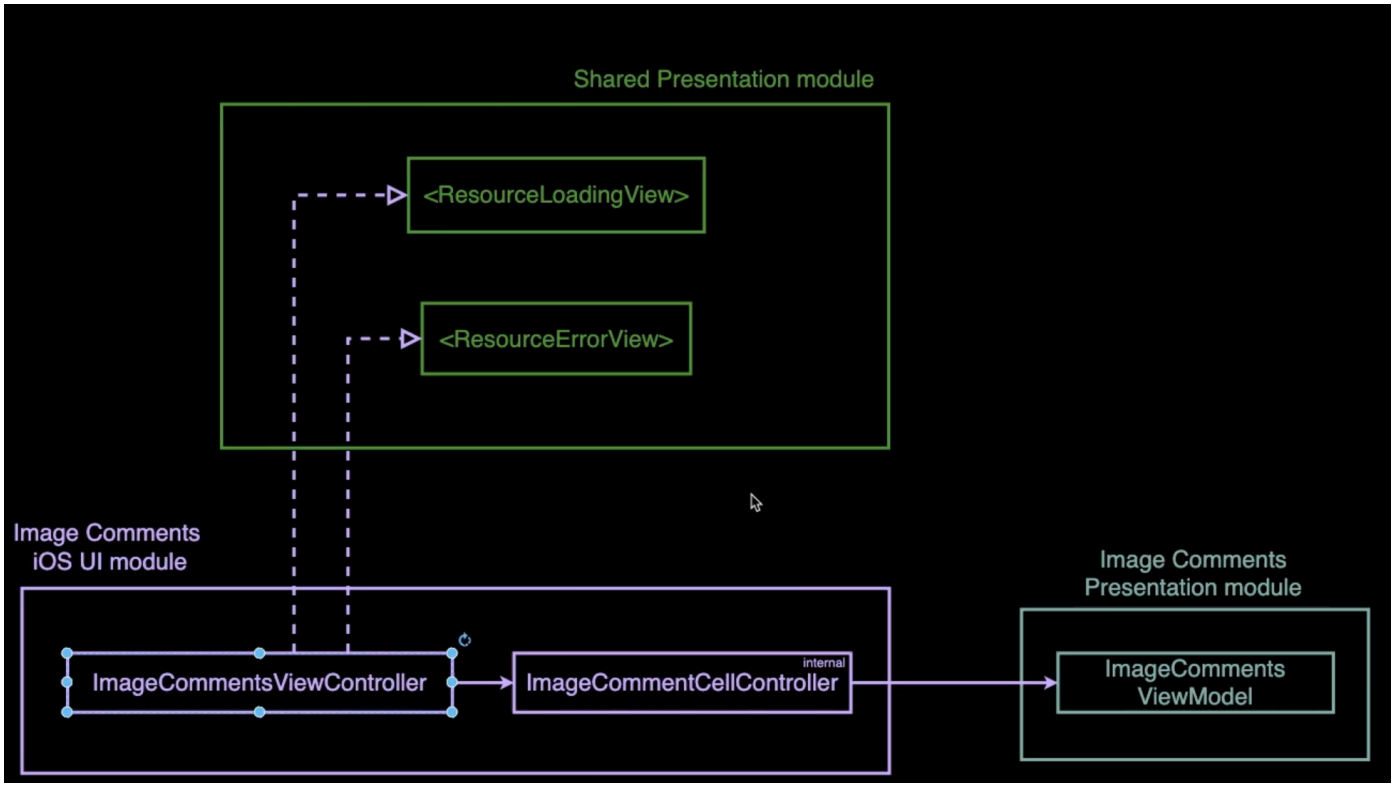
```
public final class FeedViewController: UITableViewController,
UITableViewDataSourcePrefetching, ResourceLoadingView, ResourceErrorView {
    @IBOutlet private(set) public var errorView: ErrorView?

    private var loadingControllers = [IndexPath: FeedImageCellController]()

    private var tableModel = [FeedImageCellController]() {
        didSet { tableView.reloadData() }
    }
    .....
    .....
}
```

These **FeedImageCellController**s are a bunch of tiny MVC's controlling specific parts of the UI, in this case, the Cells:

```
public final class FeedImageCellController: ResourceView, ResourceLoadingView,
ResourceErrorView {

    private let viewModel: FeedImageViewModel
    private let delegate: FeedImageCellControllerDelegate
    private var cell: FeedImageCell?

    public init(viewModel: FeedImageViewModel, delegate:
FeedImageCellControllerDelegate) {
        self.viewModel = viewModel
        self.delegate = delegate
    }
  ...
  ...
}
```

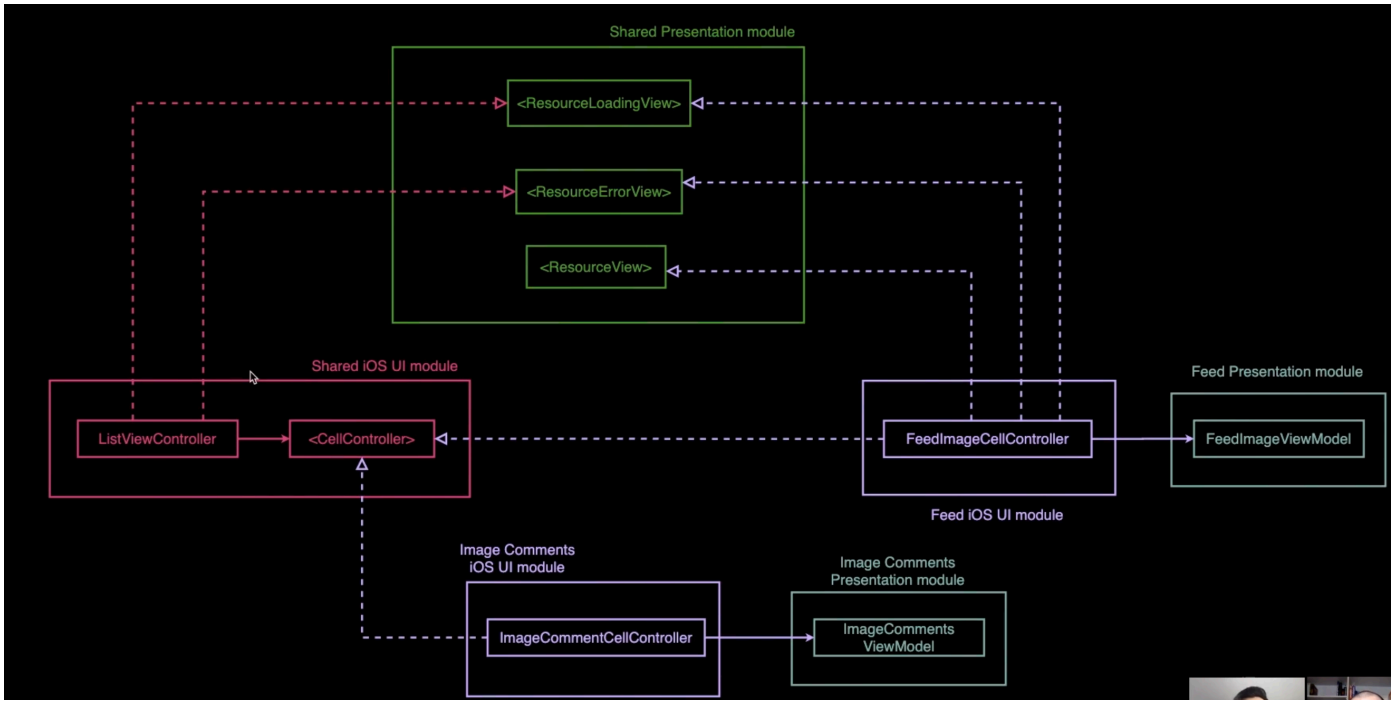
Where the ViewModel is the model, the Cell is the View and then the Controller.

So, we could follow the same design and create an **ImageCommentsViewController** that coordinates a collection of **ImageCommentCellControllers** that renders **ImageCommentsViewModel**.



But what is the **problem** with this approach? ➡️ ➡️ **Duplication** . We would have complete duplication, which is not what we want. What we want to do is to make the **FeedImageCellController** that we already have generic so that it can display whatever we want by injecting different types of cell controllers.

What we can do is create a **Shared UI Module** with shared logic. We could create a **ListViewController** that can render any type of `<CellController>` abstraction, instead of having concrete-typed **FeedViewController** and **ImageCommentsViewController**:



And both the **FeedImageCellController** and **ImageCommentCellController** would implement this abstraction: `<CellControllers>` which doesn't know about concrete, feature specific **CellControllers**, which means we can add new features and reuse the **ListViewController**.

First of all we create the protocol by extracting the method names that we need to abstract:

```
public protocol CellController {
    func view(in tableview: UITableView) -> UITableViewCell
    func preload()
    func cancelLoad()
}
```

Then we replace all the instances of **FeedImageCellController** with . Now, any concrete class that implements CellController can be used to render Cells in the FeedViewController.

So we start by making **FeedImageCellController** conform to our new abstraction :

```
public final class FeedImageCellController: CellController, ResourceView,
ResourceLoadingView, ResourceErrorView {}
```

Next step is to refactor the name of the **FeedViewController** into something more generic like **ListViewController**:

```
public final class FeedViewController: UITableViewController,
UITableViewDataSourcePrefetching, ResourceLoadingView, ResourceErrorView {}
```

→→→→→

```
public final class ListViewController: UITableViewController,
UITableViewDataSourcePrefetching, ResourceLoadingView, ResourceErrorView {}
```

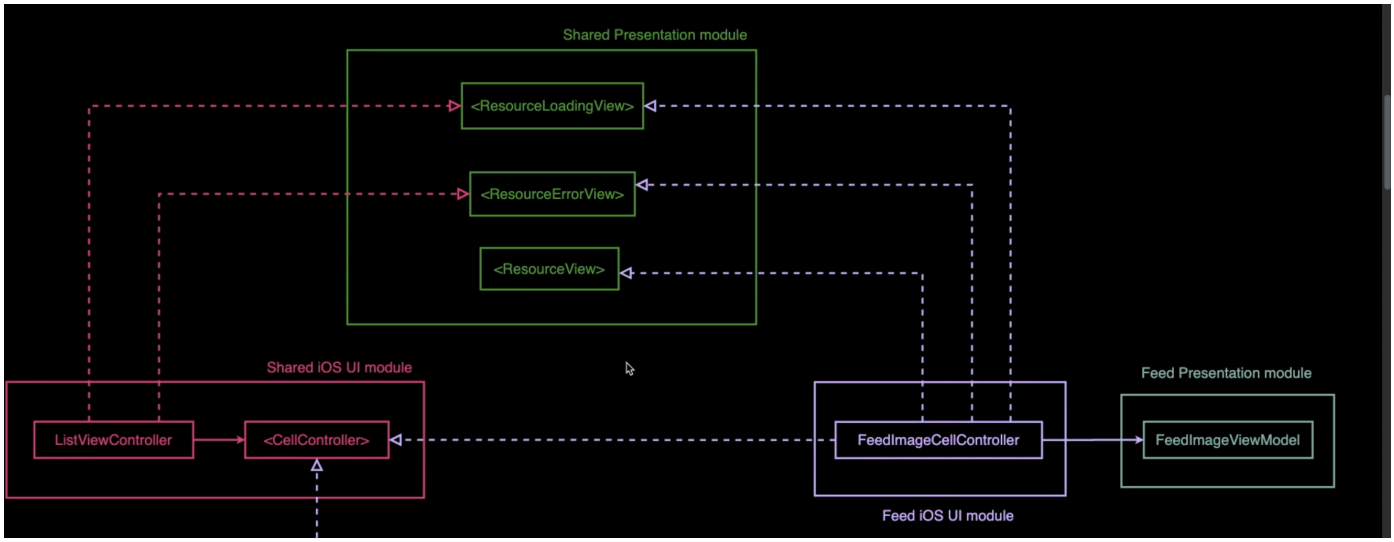
Following, we get rid of the , and we replace it with a closure, since this protocol only had one method. So we replace the ***delegate*** property with an ***onRefresh*** closure. We assign this closure in the **CompositionRoot**, in the **FeedUIComposer**.

We move the extension logic from the **LoadResourcePresentationAdapter** that conformed to the delegate, to the **FeedUIComposer** and simply assign:

```
feedController.onRefresh = presentationAdapter.loadResource
```

There is nothing wrong with using delegates/protocols, but usually protocols with one single method can be replaced by closures.

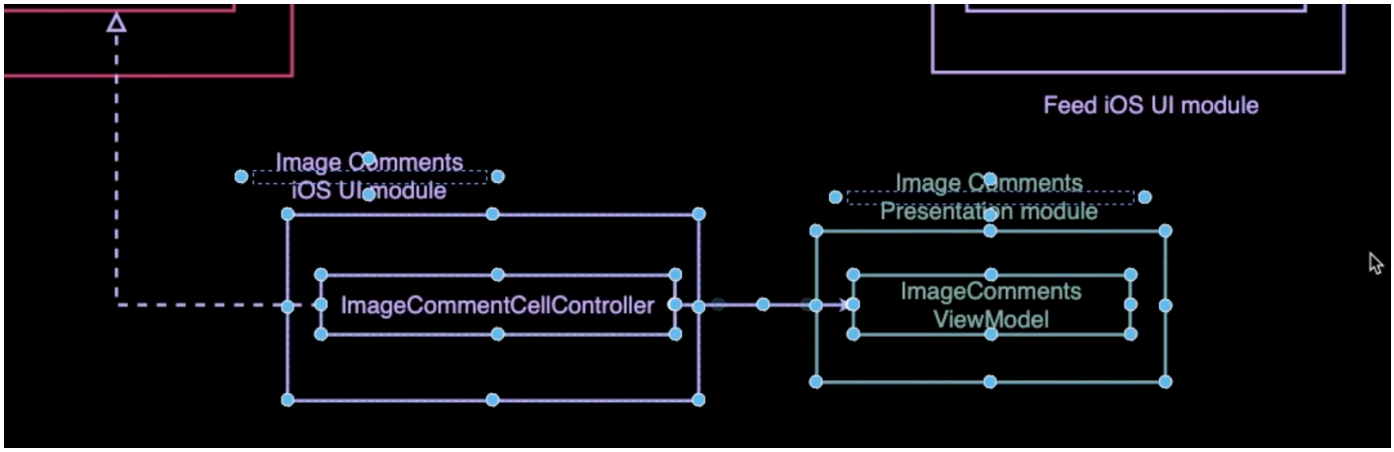
At this point we are already done with the first part:



Because all we did was rename classes and conform to the new `<CellController>` protocol.

ImageComments UI

Now its time to implement the ImageComments UI



So, as always, we start with a test, in this case with snapshot tests, where we create the required dummy data to display, we start to build our **ImageCommentCellController**

Analyzing the methods declared by the `<CellController>` protocol :

```
public protocol CellController {
    func view(in tableView: UITableView) -> UITableViewCell
    func preload()
    func cancelLoad()
}
```

Its possible to arrive to the conclusion that they look pretty similar to that of the methods established by the protocols: `<UITableViewDelegate>`, `<UITableViewDatasource>` and `<UITableViewDatasourcePrefetching>` :

```
//UITableViewDatasource
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
//Among others...

//UITableViewDatasourcePrefetching
func tableView(_ tableView: UITableView, prefetchRowsAt indexPaths: [IndexPath])
func tableView(_ tableView: UITableView, cancelPrefetchingForRowsAt indexPaths:
[IndexPath])
//Among others...

//UITableViewDelegate
func tableView(_ tableView: UITableView, didEndDisplaying cell: UITableViewCell,
forRowAt indexPath: IndexPath)
//Among others...
```

So by this logic, we will remove the protocols we gave to the `<CellController>`, and make it a typealias conforming to these other protocols.

```
public typealias CellController = UITableViewDataSource & UITableViewDelegate &
UITableViewDataSourcePrefetching
```

This way, we get the functionality we want, in a way that is harmonic with the pre-existing protocols. This way we can easily make our specific **FeedImageCellController** and **ImageCommentCellController** work in harmony with the generic **ListViewController**, by simply forwarding the generic tableView events happening in the generic **ListViewController** to the appropriate implementation, as needed.

Doing this allows us to further decouple other modules from the Shared UI Module.

But, since there are many clients that implement that do not need/aren't interested in implementing all the methods that conforming to multiple protocols entails, we replace protocol composition with a struct composition:

```
public struct CellController {
    let dataSource: UITableViewDataSource
    let delegate: UITableViewDelegate?
    let dataSourcePrefetching: UITableViewDataSourcePrefetching?

    public init(_ dataSource: UITableViewDataSource & UITableViewDelegate &
UITableViewDataSourcePrefetching) {
        self.dataSource = dataSource
        self.delegate = dataSource
        self.dataSourcePrefetching = dataSource
    }

    public init(_ dataSource: UITableViewDataSource) {
        self.dataSource = dataSource
        self.delegate = nil
        self.dataSourcePrefetching = nil
    }
}
```

With this struct composition, we stablish which protocol *must* be implemented by the clients: the `<UITableViewDataSource>` protocol, since without it it wouldn't be possible to display anything in the tableview, therefore it is mandatory. But both the delegate and dataSourcePrefetching protocols might not be something that clients need to implement, so this way when they do not need them, they simply don't.

A think to remark is the cunning double initializer that automatically detects whether the passed `dataSource` parameter is conforming to all the three protocols or only to the `UITableViewDataSource` , using runtime polymorphism. This way it can automatically choose the appropriate initializer and, if necessary, set the delegate and dataSourcePrefetching properties to nil.

This way, for example, our **ImageCommentCellController** client, that was previously conforming to the previous protocol composition, will now not need to implement the:

```
public func tableView(_ tableView: UITableView, prefetchRowsAt indexPaths: [IndexPath])
{}
```

method, and can simply conform to the `<UITableViewDataSource>` protocol, and be composed/wrapped in other places of the codebase by using the **CellController** client that takes in an object conforming to the , like this.

With this change, we compose our **ListViewController** conformance to the UITableViewDatasource, UITableViewDelegate and UITableViewDataSourcePrefetching protocols in the following way:

First we show the `<UITableViewDataSource>` protocol conformance and forwarding:

```
//Before
public override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
      let controller = cellController(forRowAt: indexPath)
      return controller.tableView(tableView, cellForRowAt: indexPath)
}
//After
public override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let datasource = cellController(forRowAt: indexPath).dataSource
    return datasource.tableView(tableView, cellForRowAt: indexPath)
}
```

We can see how we forward the generic `cellForRowAt(_:)` method from our **ListViewController** into the appropriate **CellController** clients (according to the table position) that implemented the desired **datasource** (which is always necessary for all clients, for it to be possible to display information on the tableview.)

Following, we see the conformance to the `<UITableViewDataSourcePrefetching>` protocol, which is not implemented by all the **CellController** clients , for example **ImageCommentCellController** does not implement this protocol, for which case the forwarding wouldn't execute, since the client's **datasourcePrefetching** property would be `nil`.

```
//Before
public func tableView(_ tableView: UITableView, prefetchRowsAt indexPaths: [IndexPath])
{
    indexPaths.forEach { indexPath in
        let controller = cellController(forRowAt: indexPath)
        controller.tableView(tableView, prefetchRowsAt: [indexPath])
    }
}
//After
public func tableView(_ tableView: UITableView, prefetchRowsAt indexPaths: [IndexPath])
{
    indexPaths.forEach { indexPath in
        let datasourcePrefetching = cellController(forRowAt:
indexPath).dataSourcePrefetching
        datasourcePrefetching?.tableView(tableView, prefetchRowsAt: [indexPath])
    }
}
```

```
//Before
public func tableView(_ tableView: UITableView, cancelPrefetchingForRowsAt indexPaths:
[IndexPath]) {
    indexPaths.forEach { indexPath in
        let datasourcePrefetching = removeLoadingController(forRowAt:
indexPath)?.dataSourcePrefetching
        datasourcePrefetching?.tableView?(tableView, cancelPrefetchingForRowsAt:
[indexPath])
    }
}
//After
public func tableView(_ tableView: UITableView, cancelPrefetchingForRowsAt indexPaths:
[IndexPath]) {
    indexPaths.forEach { indexPath in
        let datasourcePrefetching = cellController(forRowAt:
indexPath).dataSourcePrefetching
        datasourcePrefetching?.tableView?(tableView, cancelPrefetchingForRowsAt:
[indexPath])
    }
}
```

We can see how in the previous case, *every* **CellController** had to be able to forward the events because they all depended on the `<UITableViewDatasourcePrefetching>` protocol, but now, if the client has its `dataSourcePrefetching` property set to nil, no forwarding will be executed.

Finally we see the conformance to the `<UITableViewDelegate>` protocol, which is the protocol in charge of notifying about events that happened on the UI so that something can be done when they do. One such methods is the `didSelectRowAt(_:)`, which notifies when a tableView row has been tapped and allows for logic to be run on such cases. At this point we are not making use of this method, but we are making use of the `didEndDisplaying(_:)` method, that notifies the clients when the cells are out of view bounds so that some action can be taken:

```
//Before
public override func tableView(_ tableView: UITableView, didEndDisplaying cell:
UITableViewCell, forRowAt indexPath: IndexPath) {
    let controller = removeLoadingController(forRowAt: indexPath)
    controller?.tableView?(tableView, didEndDisplaying: cell, forRowAt: indexPath)
}
//After
public override func tableView(_ tableView: UITableView, didEndDisplaying cell:
UITableViewCell, forRowAt indexPath: IndexPath) {
    let delegate = removeLoadingController(forRowAt: indexPath)?.delegate
    delegate?.tableView?(tableView, didEndDisplaying: cell, forRowAt: indexPath)
}
```

We can see again, in the same fashion as with the previous conformances, how we obtain the delegate, should our **CellController** implement it, and forward the `didEndDisplaying(_:)` action to it, instead of making all of our **CellControllers** conform to all the protocols, which they might not need.

```
private func cellController(forRowAt indexPath: IndexPath) -> CellController {
    let controller = tableModel[indexPath.row]
    loadingControllers[indexPath] = controller
    return controller
}

private func removeLoadingController(forRowAt indexPath: IndexPath) -> CellController?
{
    let controller = loadingControllers[indexPath]
    loadingControllers[indexPath] = nil
    return controller
}
```

For clarity,the helper methods used in the mentioned in the cases above is shown in this code snippet. We see the way in which the neccesary **cellController** for the indexPath is fetched, and how it is removed.

By making these changes to our code, the Cells of our table view need only be conformant to the **CellController** struct, which consists of the three protocol properties, which are the ones that **ListViewController** needs to forward the tableview events generated by the tableview delegate protocol and the ones that need implementation from the prefetching and datasource protocols.

If in the future, further methods were needed from these protocols/delegate it would just suffice to add the new case and forward the events.

Refactoring datasource to use DiffableDataSources

The idea is to migrate the datasource from the conventional tableView datasource to a snapshot-based diffable datasource. For this, we will need to unequivocally identify each of the items of our datasource, namely, each of our CellControllers. For this, we will add a new property to CellController `id` of the type `AnyHashable`, since, too use a value as an identifier, its data type must conform to the `<Hashable>` protocol.

Hashing allows data collections such as `Set`, `Dictionary`, and snapshots — instances of `NSDiffableDataSourceSnapshot` and `NSDiffableDataSourceSectionSnapshot` — to use values as keys, providing quick and efficient lookups. Hashable types also conform to the `Equatable` protocol, so your identifiers must properly implement equality. For more information, see `Equatable`.

Because identifiers are hashable and equatable, a diffable data source can determine the differences between its current snapshot and another snapshot. Then it can insert, delete, and move sections and items within a collection view for you based on those differences, eliminating the need for custom code that performs batch updates.

So, we make our **CellController** conform to `<Hashable>` and `<Equatable>` by hashing and comparing its `id` property, respectively.

Next step, we change our existing datasource on the **ListViewController**, in lieu of the new proposed diffable datasource:

```
//Before
private var tableModel = [CellController]() {
    didSet { tableView.reloadData() }
}
//After
private lazy var dataSource: UITableViewDiffableDataSource<Int, CellController> = {
    .init(tableView: tableView) { (tableView, indexPath, cellController) in
        cellController.dataSource.tableView(tableView, cellForRowAt: indexPath)
    }
}()
```

This method is equivalent to calling the usual `cellForRowAt(_: IndexPath)` from the traditional tableview datasource. So when adding diffable datasources, we will delete the datasource method `cellForRowAt(_: IndexPath)` .

At the same time, we need to modify our `display(_: [CellController])` method that takes in the datasource to display it, to make it work with the diffable datasource:

```
//Before
public func display(_ cellControllers: [CellController]) {
    loadingControllers = [:]
    tableModel = cellControllers
}
//After
public func display(_ cellControllers: [CellController]) {
    var snapshot = NSDiffableDataSourceSnapshot<Int, CellController>()
    snapshot.appendSections([0])
    snapshot.appendItems(cellControllers, toSection: 0)
    dataSource.apply(snapshot)
}
```

We will also need to override the `traitCollectionDidChange(_:)` method from our ViewController, to detect if the user has made content size category changes and update the tableview if need be:

```
public override func traitCollectionDidChange(_ previousTraitCollection:
UITraitCollection?) {
    if previousTraitCollection?.preferredContentSizeCategory !=
traitCollection.preferredContentSizeCategory {
            tableView.reloadData()
    }
}
```

We must also change our method in charge of retrieving the current **cellController** for the **indexPath**, to use the diffable datasource provided method to retrieve the desired item at indexPath `itemIdentifier(for: IndexPath)`:

```
//Before
private func cellController(forRowAt indexPath: IndexPath) -> CellController {
    let controller = tableModel[indexPath.row]
    loadingControllers[indexPath] = controller
    return controller
}
//After
private func cellController(forRowAt indexPath: IndexPath) -> CellController? {
    dataSource.itemIdentifier(for: indexPath)
}
```

What this method does is return the datasource item for the requested tableview indexPath, and return `nil` if no item is provided at that index.

At the same time, we wont be needing the `loadingControllers` property anymore. We had introduced this property because we needed a way to keep track of which Cells were being pre-fetched in order to be able to properly request for pre-fetch and also to be able to cancel prefetching when the cell is out of the view bounds, without losing track, since due to the speed that the user could scroll the tableview it could mean trouble by trying to ask again for prefetchs or cancel-prefetchs that had already been asked for.

But because of the way that diffable datasources manage data, they will always be up to date with the updated data, therefore we can also delete the `removeLoadingController(forRowAt:)` method that we had previously introduced to delete the loading controllers.

As said earlier, to use DiffableDataSources, we need to have a **hashable identifier** in our datasource model, now, while our `id` property needs to be conform to the Hashable protocol, it doesnt need to be a specific identifier, which allows us to use any object that conforms to Hashable as an id. This allows us, for example to use raw immutable models like for example **FeedImage** , which conforms to Hashable.

This way, as we can see in the following piece of code from the **FeedViewAdapter** (which conforms to and therefore implements the `display(_ viewModel: ResourceViewModel)` method). The listViewController's `display(_ cellControllers: [CellController])` is called and provided with a closure mapping that takes in each **FeedViewModel** and maps it the desired **CellController** datasource. When creating the

mapped **CellController**, add our raw **FeedImage** model as the hashable id.

This is very handy, because now our CellController will have access to both the UITableView's delegate/datasource/prefetching protocols from the specific views (FeedImageCellController, ImageCommentCellController), and the raw original model that populates the controller.

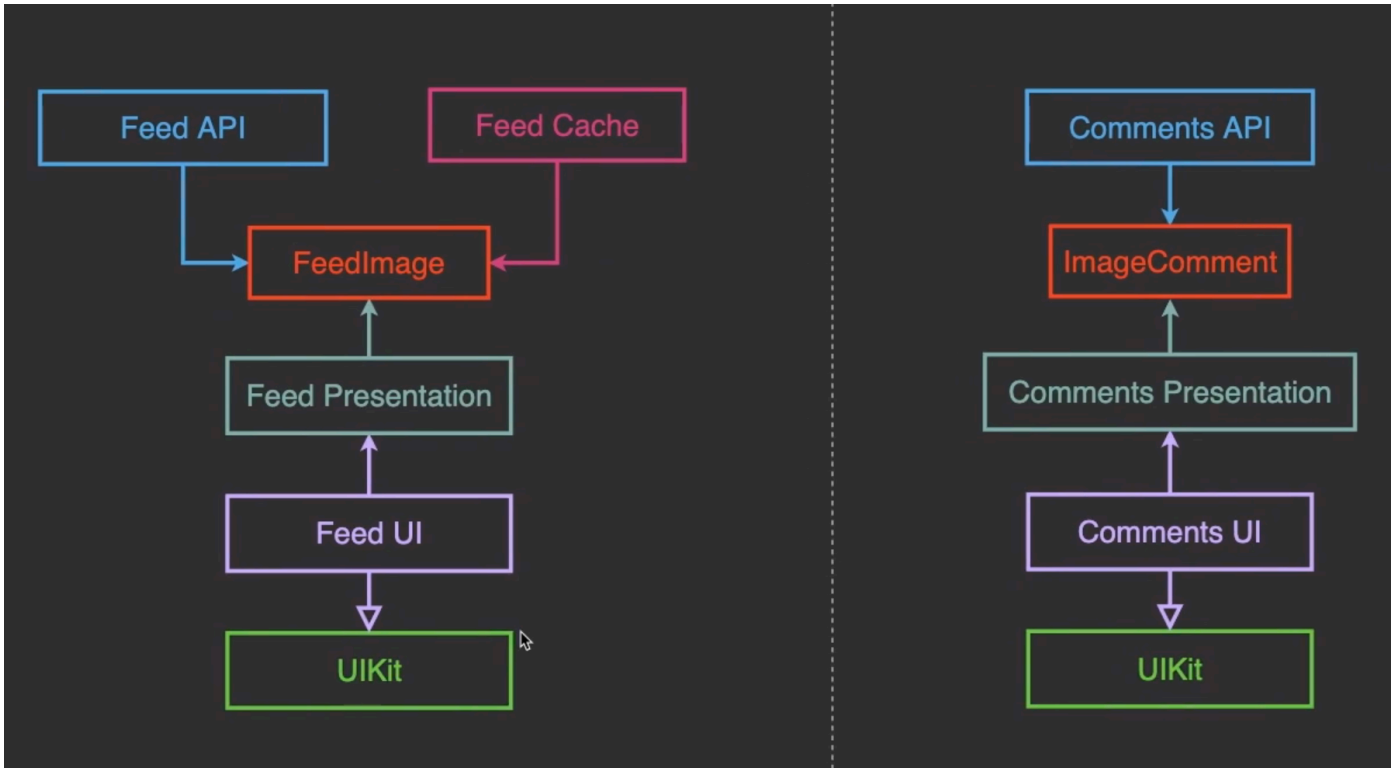
```
private weak var controller: ListViewController?
public typealias ResourceViewModel = FeedViewModel

func display(_ viewModel: ResourceViewModel) {
    controller?.display(viewModel.feed.map { model in
        let adapter = ImageDataPresentationAdapter(loader: { [imageLoader] in
            imageLoader(model.url)
        })

        let view = FeedImageCellController(
            viewModel: FeedImagePresenter.map(model),
            delegate: adapter)

        adapter.presenter = LoadResourcePresenter(
            resourceView: WeakRefVirtualProxy(view),
            loadingView: WeakRefVirtualProxy(view),
            errorView: WeakRefVirtualProxy(view),
            mapper: UIImage.tryMake)

        return CellController(id: model, view)
    })
}
```

Composition and Navigation

Goal:

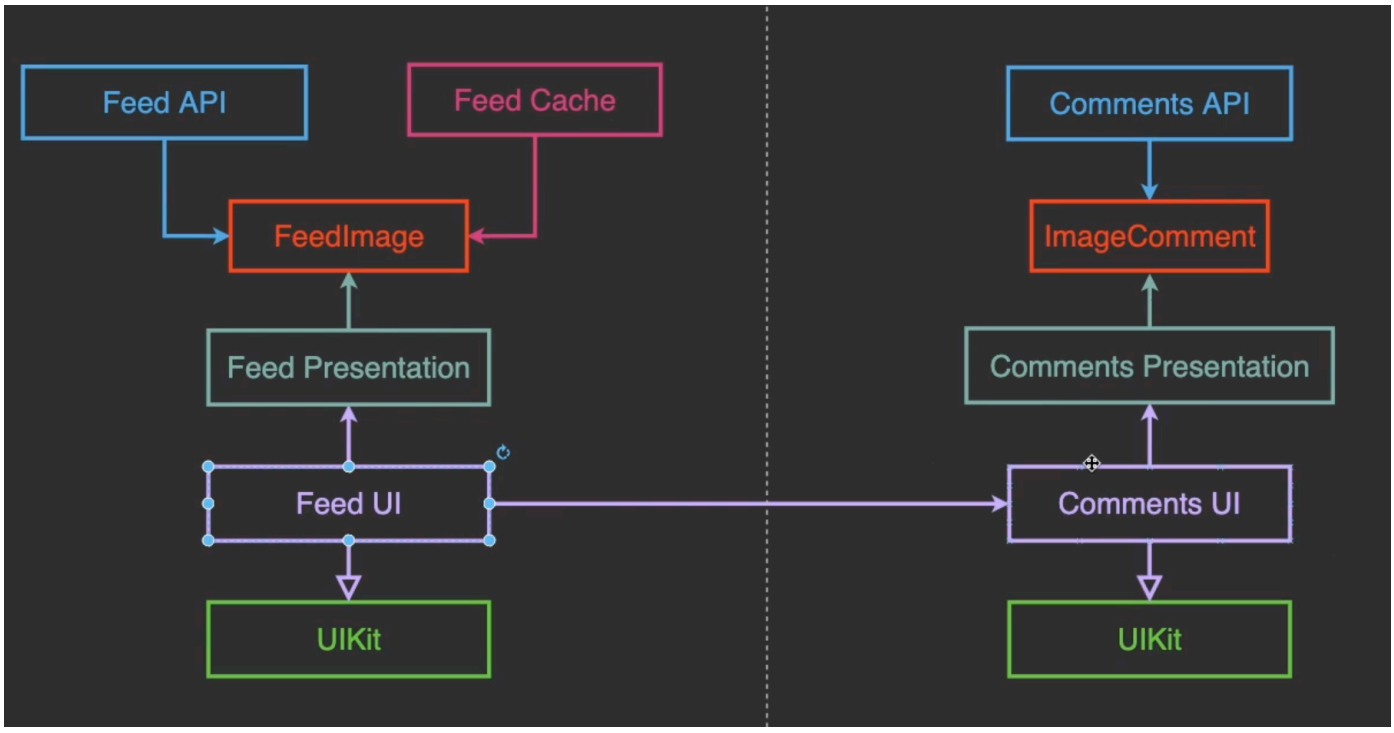
- [x] Display a List of comments when the user taps on an image in the feed.
- [x] At all times , the user should have a back button to return to the feed screen.
- [x] Cancel any running comments API requests when the user navigates back.
- [ ] Integration tests
- [ ] Acceptance tests

Taking a look at the diagram:

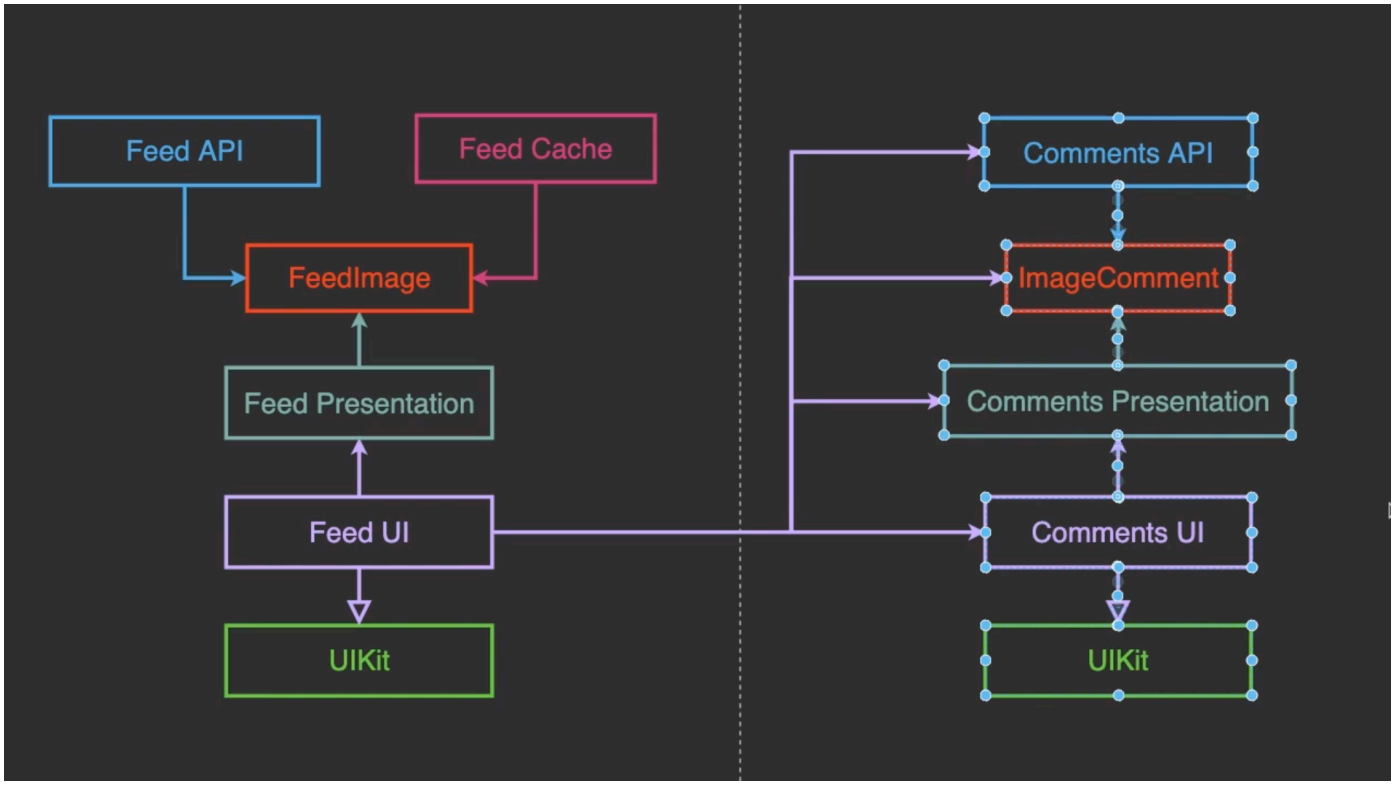


We've finished implementing all the layers, API, Presentation, UI and the data model, so now we need to plug the feed UI with the comments UI.

One way to approach this is the traditional approach, which is pushing the comments UI directly:



The problem is that the Comments scene is quite complex and its comprised by multiple layers (api/presentation/ui), so the **Feed UI** would have to know how to integrate all these layers to be able to create the Comments UI Object Graph



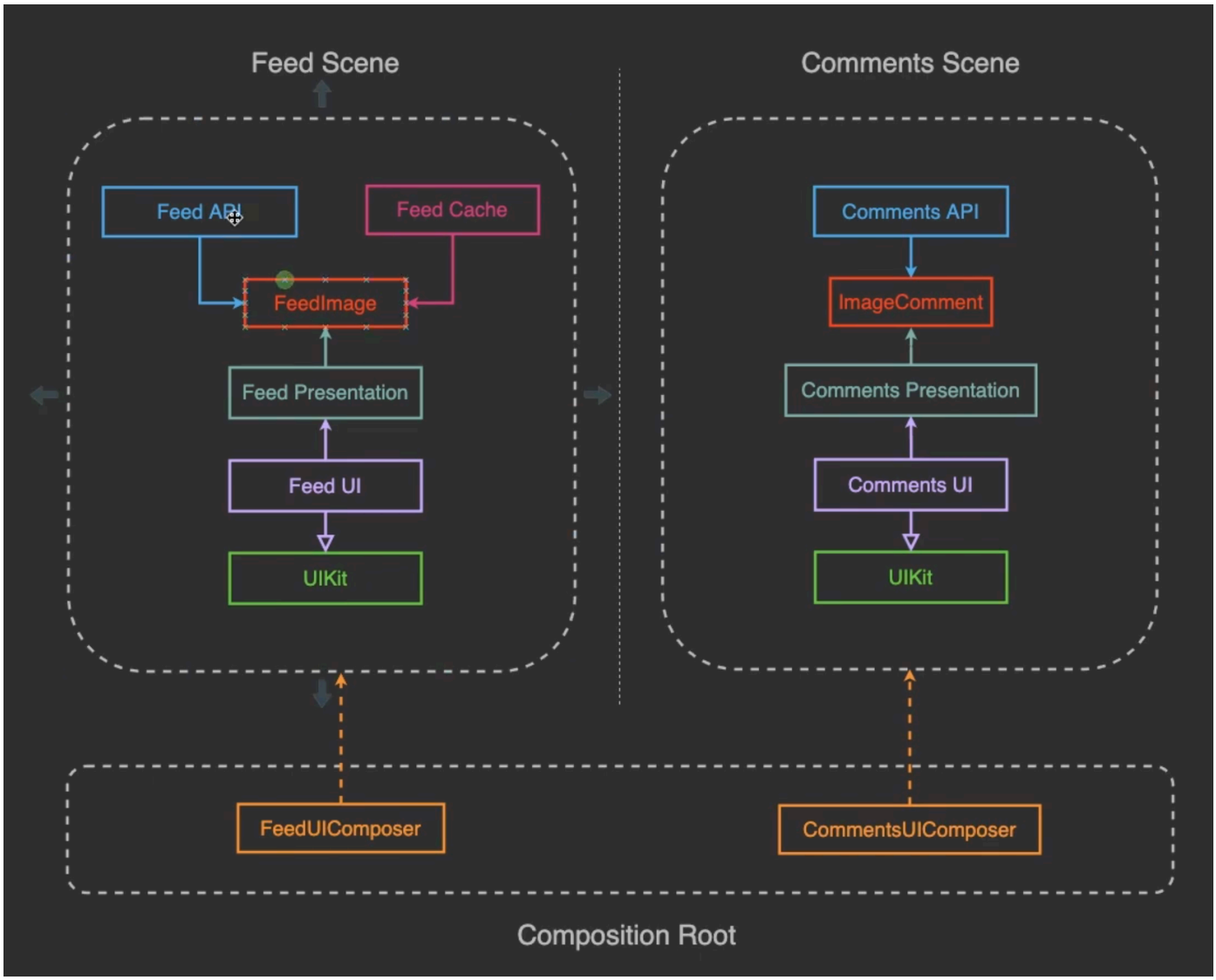
For simple UI transitions that don't require this kind of composition of dependencies, just pushing a viewcontroller within another works well, there is no problem with that, for example if our **FeedImageViewModel** already had access to all the comments (via a `comments: [ImageCommentViewModel]` property) and we were using a **FeedViewController** and we selected a cell, we could use `didSelect(rowAt)` to navigate to the **ImageCommentsViewController** , and to do that we could even show the API method `show(_ VC: sender)` which is a way to present a new VC but the OS decides if it does it by pushing the VC or by presenting it, depending on context.

But this is not our case. If you are dealing with a complex object graph, and need to compose a bunch of layers/modules, it becomes cumbersome to do the simple procedure aforementioned. So we have to handle navigation somewhere else.

Note that there are also cases that even with a simple transition you might not want to couple one VC to the next or have VC's having knowledge of what they are presenting (e.g **ListViewController** is generic and shouldnt know anything about anything it presents).

Another very used way of navigation is by having it in the presentation logic (like for example in viper) but that is just the same as having it in the UI since, now the Presentation layer needs to know how to navigate to the next VC and/or all the other components, and our goal is to have completely de-coupled scenes without any dependencies between eachother.

What we have at the moment



Right now we have the **Feed Scene**, which is composed in the composition root by the **FeedUIComposer**. Now we also need to compose the Comments scene in the composition root, which means we now need to create a **ComponentsUIComposer** as well, that will instantiate the **Comments UI** with all the dependencies it needs. The Composition Root, is your place in the application where you compose all the modules together, and it's the place that allows you to keep all the modules decoupled.

As we can see in the diagram, it is **ONLY** the Composition Root that has dependencies on the modules, and not any other way. The modules have to be decoupled between eachother and **FROM** the **Composition Root.**

As a result, we want to handle navigation between modules inside the **Composition Root,** which already knows about the concrete implementations of said modules.

So, what we will do now is: we will compose the **Comments Scene** in the Composition Root and then handle the navigation also in the **Composition Root.**

As usual we start by creating tests, then creating types, then creating behaviour. First thing we do is copy the already existing **FeedUIIntegrationTests** into the new **CommentsUIIntegrationTests** and we start modifying our code bit by bit. In an analogous way we copy our **FeedUIComposer** into a **CommentsUIComposer** and alongside the tests we start changing the code slowly to reflect our goal.

Our CommentsUIComposer has the following look:

```
public final class CommentsUIComposer {
    private init() {}

    private typealias CommentsPresentationAdapter =
LoadResourcePresentationAdapter<[ImageComment], CommentsViewAdapter>

    public static func commentsComposedWith(
        commentsLoader: @escaping () -> AnyPublisher<[ImageComment], Error>
    ) -> ListViewController {
        let presentationAdapter = CommentsPresentationAdapter(loader: commentsLoader)

        let commentsController = makeCommentsViewController(title:
ImageCommentsPresenter.title)
        commentsController.onRefresh = presentationAdapter.loadResource

        presentationAdapter.presenter = LoadResourcePresenter(
            resourceView: CommentsViewAdapter(controller: commentsController),
            loadingView: WeakRefVirtualProxy(commentsController),
            errorView: WeakRefVirtualProxy(commentsController),
            mapper: { ImageCommentsPresenter.map($0) })

        return commentsController
    }

    private static func makeCommentsViewController(title: String) -> ListViewController
{
        let bundle = Bundle(for: ListViewController.self)
        let storyboard = UIStoryboard(name: "ImageComments", bundle: bundle)
        let controller = storyboard.instantiateInitialViewController() as!
ListViewController
        controller.title = title
        return controller
    }
}
```

With

```
final class CommentsViewAdapter: ResourceView {
    private weak var controller: ListViewController?

    init(controller: ListViewController) {
        self.controller = controller
    }

    func display(_ viewModel: ImageCommentsViewModel) {
        controller?.display(viewModel.comments.map { viewModel in
            CellController(id: viewModel, ImageCommentCellController(model: viewModel))
        })
    }
}
```

Check for deallocation of the Resources

One important thing that we have to ensure is that we dont leak any memory when the user navigates back, and that includes properly cancelling all the comment load requests in progress.

When we navigate back from the ViewController, the whole Object Graph will be deallocated, because the only thing that holds it in memory is the viewcontroller being in the view hierarchy, in the stack. So in principle it should automatically cancel the requests when this happens, since our **LoadResourcePresentationAdapter** class holds a reference to the **cancellable** when it is running a request and when the adapter is deallocated the cancellable is also deallocated and it calls the Combine `cancel()` method when the AnyCancellable is deinitialized.

So, as long as we don't have any memory leaks this process operation should be cancelled automatically when the object graph is deallocated, and we already have proved that it is deallocated because we have the `trackForMemoryLeaks` in the tests.

But if we want to prove this, we can add one more test. Our goal is to test that before we make our SUT nil, the number of times that the cancel method was called is exactly 0, and after we make it nil, (it gets deallocated), the number of times that the cancel method is called is exactly 1:

```
func test_deinit_cancelsRunningRequest() {
    var cancelCallCount = 0

    var sut: ListViewController?

    autoreleasepool {
        sut = CommentsUIComposer.commentsComposedWith(commentsLoader: {
            PassthroughSubject<[ImageComment], Error>()
                .handleEvents(receiveCancel: {
                    cancelCallCount += 1
                }).eraseToAnyPublisher()
        })
```

```
        sut?.loadViewIfNeeded()
    }

    XCTAssertEqual(cancelCallCount, 0)

    sut = nil

    XCTAssertEqual(cancelCallCount, 1)
}
```

We need to use the `autoreleasepool`, because ListViewController is being kept referenced in memory by the auto relase pool as an autoreleased object that is kept until the next cycle because probable this test is running on its autorelease pool, and it gets dereferenced after the method returns, but since we need to make sure it works before, we use autoreleasepool to create our own autoreleasepool to hold our object locally. **autoreleasepool** comes from old Objective-C runtime, and its not really needed in Swift, since Swift relies on ARC, which is a deterministic way to count memory references. The only reason to use it is in cases where everything else has failed but we can still see the object leaked in the memory graph. In this case it probably got autoreleased captured since the way of instantiating viewcontrollers from storyboards/xibs still relies on underlying uikit code that was made in objective c.

What happens here is that, the reference release only gets invocated after our test finishes, in the tearDown method, AFTER our `trackForMemoryLeaks` method has been executed, that is why if we dont use the **autoreleasepool** to capture our object, our memory leak tracker still finds it leaking.

Registering user touch selection of the CellController Cells

Next step is to navigate to the comments' section of the image the user selects (taps). To do this, the most basic approach is that we need to implement `<UITableViewDelegate>` protocol in our **ListViewController** and implement the `tableView(:didSelectRowAt: indexPath)` , to execute the navigation we are interested in.

As we did before, the idea is to forward **ListViewController's** event to the appropriate specific **CellController** so that each specific class/client/controller can implement `tableView(:didSelectRowAt:indexPath)` and run whatever piece of code we want to. In this case, our **FeedImageCellController** will of course implement the tableViewdelegate's needed methods, and it will need to execute code to the navigation. This will be done by executing a closure handler :

```
private let selection: () -> Void
```

that will be executed whenever a cell is tapped on by the user.

As we mentioned before, in order not to couple any of the modules, all the navigation logic and module interaction is done in the **Composition** Root, where both the **FeedUIComposer** and the **ImageCommentsUIComposer** live.

This means that we will inject the `selection: () ->Void` closure into the **FeedImageCellController** at the moment of composition in the Composition Root.

As usual, to develop this, we begin with a test inside the FeedUIIntegrationTests, our goal is to test that selecting an image notifies the selector handle:

```
func test_imageSelection_notifiesHandler() {
    let image0 = makeImage()
    let image1 = makeImage()
    var selectedImages = [FeedImage]()
    let (sut, loader) = makeSUT(selection: { selectedImages.append($0) })

    sut.loadViewIfNeeded()
    loader.completeFeedLoading(with: [image0, image1], at: 0)

    sut.simulateTapOnFeedImage(at: 0)
    XCTAssertEqual(selectedImages, [image0])

    sut.simulateTapOnFeedImage(at: 1)
    XCTAssertEqual(selectedImages, [image0, image1])
}
```

to carry out this test we will have to modify step by step the required code and add what's needed. The idea here is that by stablishing what we want to achieve, we can then make it happen.

For starters, we need to implement the UITableView's delegate method `tableView(_: didSelectRowAt: indexPath)` in the **ListViewController**, and in the same fashion as with the rest of the delegate/protocol/datasource methods described in the previous chapters, this implementation looks like:

```
public override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    let delegate = cellController(at: indexPath)?.delegate
    delegate?.tableView?(tableView, didSelectRowAt: indexPath)
}
```

We get the CellController corresponding to the indexPath, and should it conform to UITableViewDelegate, it will be forwarded the `tableView(_: didSelectRowAt: indexPath)` method.

Next, we have to implement `tableView(_: didSelectRowAt: indexPath)` in **FeedImageCellController** to receive the forwarded delegate methods from **ListViewController**, and as said before, it will just execute the **selection handler** closure:

```
private let selection: () -> Void
.......
.......
.......
public func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
   selection()
}
```

To finish this part, we modify the corresponding **FeedUIComposer**, and **FeedViewAdapter** classes, to take in a selection handler parameter in their initializers.

Showing Comments on Image selection

The next step is to display on screen, the comments for the selected image. Which means that it's time to compose the navigation and behavior in the **Composition Root** (Our **SceneDelegate**).

As per usual we start with a test and its related code, and we build-up the neccesary code.

In our **Composition Root** (**SceneDelegate**), we implement the `selection handler` closure that specifies what to do when a image cell is tapped:

We mentioned earlier that the handler executed in the **FeedImageCellController** is a closure that takes in no parameters and returns no parameters, `selection: () -> Void`, and that is correct, however at the composition root stage, and given that the **FeedImageCellController** is instanciated with the **FeedUIComposer** using the **FeedViewAdapter** , our `selection handler` has a different signature:

```
selection: (FeedImage) -> Void
```

Which is then *adapted* by the **FeedViewAdapter** into the required closure that the `tableView(_: didSelectRowAt: indexPath)` executes, inside the **FeedImageCellController**.

So, the method to be invoked when a user taps the desired Cell is:

```
private func showComments(for image: FeedImage) {
    let url = baseURL.appendingPathComponent("/v1/image/\(image.id)/comments")
    let comments = CommentsUIComposer.commentsComposedWith(commentsLoader:
makeRemoteCommentsLoader(url: url))
        navigationController.pushViewController(comments, animated: true)
}
```

Which as can be seen has the same signature as the **selection** handler that we need. For this composition we also need to make the `makeRemoteCommentsLoader(url:)` method:

```
private func makeRemoteCommentsLoader(url: URL) -> () -> AnyPublisher<[ImageComment],
Error> {
    return { [httpClient] in
           return httpClient
               .getPublisher(url: url)
               .tryMap(ImageCommentsMapper.map)
               .eraseToAnyPublisher()
    }
}
```

That, in an analog way to the :

```
private func makeRemoteFeedLoaderWithLocalFallback() -> AnyPublisher<[FeedImage],
Error>
```

Returns a publisher with the desired resource or error, that makes loader to make the network request.

Also, since the **FeedImageCellController** is the entry point screen in the view, it is only normal that we embed it inside a navigation controller and make that the rootViewController in our main window, inside the Composition Root (**SceneDelegate**)

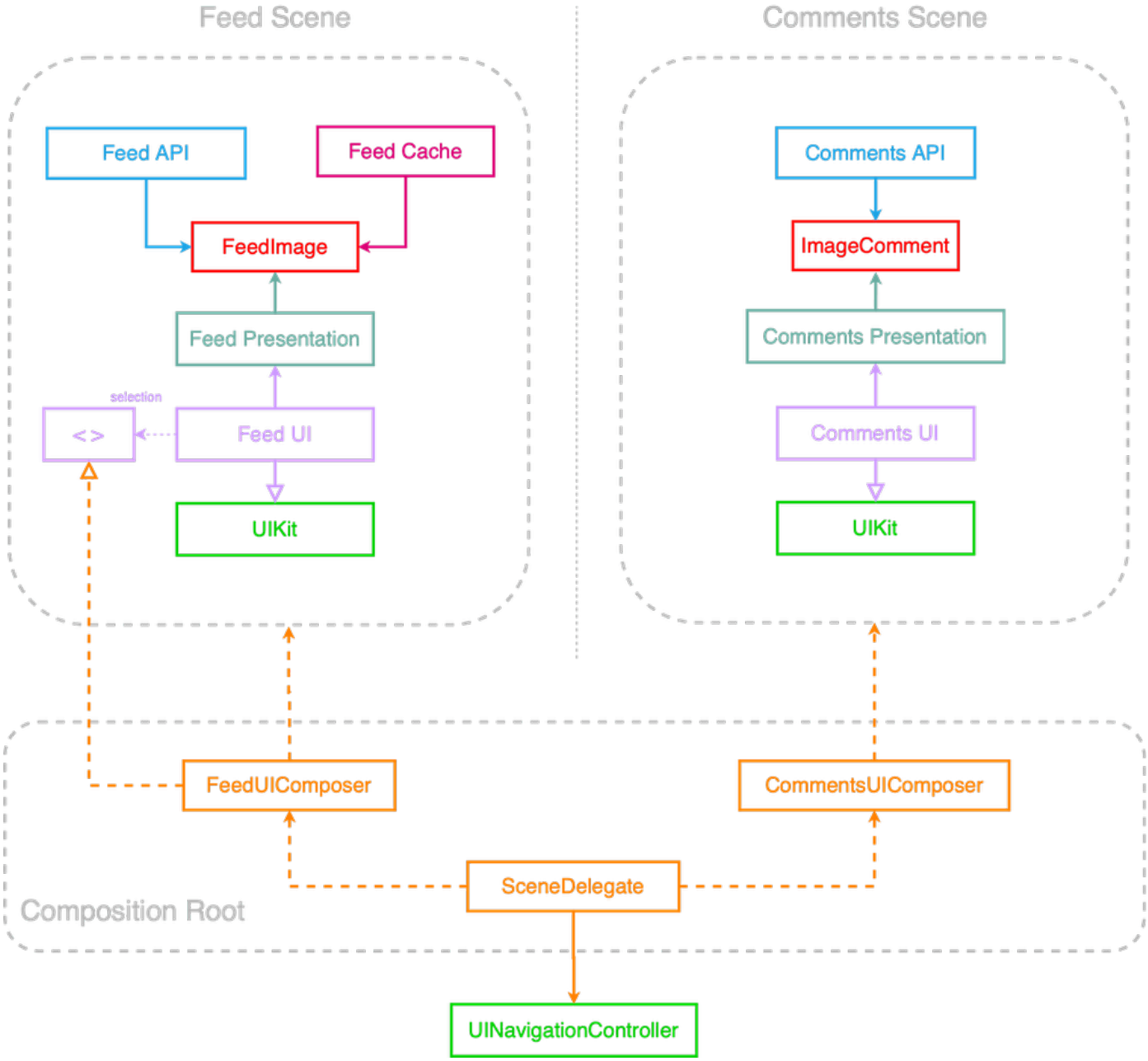
```
private lazy var navigationController = UINavigationController(
        rootViewController: FeedUIComposer.feedComposedWith(
            feedLoader: makeRemoteFeedLoaderWithLocalFallback,
            imageLoader: makeLocalImageLoaderWithRemoteFallback,
            selection: showComments))

func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {
        guard let scene = (scene as? UIWindowScene) else { return }

        window = UIWindow(windowScene: scene)
        configureWindow()
}

func configureWindow() {
    window?.rootViewController = navigationController
    window?.makeKeyAndVisible()
}
```

Cleaning up the API Endpoints

To clean up the API Endpoints we create two API Files inside the Feed API and Image Comments API modules:

```
//FeedEndpoint
public enum FeedEndpoint {
    case get

    public func url(baseURL: URL) -> URL {
        switch self {
        case .get:
            return baseURL.appendingPathComponent("/v1/feed")
        }
    }
}

//ImageCommentsEndpoint
public enum ImageCommentsEndpoint {
    case get(UUID)

    public func url(baseURL: URL) -> URL {
        switch self {
        case let .get(id):
            return baseURL.appendingPathComponent("/v1/image/\(id)/comments")
        }
    }
}
```

Final App Architecture



Our Final diagram shows how we managed to keep our modules separated and decoupled from eachother. Everything happens in the Composition Root, which is our **SceneDelegate**.

Pagination Methods

Now that the basic project is done, we will delve into the different pagination strategies, both to update the UI and to fetch and save the paginated data.

Things that we will see:

- [x] Common Pagination Methods.
- [ ] Pagination UI/UX best practices

- [ ] How to paginate linear content from an API
- [ ] How to cache page results

Goals

- [ ] Layout
- [ ] Infinite Scroll Experience
  - [ ] Trigger "Load more" action on scroll to bottom
    - [ ] Only if there are more items to load
    - [ ] Only if not already loading
  - [ ] Show loading indicator while loading
  - [ ] Show error message on failure
    - [ ] Tap on error to retry

Common Pagination Methods

Limit/Offset or Page-Based

This pagination type requires you to provide a `limit` of items per page, and then you go page by page.

```
/feed?limit={limit}&page=[page]
```

For example executing `/feed?limit=2&page=1` will bring us **itemA** and **itemB**, and subsequently if we execute `/feed?limit=2&page=2` we will get **itemC** and **itemD**. We could also change the limit, for example to 4, and get all four items in a single request `/feed?limit=4&page=1`.

Pros

- Easy to implement in the backend
- Supports random access to pages

Cons

- Inefficient for large offsets (few hundred/thousands)
- Inconsistent paging (because the pages' content isn't fixed, and it can happen that after we start requesting paginated items, new items are added to the dataset and that shifts the items out of position making the client and server out of sync.) :

Example: We requested like shown before `/feed?limit=2&page=1` for page 1, then `/feed?limit=2&page=2` for page 2, and we got items A,B,C,D, then, before requesting for page 3, one new item X is added, meaning that when requesting `/feed?limit=2&page=3`, instead of getting back items E,F, we will get D,E, because the new item was added on top of page 1 and thus shifting all the items. This leads to inconsistencies like duplications and omissions, since we won't be getting item X and we will be getting a duplicate of item D.

- Limit size must be constant while traversing pages
- Not great for caching (due to its inconsistency)

Good For

- Small collections of data
- Short-lived content (seconds/minutes)
- Content that doesn't change as often
- Discardable content (without caching)

Keyset based (aka Seek Method) (Will implement this here.)

In this method you provide an `after_key` or a `before_key` to the backend, aswell as an item limit `limit`, where both the **after/before** keys make reference to the items identifiers:

```
/feed?limit={limit}&after_key={itemX.key}  or  /feed?limit={limit}&before_key=
{itemX.key}
```

For example, if we execute: `/feed?limit=2`, we would get itemA, itemB, then we execute: `/feed?limit=2&after_key={itemB.key}`, we will get itemC and itemD, if after that we execute: `/feed?limit=3&after_key={itemD.key}` we would get tambien itemE, itemF and itemG.

But now, lets say that we made the first request `/feed?limit=2` and got itemA and itemB, but after that, an itemX is added to the dataset to the top, if we execute `/feed?limit=2&after_key={itemB.key}`, we will still get the correct results: itemC and itemD, because we are clearly asking that we want 2 items after the itemB.

So say that after this itemX was added we want to retrieve it, that is no problem, we just execute: `/feed?limit=2&before_key={itemA.key}` and we will get itemX and any other new item that was added.

Pros

- Fast
- Consistent in both directions (after/before)
- Allows consistent caching

- Flexible limit size

Cons

- Harder to implement in the backend
- Doesn't support random access to pages (unless the id is a sequential number, although this is not something that is really needed)

Good for

- Large collections of linear/sequential data
- Long-lived content
- Caching

Keyset Pagination, Implementation

Today we will take an "outside-in" approach, which means, we will start by the UI instead of by the logic.

First thing we want to add is the loading spinner at the bottom of the screen when the user is scrolling to get more feed items. (or error if it fails).

One idea is that we could add it to the ListViewController directly, in the footer, and also adding another closure `onLoadMore` to execute code when this stage is reached. Adding this in the ListViewController would not be a bad thing, since pagination is a generic concept to any kind of list that needs pagination. However, we shouldn't be adding any feature specific functionality to the LVC, because it is a generic component. Because in this case, only the feed needs this functionality. If we added the feature in the generic LVC, then, when we dont use it, we would need to disable it, probably with a boolean, and all of this addition of side-effects and code that won't be used is not good.

Generic objects should only have features that all the implementing clients will use. Our goal is to add features without modifying existing code, being true to the open-close principle.

We could inject a `loadMore` closure and loading gear to the **footer** in the in the **Feed Storyboard** since for example UIKit already provides us with a refresh mechanism. The problem is that when to show it and for how long and etcetera requires application logic which can't happen in Storyboards, so we would have to add this logic in the LVC and we don't want to do that.

What we could do, though is make the whole spinning mechanism to load more data as a **CellController** cell, and inject it to the ListViewController so the LVC doesn't need to know about these changes. Basically what we do is create a custom cell that conforms to the **CellController** type that we populate our LVC with, and just add it always in the last position.

To test-drive this idea, we will create a new FeedSnapshot test to test this, and start building up our CellController cell. The idea is to check that we are successfully displaying our new cell in the tableview.

For this, we will need to create both a **LoadMoreCellController** and a **LoadMoreCell**. The **LoadMoreCellController** will implement the datasource protocols from tableView to create the **LoadMoreCell**, in the `tableView(_:cellForRowAt: indexPath)` . The **LoadMoreCellController** must also implement the **ResourceLoadingView**, since we want it to start showing the spinner when our **ResourceLoadingViewModel** is loading.

Also, our **LoadMoreCellController**, will hold a non-reusable cell, therefore there is no need to dequeue it from the tableview, and we can define it directly as a constant inside the controller.

Regarding the **LoadMoreCell** it will only contain a UIActivityIndicatorView, which will be centered inside the **contentView** and it will hold a public `isLoading` property with a getter and a setter to either get the loading status or to set the spinner to start/stop rotating.

```
public class LoadMoreCell: UITableViewCell {
    private lazy var spinner: UIActivityIndicatorView = {
        let spinner = UIActivityIndicatorView(style: .medium)
        contentView.addSubview(spinner)

        spinner.translatesAutoresizingMaskIntoConstraints = false
        NSLayoutConstraint.activate([
            spinner.centerXAnchor.constraint(equalTo: contentView.centerXAnchor),
            spinner.centerYAnchor.constraint(equalTo: contentView.centerYAnchor),
            contentView.heightAnchor.constraint(greaterThanOrEqualToConstant: 40)
        ])

        return spinner
    }()

    public var isLoading: Bool {
        get { spinner.isAnimating }
        set {
            if newValue {
                spinner.startAnimating()
            } else {
                spinner.stopAnimating()
            }
        }
    }
}
```

And the **LoadMoreCellController** :

```
public class LoadMoreCellController: NSObject, UITableViewDataSource {
    private let cell = LoadMoreCell()
```

```
    public func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
-> Int {
        1
    }

    public func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
-> UITableViewCell {
        cell
    }
}

extension LoadMoreCellController: ResourceLoadingView {
    public func display(_ viewModel: ResourceLoadingViewModel) {
        cell.isLoading = viewModel.isLoading
    }
}
```

That is the basic setup for the spinner loading/not loading, now we have to configure the error message. After we created the snapshot tests to set what we expect to see. We arrive to the conclusion that our LoadMoreCellController must also implement `<ResourceErrorView>` protocol, to display errors. So we add conformance and implement the method:

```
extension LoadMoreCellController: ResourceErrorView {
    public func display(_ viewModel: ResourceErrorViewModel) {
        cell.message = viewModel.message
    }
}
```

To do this, we need to add an error message UILabel to the **LoadMoreCell** :

```
public class LoadMoreCell: UITableViewCell {
    private lazy var spinner: UIActivityIndicatorView = {
        let spinner = UIActivityIndicatorView(style: .medium)
        contentView.addSubview(spinner)

        spinner.translatesAutoresizingMaskIntoConstraints = false
        NSLayoutConstraint.activate([
            spinner.centerXAnchor.constraint(equalTo: contentView.centerXAnchor),
            spinner.centerYAnchor.constraint(equalTo: contentView.centerYAnchor),
            contentView.heightAnchor.constraint(greaterThanOrEqualToConstant: 40)
        ])

        return spinner
    }()
```

```
    private lazy var messageLabel: UILabel = {
        let label = UILabel()
        label.textColor = .tertiaryLabel
        label.font = .preferredFont(forTextStyle: .footnote)
        label.numberOfLines = 0
        label.textAlignment = .center
        label.adjustsFontForContentSizeCategory = true
        contentView.addSubview(label)

        label.translatesAutoresizingMaskIntoConstraints = false
        NSLayoutConstraint.activate([
            label.leadingAnchor.constraint(equalTo: contentView.leadingAnchor,
constant: 8),
            contentView.trailingAnchor.constraint(equalTo: label.trailingAnchor,
constant: 8),
            label.topAnchor.constraint(equalTo: contentView.topAnchor, constant: 8),
            contentView.bottomAnchor.constraint(equalTo: label.bottomAnchor, constant:
8),
        ])

        return label
    }()

    public var isLoading: Bool {
        get { spinner.isAnimating }
        set {
            if newValue {
                spinner.startAnimating()
            } else {
                spinner.stopAnimating()
            }
        }
    }

    public var message: String? {
        get { messageLabel.text }
        set { messageLabel.text = newValue }
    }
}
```

We call this label "messageLabel" and not "errorLabel", because while not it's designed to be used as an error label, it could be used as a way to display the last updated date, for example.

Paging Implementation

UI Integration

Now that the UI components are ready, we have to test them in integration to make sure we trigger reloadRequest when we scroll to the bottom of the scrollView, but we should only trigger it after loading the first set of items and if there are more items to load, but how can we know if we've loaded the first set of items and if there are more items to load?

Looking at the FeedUIComposer, we see that the result we get from the **FeedLoader** is an array of **FeedImage**, so we can say that if we receive a first set of items, we could load more items. but how can we know if there are more items to load?

Since nothing in the FeedImage model can tell us this, ideally we need a new type that has information about the both the data we need and about pagination. For this, we make a new type that takes in a generic , **PaginatedFeed**, which will contain an array of Item, representing the data we got back from the network request, and an optional `loadMore` closure, that if it isnt nil, it means we have more items to load.

This way we decouple the UI from pagination details, the UI only has a closure for loading the next page, but it doesnt know which logic or algorithm for pagination is going on behind the scenes, meaning it doesn't care about what kind of pagination is happening, allowing us to change the pagination if we need/want to, without breaking our UI composition. Because of this, caching is much easier aswell, because it's not driven by the UI. Normally we see pagination where the caching strategy is driven by the UI and this can lead to problems, of coupling the UI with the algorithm in use and the caching.

This way the UI doesn't need to know how our feed data was loaded, which algorithm was used, it just knows that if there is more it just calls `loadMore` closure to which we pass nothing and we expect a result at some point.

```
struct Paginated<Item> {
    let items: [Item]
    let loadMore: (() -> AnyPublisher<Paginated<Item>, Error>)?
}
```

This way we keep traversing from one page to the other and other, and other, until we got all the pages.

Another benefit of this approach is that the UI doesnt need to know when to append new data and how to deal with duplication, it's up to the clients that create the pages, to deal with the de-duplication, (if using page-based pagination).

Pagination is an API specific detail, Ideally we would want to always have every collection of items available in memory at all times, it just happens that we need to load a little bit at a time because of resource limitations, that is an API Infrastructure detail. So we can move this Paginated data model type into the Shared API module. But, it is referencing Combine, which means that we are coupling the shared API Module with Combine, which would make every client of the Shared API module also depend on Combine, maybe we are happy doing this, but we could instead use **Closure-Based-Loading** , since closures are native first-class citizens in swift.

So what we do is to create a typealias with a `public typealias LoadMoreCompletion = Result<Paginated<Item>,Error> -> Void` and we make our loadMore closure take in an escaping `LoadMoreCompletion` closure as argument, and return Void:

```
import Foundation

public struct Paginated<Item> {
    public typealias LoadMoreCompletion = (Result<Self, Error>) -> Void

    public let items: [Item]
    public let loadMore: ((@escaping LoadMoreCompletion) -> Void)?

    public init(items: [Item], loadMore: ((@escaping LoadMoreCompletion) -> Void)? =
nil) {
        self.items = items
        self.loadMore = loadMore
    }
}
```

This way we do not need our clients to depend on Combine anymore. (maybe our clients are using older ios versions where combine is not available).

This doesn't mean we cant use Combine anymore, we just compose it in the Composition root using some combine helpers that will convert this closure-based approach into a publisher.

This **Paginated** type provides us what we need: a way of knowing if there are more items to load.

Now we need to replace all the parts of the codebase that refer to arrays of **FeedImage** to return **Paginated** instead of **Array** , as a result we find that in the FeedUIComposer, at the moment of creating the presentation adapter's presenter we dont need to use the **FeedPresenter.map** method, since all that mapping does is wrapping the **Array** into the **FeedViewModel** , which isn't necessary now since our types have been changed, therefore we can pass the **Paginated** directly to the adapter.

Inside our SceneDelegate (Composition Root), we need to modify the `makeRemoteFeedLoaderWithLocalFallback()` method, so that it returns our new wrapped type **Paginated** :

```
private func makeRemoteFeedLoaderWithLocalFallback() ->
AnyPublisher<Paginated<FeedImage>, Error> {
    let url = FeedEndpoint.get.url(baseURL: baseURL)

    return httpClient
        .getPublisher(url: url)
        .tryMap(FeedItemsMapper.map)
        .caching(to: localFeedLoader)
        .fallback(to: localFeedLoader.loadPublisher)
        .map {
            Paginated(items: $0)
        }
        .eraseToAnyPublisher()
}
```

Now that we have both the UI , the Pagination type and we have changed the instances of **Array** for **Paginated**, we have to wire everything together.

To begin, are going to implement integration tests, to make sure we got all the infinite scroll experience features covered.

```
 func test_loadMoreActions_requestMoreFromLoader() {
      let (sut, loader) = makeSUT()
      sut.loadViewIfNeeded()
      loader.completeFeedLoading()

      XCTAssertEqual(loader.loadMoreCallCount, 0, "Expected no requests until load-more
action")

      sut.simulateLoadMoreFeedAction()
      XCTAssertEqual(loader.loadMoreCallCount, 1, "Expected load-more request")
}
```

We add this test since what we want is that the loader spy loadMoreCallCount propety is only 1 after simulating a loadMoreFeedAction(). From here we start completing the code in order to make it pass.

Here we have to pay special attention to the mock method: `func simulateLoadMoreFeedAction()` , since we need to decide how to properly mock the behaviour of getting to the loadMore case.

We could keep track of the scrolling offset, but since we are using a CellController cell to display the loading spinner/ message, and we know that it will always be the last cell of the array, we can use the `willDisplay:cellForRowAt:indexPath` delegate method from UITableViewDelegate protocol, and track for the **LoadMoreCell**.

In order not to mix our **FeedImageCellController** cells with our **LoadMoreCell**, instead of appending our **LoadMoreCellController** to the bottom of our CellControllers array, what we are going to do is append it to the 0th position of the next section.

This way, we have a section with the FeedImages, a section with the LoadMore, etc. (But it could be done all in a single section without problem).

our test helper method results:

```
func simulateLoadMoreFeedAction() {
      guard let view = cell(row: 0, section: feedLoadMoreSection) else { return }
      let delegate = tableView.delegate
      let index = IndexPath(row: 0, section: 1)
      delegate?.tableView?(tableView, willDisplay: view, forRowAt: index)
}
```

where `feedLoadMoreSection` is 1 (since the `feedImagesSection` was 0). The guard is added because the cell may not exist, because you may not be able to load more, because there is no more to load (although that could be fixed because if there is nothing more to load we could add a message), so we check that it exists and if it does, we trigger its appearance to simulate a loadMore action.

For this procedure to work, we need to add UITableViewDelegate conformance to our LoadMoreCellController, and implement the `tableView(_: willDisplay cell: forRow at: indexPath)`, where we will execute a callback method. So we add a new `callback: () -> Void` property to our **LoadMoreCellController** that we will inject via initializer to our **LoadMoreCellController**

Of course with this procedure we also have to append a new section into the feed with the **LoadMoreCellController**. The creation of the feed CellControllers is done in the **FeedViewAdapter**, inside the `display(_: Paginated<FeedImage>)` method.

What we need to do there is to append a new section to the array of CellControllers. To do this, we assign the mapping of the viewModel into the CellControllers to a property (which we were returning directly), and now we need to create a

```
let loadMoreCellController = LoadMoreCellController({
  viewModel.loadMore?( { _ in })
})
```

What this callback will do, is simply trigger the `loadMore` action from the viewModel (Where we will need to pass another closure to execute during the loadMore, which we will do later).

We also need to create the new section we mentioned before, composed by an array of CellControllers that will only contain one item, the loadMoreCellController:

```
let loadMoreSection = [CellController(id: UUID(), loadMoreCellController)]
```

We can now either append the loadMoreSection to the feedSection by doing:

```
controller?.display(feed + loadMoreSection)
```

The problem with this, is that by doing this everything is going to be in the first section (section 0) and that is not what we want, what we want is to have mutliple sections, for this, we need to modify our `ListViewController`'s `display(_ cellControllers: [CellController])` to take in a variadic parameter and be able to call:

```
controller?.display(feed, loadMoreSection)
```

Our new display method in **ListViewController** is:

```
public func display(_ sections: [CellController]...) {
      var snapshot = NSDiffableDataSourceSnapshot<Int, CellController>()
      sections.enumerated().forEach { section, cellControllers in
          snapshot.appendSections([section])
          snapshot.appendItems(cellControllers, toSection: section)
      }
      dataSource.apply(snapshot)
}
```

We can see that we have made a change for the internal name of the arguments received, from "cellControllers" to "sections", and now we take in the array of array of cell controllers, and for each of them, we append them to the respective sections, section 0 for the feed cellControllers, and section 1 for the loadMoreCellController.

There is a problem though, what if the callback request fails?. As it stands now, it will only try again when the "willDisplay" method is fired again by the delegate, which will only happens when the cell is removed from the hierarchy and is then re-added. To solve this we could observe the scrolling events aswell, and trigger the callback while the cell is visible (**this will be explained later in this session**).

On the previous test, there was something that we didn't test that we will test now, that is, if the request hasnt completed and another loadMore is triggered it shouldnt execute (this not incrementing the loadMoreCallCount in the spy). Initially this test fails which means that it's loading again and again when the loadMore triggers, this is not what we want.

To do this we need to add a guard statement to our willDisplay delegate method implementation in LoadMoreCellController, so that it checks if the cell is loading, and if it is, not to execute the callback.

```
public func tableView(_ tableView: UITableView, willDisplay: UITableViewCell, forRowAt
indexPath: IndexPath) {
        guard !cell.isLoading else { return }
        callback()
}
```

Another thing that arose while doing these tests (and why tests are so important), is that we have never set `isLoading` property from the cell, because we never hooked up the methods with the cellcontroller.

```
extension LoadMoreCellController: ResourceLoadingView {
    public func display(_ viewModel: ResourceLoadingViewModel) {
        cell.isLoading = viewModel.isLoading
    }
}

extension LoadMoreCellController: ResourceErrorView {
    public func display(_ viewModel: ResourceErrorViewModel) {
           cell.message = viewModel.message
       }
}
```

Meaning, we are implementing the display methods from the ResourceLoadingView and the ResourceErrorView protocols but we never call these methods, therefore our cell loading status isnt changing.

This is because we never created a presentation adapter that will handle the resource loading.

For this, in the **FeedViewAdapter**, we need to create presentation adapter for the new page, and we can reuse the existing **LoadResourcePresentationAdapter**, but with a different type and name:

```
private typealias LoadMorePresentationAdapter =
LoadResourcePresentationAdapter<Paginated<FeedImage>, FeedViewAdapter>
```

This presentation adapter will load **Paginated** , and its View type will be a FeedViewAdapter because its loading a feed.

To do this, since our Paginated type is decoupled from combine, we need to create a Combine adapter, in the CombineHelpers class:

```
public extension Paginated {
    var loadMorePublisher: (() -> AnyPublisher<Self, Error>)? {
        guard let loadMore = loadMore else { return nil }

        return {
            Deferred {
                Future(loadMore)
            }.eraseToAnyPublisher()
        }
    }
}
```

This is the bridging from the closure that our `loadMore` from the **Paginated** type takes in, into the **AnyPublisher** that we need.

This way we can easily keep combine decoupled and have the helpers in the composition root.

With this combine helper, we can now create our **LoadMorePresentationAdapter** with our `loadMorePublisher` , and use the adapter's loadResource to create our **LoadMoreCellController**:

```
 guard let loadMorePublisher = viewModel.loadMorePublisher else {
            controller?.display(feed)
            return
}
let loadMoreAdapter = LoadMorePresentationAdapter(loader: loadMorePublisher)

let loadMore = LoadMoreCellController(callback: loadMoreAdapter.loadResource)
```

Here, we check that the loadPublisher method isnt nil, otherwise we display a single feed section in our ListViewController. If all is okay, we crate the adapter, and the loadMore cell controller:

```
loadMoreAdapter.presenter = LoadResourcePresenter(
    resourceView: self,
    loadingView: WeakRefVirtualProxy(loadMore),
    errorView: WeakRefVirtualProxy(loadMore),
    mapper: { $0 })

let loadMoreSection = [CellController(id: UUID(), loadMore)]

controller?.display(feed, loadMoreSection)
```

finally, we create our adapter's presenter using `self` as resource, since the adapter's resource is of the type **FeedViewAdapter**, and using our loadMore cell controller as both our loading and error view's. Since **FeedViewAdapter**'s ResourceViewModel is the same type as the adapter's presenter's recipient mapper, we simply pass the closure's captured value, as mapper. At this point it means we have both the needed sections to display on our **ListViewController**

Now, we have to test that our spy loadMoreCallCount is successfully incrementing when we request more pages, either with success or with failure, but bearing in mind it isnt the last page, and if it is, it shouldn't increment the call count spy:

```
loader.completeLoadMore(lastPage: false, at: 0)
sut.simulateLoadMoreFeedAction()
XCTAssertEqual(loader.loadMoreCallCount, 2, "Expected request after load more completed
with more pages")

loader.completeLoadMoreWithError(at: 1)
sut.simulateLoadMoreFeedAction()
XCTAssertEqual(loader.loadMoreCallCount, 3, "Expected request after load more failure
")

loader.completeLoadMore(lastPage: true, at: 2)
sut.simulateLoadMoreFeedAction()
XCTAssertEqual(loader.loadMoreCallCount, 3, "Expected no request after loading all
pages")
```

While creating the method helpers to carry out this test, we realize that our Paginated Combine helper has a *closure-to-publisher* converter, but it also needs a *publisher-to-closure* one, for this, we add another init, that takes in a publisher and converts it into a closure, to pass it as argument to the Paginated type, so we need a bridge/mapping.

So what we need to do is, call the publisher we are passing, subscribe to it using the **Subscribers.Sink** , receiving the completion (meaning the event ended streaming) and the received value (the success case). Usually, the completion means that it has either finished the iteration without success (but not necessarily with failure), or with failure, so we check for failure too. The finished mapping results:

```
init(items: [Item], loadMorePublisher: (() -> AnyPublisher<Self, Error>)?) {
        self.init(items: items, loadMore: loadMorePublisher.map { publisher in
            return { completion in
                publisher().subscribe(Subscribers.Sink(receiveCompletion: { result in
                    if case let .failure(error) = result {
                        completion(.failure(error))
                    }
                }, receiveValue: { result in
                    completion(.success(result))
                }))
            }
        })
    }
```

and our finished helper, that converts from closure to publisher and inits from publisher to closure is:

```
public extension Paginated {
    init(items: [Item], loadMorePublisher: (() -> AnyPublisher<Self, Error>)?) {
        self.init(items: items, loadMore: loadMorePublisher.map { publisher in
            return { completion in
```

```
                publisher().subscribe(Subscribers.Sink(receiveCompletion: { result in
                    if case let .failure(error) = result {
                        completion(.failure(error))
                    }
                }, receiveValue: { result in
                    completion(.success(result))
                }))
            }
        })
    }
    var loadMorePublisher: (() -> AnyPublisher<Self, Error>)? {
        guard let loadMore = loadMore else { return nil }

        return {
            Deferred {
                Future(loadMore)
            }.eraseToAnyPublisher()
        }
    }
}
```

By using the `Subscribers.Sink` combine automatically manages the lifecycle and we dont need a Cancellable for the subscription, the subscription will be alive until it completes.

Next goal is to test our load more spinner indicator, it needs to spin while we are loading more, so we add some tests:

```
func test_loadingMoreIndicator_isVisibleWhileLoadingMore() {
    let (sut, loader) = makeSUT()

    sut.loadViewIfNeeded()
    XCTAssertFalse(sut.isShowingLoadMoreFeedIndicator, "Expected no loading indicator
once view is loaded")

    loader.completeFeedLoading(at: 0)
    XCTAssertFalse(sut.isShowingLoadMoreFeedIndicator, "Expected no loading indicator
once loading completes successfully")

    sut.simulateLoadMoreFeedAction()
    XCTAssertTrue(sut.isShowingLoadMoreFeedIndicator, "Expected loading indicator on
load more action")

    loader.completeLoadMore(at: 0)
    XCTAssertFalse(sut.isShowingLoadMoreFeedIndicator, "Expected no loading indicator
once user initiated loading completes successfully")

    sut.simulateLoadMoreFeedAction()
```

```
    XCTAssertTrue(sut.isShowingLoadMoreFeedIndicator, "Expected loading indicator on
second load more action")

    loader.completeLoadMoreWithError(at: 1)
    XCTAssertFalse(sut.isShowingLoadMoreFeedIndicator, "Expected no loading indicator
once user initiated loading completes with error")
}
```

Following the same procedure, we test for error messages on success and on error. Next step is to test if, on error, when we tap the **LoadMoreCell** , it loads more. Same as always we start with tests and its helpers

```
func test_tapOnLoadMoreErrorView_loadsMore() {
    let (sut, loader) = makeSUT()
    sut.loadViewIfNeeded()
    loader.completeFeedLoading()

    sut.simulateLoadMoreFeedAction()
    XCTAssertEqual(loader.loadMoreCallCount, 1)

    sut.simulateTapOnLoadMoreFeedError()
    XCTAssertEqual(loader.loadMoreCallCount, 1)

    loader.completeLoadMoreWithError()
    sut.simulateTapOnLoadMoreFeedError()
    XCTAssertEqual(loader.loadMoreCallCount, 2)
}
```

This test helps us to create necessary code in the **LoadMoreCellController** , the method 'didSelectCellForRowAt' wasnt implemented. This implementation will have the same behaviour as 'willDisplayCellForRowAt', it will execute the passed closure that executes the load more action.

With this, we finish our UI. Now we move to the API ☁️ Pagination.

API ☁️ , Pagination

Now we have to implement the logic to properly handle the API pagination.

- [ ] Load 10 items at the time using Keyset Pagination
  - [ ] First: GET /feed?limit=10
  - [ ] Load More: GET /feed?limit=10&after_id={last_id}

Right now, we are loading the whole dataset, whch is a bunch of images, the idea is to limit it using the pagination.

We start with a test, testing that the `FeedEndpoint.get.url(baseURL:)` works as intended, by checking if the query of the created URL matches the query parameters we need regarding the limits:

```
XCTAssertEqual(received.query, "limit=10", "query")
```

We can also take this opportunity to add a couple of more tests for better feedback:

```
XCTAssertEqual(received.scheme, "http", "scheme")
XCTAssertEqual(received.host, "base-url.com", "host")
XCTAssertEqual(received.path, "v1/feed", "path")
XCTAssertEqual(received.query, "limit=10", "query")
```

This way we make sure that the **scheme**, **host**, **path** and **query** are all correct

Rest assuredly, our test fails, because we have not implemented the **query** ,now we must make it pass.

Right now our FeedEndpoint.get is returning a non-optional URL, we could return an optional, but that would break clients, the only way this could fail would be an programming error, since the baseURL is a parameter provided by the programmer, so, since this enum and its get method are merely programmer helpers, meaning they won't be used for a lot of users, then a failure would be the programmers fault, therefore it should fail as soon as possible to be easy to find the error. Now , on the other hand if we were writing an API or a public library , we would need to return an optional because we are not responsible for what the consumer of our api/framework is doing.

But for this case, force unwrapping the url is okay. Our final new get method is:

```
public enum FeedEndpoint {
    case get

    public func url(baseURL: URL) -> URL {
        switch self {
        case .get:
            let url = baseURL.appendingPathComponent("/v1/feed")
            var components = URLComponents()
            components.scheme = baseURL.scheme
            components.host = baseURL.host
            components.path = baseURL.path + "/v1/feed"
            components.queryItems = [
                URLQueryItem(name: "limit", value: "10")
            ]
            return components.url!
        }
    }
}
```

Note: to force unwrap the url, we have to be 100% certain that the baseURL we are providing is not the result of, for example, an API request, and its 100% a string that we have provided, otherwise this might fail and the app would break.

Next step is to make it work with the "after_id", which means we need to pass that id. For this, we are going to modify our already existent get case to give it an associated value parameter, that will be an optional FeedImage to which we will give a default nil value.

```
public enum FeedEndpoint {
    case get(after: FeedImage? = nil)
    ...
    ...
}
```

This change will break our clients, (this is a breaking change), we could also have created a new case that didnt make this a necessity, but it wouldn't make sense becausewe would just be duplicating code, and also it would mean that clients would have to switch over this enum, which is something that is best avoided if possible. If we dont want to use enums and break clients, we can also use structs with static factory methods, but for this case, only the composition root uses the endpoint, so we are going to keep the enum.

As usual, we create a new test, this time with a mock image to test that the query is correct,

```
func test_feed_endpointURLAfterGivenImage() {
    let image = uniqueImage()
    let baseURL = URL(string: "http://base-url.com")!

    let received = FeedEndpoint.get(after: image).url(baseURL: baseURL)
    let expected = URL(string: "http://base-url.com/v1/feed")!

    XCTAssertEqual(received.scheme, "http", "scheme")
    XCTAssertEqual(received.host, "base-url.com", "host")
    XCTAssertEqual(received.path, "/v1/feed", "path")
    XCTAssertEqual(received.query, "limit=10&after_id=\(image.id)", "query")
}
```

Now we refactor the FeedEnpoint so that it takes into account the image's id:

```
public enum FeedEndpoint {
    case get(after: FeedImage? = nil)

    public func url(baseURL: URL) -> URL {
        switch self {
        case let .get(image):
            let url = baseURL.appendingPathComponent("/v1/feed")
            var components = URLComponents()
            components.scheme = baseURL.scheme
            components.host = baseURL.host
            components.path = baseURL.path + "/v1/feed"
            components.queryItems = [
                URLQueryItem(name: "limit", value: "10"),
                image.map { URLQueryItem(name: "after_id", value: $0.id.uuidString) }
            ].compactMap { $0 }

            return components.url!
        }
    }
```

```
}
```

Now, for the most part, order is important, since in the URL path the order matters, but with the query items, order is ir relevant, since the limit=10 and the after_id={image.id} may come in different orders, for this we must change our tests to test accordingly to check the url query parameters without order, otherwise it may be problematic when refactoring, or if the backend returns values with the query parameters out of order:

```
XCTAssertEqual(received.query?.contains("limit=10"), true, "limit query param")
XCTAssertEqual(received.query?.contains("&after_id=\(image.id)"), true, "after id query param")
```

With this, we are done with the API ☁️ . Now we have to move to **Composition**

Composition

- [ ] Paginate loading feed
- [ ] Cache all loaded pages for offline support

Now that we've finished with the API, its time to move to composition, for this, we will create some acceptance tests.

By doing these tests, with the desired behaviour, we see that we have not passed a **loadMore** closure in our Composition Root (SceneDelegate) when we initialize our Paginated, so we pass the loadMorePublisher, also what we have to do is to get the corresponding url to query the next items, this url will be composed with the last feedimage's id, to be able to create a "after_id" url.

First we'll remember the current state of our makeRemoteFeedLoaderWithLocalFallback:

```
  private func makeRemoteFeedLoaderWithLocalFallback() ->
AnyPublisher<Paginated<FeedImage>, Error> {
        let url = FeedEndpoint.get().url(baseURL: baseURL)

        return httpClient
            .getPublisher(url: url)
            .tryMap(FeedItemsMapper.map)
            .caching(to: localFeedLoader)
            .fallback(to: localFeedLoader.loadPublisher)
            .map {
                Paginated(items: $0)
            }
            .eraseToAnyPublisher()
    }
```

as we can see, we are not passing in the loadMore publisher that is needed, the first approach to fix this is to do:

```
private func makeRemoteFeedLoaderWithLocalFallback() ->
    AnyPublisher<Paginated<FeedImage>, Error> {
    let url = FeedEndpoint.get().url(baseURL: baseURL)

    return httpClient
        .getPublisher(url: url)
        .tryMap(FeedItemsMapper.map)
        .caching(to: localFeedLoader)
        .fallback(to: localFeedLoader.loadPublisher)
        .map { [httpClient, baseURL] items in
            Paginated(items: items, loadMorePublisher: items.last.map { lastItem in
                let url = FeedEndpoint.get(after: lastItem).url(baseURL: baseURL)

                return { [httpClient] in
                    httpClient
                        .getPublisher(url: url)
                        .tryMap(FeedItemsMapper.map)
                        .map { newItems in
                            Paginated(items: items + newItems, loadMorePublisher: {
                                Empty().eraseToAnyPublisher()
                            })
                        }.eraseToAnyPublisher()
                }
            })
        }
```

But unfortunately this will only work once and we can see that the code is getting messy and wont scale well, which we can confirm this by more acceptance tests, this means that we need a new approach.

Instead of nesting the operation, we have to use recursion. For this we will create a new method `makeRemoteLoader(items: [FeedImage]) -> (() -> AnyPublisher<Paginated<FeedImage>,Error>)?` and start moving previous logic inside it:

```
private func makeRemoteLoadMoreLoader(items: [FeedImage]) -> (() ->
    AnyPublisher<Paginated<FeedImage>, Error>)? {
    items.last.map { lastItem in
        let url = FeedEndpoint.get(after: lastItem).url(baseURL: baseURL)

        return { [httpClient] in
            httpClient
                .getPublisher(url: url)
                .tryMap(FeedItemsMapper.map)
                .map { newItems in
                    Paginated(items: items + newItems, loadMorePublisher: {
                        Empty().eraseToAnyPublisher()
                    })
                }.eraseToAnyPublisher()
        }
    }
}
```

and replace the code in the previous method:

```
private func makeRemoteFeedLoaderWithLocalFallback() ->
    AnyPublisher<Paginated<FeedImage>, Error> {
    let url = FeedEndpoint.get().url(baseURL: baseURL)

    return httpClient
        .getPublisher(url: url)
        .tryMap(FeedItemsMapper.map)
        .caching(to: localFeedLoader)
        .fallback(to: localFeedLoader.loadPublisher)
        .map { items in
            Paginated(items: items, loadMorePublisher:
                self.makeRemoteLoadMoreLoader(items: items))
        }
        .eraseToAnyPublisher()
}
```

But this is not its final form yet, we need to call **makeRemoteLoader** recursively:

```
private func makeRemoteLoadMoreLoader(items: [FeedImage]) -> (() ->
    AnyPublisher<Paginated<FeedImage>, Error>)? {
    items.last.map { lastItem in
        let url = FeedEndpoint.get(after: lastItem).url(baseURL: baseURL)

        return { [httpClient] in
            httpClient
                .getPublisher(url: url)
                .tryMap(FeedItemsMapper.map)
                .map { newItems in
                    let allItems = items + newItems
                    return Paginated(items: allItems, loadMorePublisher:
                        self.makeRemoteLoadMoreLoader(items: allItems))
                }.eraseToAnyPublisher()
        }
    }
}
```

Of course it is still not passing, because we need a **last** item for each case:

```
private func makeRemoteLoadMoreLoader(items: [FeedImage], last: FeedImage?) -> (() ->
    AnyPublisher<Paginated<FeedImage>, Error>)? {
    last.map { lastItem in
        let url = FeedEndpoint.get(after: lastItem).url(baseURL: baseURL)

        return { [httpClient] in
            httpClient
                .getPublisher(url: url)
                .tryMap(FeedItemsMapper.map)
                .map { newItems in
                    let allItems = items + newItems
                    return Paginated(items: allItems, loadMorePublisher:
                        self.makeRemoteLoadMoreLoader(items: allItems, last:
                        newItems.last))
                }.eraseToAnyPublisher()
        }
    }
}
```

When **newItems** is empty it means we've reached the end. We keep appending until the last item is empty, which means that we've reached the end of pagination.

```
private func makeRemoteFeedLoaderWithLocalFallback() ->
    AnyPublisher<Paginated<FeedImage>, Error> {
    let url = FeedEndpoint.get().url(baseURL: baseURL)

    return httpClient
        .getPublisher(url: url)
        .tryMap(FeedItemsMapper.map)
        .caching(to: localFeedLoader)
        .fallback(to: localFeedLoader.loadPublisher)
        .map { items in
            Paginated(items: items, loadMorePublisher:
                self.makeRemoteLoadMoreLoader(items: items, last: items.last))
        }
        .eraseToAnyPublisher()
}
```

Now the code works correctly.

So what this code does is it calls the makeRemoteLoadMoreLoader until we have no more items, to find the last, and then it returns that publisher and passes as the `loadMorePublisher` to the **Paginated** init.

Code as we see it is still messy, and we will refactor it, but first we will make sure the caching works properly.

Caching the paginated response

For this we will want to test that, we start with an online status and fetch the data, then when we are offline we still see the data, in the same fashion we did before with this test:

```
func test_onLaunch_displaysCachedRemoteFeedWhenCustomerHasNoConnectivity() {
	let sharedStore = InMemoryFeedStore.empty
	let onlineFeed = launch(httpClient: .online(response), store: sharedStore)
	onlineFeed.simulateFeedImageViewVisible(at: 0)
	onlineFeed.simulateFeedImageViewVisible(at: 1)

	let offlineFeed = launch(httpClient: .offline, store: sharedStore)

	XCTAssertEqual(offlineFeed.numberOfRenderedFeedImageViews(), 2)
	XCTAssertEqual(offlineFeed.renderedFeedImageData(at: 0), makeImageData0())
	XCTAssertEqual(offlineFeed.renderedFeedImageData(at: 1), makeImageData1())
}
```

So, if our system was working properly, we would be able to just add one more item during the online phase, and be able to retrieve it offline:

```
func test_onLaunch_displaysCachedRemoteFeedWhenCustomerHasNoConnectivity() {
	let sharedStore = InMemoryFeedStore.empty

	let onlineFeed = launch(httpClient: .online(response), store: sharedStore)
	onlineFeed.simulateFeedImageViewVisible(at: 0)
	onlineFeed.simulateFeedImageViewVisible(at: 1)
	onlineFeed.simulateLoadMoreFeedAction()
	onlineFeed.simulateFeedImageViewVisible(at: 2)

	let offlineFeed = launch(httpClient: .offline, store: sharedStore)

	XCTAssertEqual(offlineFeed.numberOfRenderedFeedImageViews(), 3)
	XCTAssertEqual(offlineFeed.renderedFeedImageData(at: 0), makeImageData0())
	XCTAssertEqual(offlineFeed.renderedFeedImageData(at: 1), makeImageData1())
	XCTAssertEqual(offlineFeed.renderedFeedImageData(at: 2), makeImageData2())
}
```

but its failing.

Fixing this is easy, we go to the composition and in the method we just created `makeRemoteMoreFeedLoader` we add a caching command just before the end of the pipeline, for this we have to create a new caching function for the paginated case inside the combine helpers:

```
extension Publisher {
    func caching(to cache: FeedCache) -> AnyPublisher<Output, Failure> where Output ==
[FeedImage] {
        handleEvents(receiveOutput: cache.saveIgnoringResult).eraseToAnyPublisher()
    }

    func caching(to cache: FeedCache) -> AnyPublisher<Output, Failure> where Output ==
Paginated<FeedImage> {
        handleEvents(receiveOutput: cache.saveIgnoringResult).eraseToAnyPublisher()
    }
}
```

This way we can handle paginating both cases with pagination and with managing the whole feed. We also add a helper:

```
private extension FeedCache {
    func saveIgnoringResult(_ feed: [FeedImage]) {
        save(feed) { _ in }
    }

    func saveIgnoringResult(_ page: Paginated<FeedImage>) {
        saveIgnoringResult(page.items)
    }
}
```

Back to our Composition Root we have:

```
private func makeRemoteLoadMoreLoader(items: [FeedImage], last: FeedImage?) -> (() ->
AnyPublisher<Paginated<FeedImage>, Error>)? {
    last.map { lastItem in
        let url = FeedEndpoint.get(after: lastItem).url(baseURL: baseURL)

        return { [httpClient, localFeedLoader] in
            httpClient
                .getPublisher(url: url)
                .tryMap(FeedItemsMapper.map)
                .map { newItems in
                    let allItems = items + newItems
                    return Paginated(items: allItems, loadMorePublisher:
self.makeRemoteLoadMoreLoader(items: allItems, last: newItems.last))
                }
                .caching(to: localFeedLoader)
        }
    }
}
```

Now, our tests are passing and the pages are being cached. Our cache doesnt need to know anything about pagination, since it's an api detail, this is why we didnt have to make any changes to the cache logic.

Refactoring

Now that everthing is working, both cache and remote we will refactor the `makeRemoteLoadMoreLoader` and the `makeRemoteFeedLoaderWithLocalFallback` methods.

First thing we will do is create a private helper method and extract all the map logic:

```
private func makePage(items: [FeedImage], last: FeedImage?) -> Paginated<FeedImage> {
      Paginated(items: items, loadMorePublisher: last.map { last in
          { self.makeRemoteLoadMoreLoader(items: items, last: last) }
      })
}
```

Then we change our `makeRemoteFeedLoaderWithFallback` to :

```
private func makeRemoteFeedLoaderWithLocalFallback() ->
AnyPublisher<Paginated<FeedImage>, Error> {
    makeRemoteFeedLoader()
        .caching(to: localFeedLoader)
        .fallback(to: localFeedLoader.loadPublisher)
        .map(makeFirstPage)
        .eraseToAnyPublisher()
}
```

Now we will modify our `makeRemoteLoadMoreLoader` helper by creating a "createPage" method:

```
private func makePage(items: [FeedImage], last: FeedImage?) -> Paginated<FeedImage> {
      Paginated(items: items, loadMorePublisher: last.map { last in
          { self.makeRemoteLoadMoreLoader(items: items, last: last) }
      })
}
```

and now:

```
private func makeRemoteLoadMoreLoader(items: [FeedImage], last: FeedImage?) -> (() ->
    AnyPublisher<Paginated<FeedImage>, Error>)? {
    last.map { lastItem in
        let url = FeedEndpoint.get(after: lastItem).url(baseURL: baseURL)

        return { [httpClient, localFeedLoader, makePage] in
            httpClient
                .getPublisher(url: url)
                .tryMap(FeedItemsMapper.map)
                .map { newItems in
                    (items + newItems, newItems.last)
                }.map(makePage)
                .caching(to: localFeedLoader)
        }
    }
```

but we can make it even cleaner, we can make `makeFirstPage` call our `makePage` providing a last item:

```
private func makeFirstPage(items: [FeedImage]) -> Paginated<FeedImage> {
    makePage(items: items, last: items.last)
}
```

Now, to avoid having a lot of code we will create a new function `makeRemoteFeedLoader` to which we will extract the logic inside **makeRemoteLoadMoreLoader** :

```
 private func makeRemoteFeedLoader(after: FeedImage? = nil) ->
AnyPublisher<[FeedImage], Error> {
          let url = FeedEndpoint.get(after: after).url(baseURL: baseURL)

          return httpClient
              .getPublisher(url: url)
              .tryMap(FeedItemsMapper.map)
              .eraseToAnyPublisher()
}
```

and then replacing in our `makeRemoteLoadMoreLoader`:

```
private func makeRemoteLoadMoreLoader(items: [FeedImage], last: FeedImage?) ->
AnyPublisher<Paginated<FeedImage>, Error> {
      makeRemoteFeedLoader(after: last)
          .map { newItems in
              (items + newItems, newItems.last)
          }.map(makePage)
          .caching(to: localFeedLoader)
}
```

Which significantly reduces the amount of lines and readability of the code and avoid nested closures, which we dont want.

CHECK ALL OF THIS CODE WHICH IS HARD, VIDEO 59, LAST 40 MINUTES.

If we wanted to simulate an error we could do:

```
 private func makeRemoteLoadMoreLoader(items: [FeedImage], last: FeedImage?) ->
AnyPublisher<Paginated<FeedImage>, Error> {
        makeRemoteFeedLoader(after: last)
            .map { newItems in
                (items + newItems, newItems.last)
            }.map(makePage)
            .delay(for: 2, scheduler: DispatchQueue.main)
            .flatMap { _ in
                Fail(error: NSError())
            }
            .caching(to: localFeedLoader)
}
```

And after 2 seconds we can see the error message on the **LoadMoreCell**. When we tap the cell we see it has a selection style, so we will change that now in **LoadMoreCellController** in the `cellForRowAt` method:

```
public func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    cell.selectionStyle = .none
    return cell
}
```

and now it works and displays as expected.

Next thing we want is to automatically load more items on scroll after error

Load More items after Load More error when the user scrolls

So, when the user encounters an error, it's pretty common to keep scrolling imitating in a way the pull to refresh, so the idea is that on error, on a scroll it should try to load more on error. Right now what happens is that until the **LoadMoreCellController** cell is removed from the hierarchy, and reappears this wont happen, because we are using **willDisplayCell** to trigger this reload, so we have to change that.

What we can do to change this is use **Key-Value-Observers (KVO)**.

In the UITableViewDelegate method `willDisplayCell`, we will add:

```
 public func tableView(_ tableView: UITableView, willDisplay: UITableViewCell, forRowAt
indexPath: IndexPath) {
    reloadIfNeeded()

    offsetObserver = tableView.observe(\.contentOffset, options: .new) { [weak self]
tableView, _ in
        guard tableView.isDragging else { return }

        self?.reloadIfNeeded()
    }
}

public func tableView(_ tableView: UITableView, didEndDisplaying cell: UITableViewCell,
forRowAt indexPath: IndexPath) {
       offsetObserver = nil
}
```

So what we are doing is, we add an observer to the **contentOffset** key, and we check if the cell is being dragged, if it is, and it is not loading, we will trigger a reload (thats why we use a reloadIfNeeded), once the cell is about to go out of sight, in the `didEndDisplayingCell`, we turn the observer off, by setting it to nil.

If we wanted to track the threshold not only when the view is visible or not we could move the observation somewhere else, like in the `didScroll` method and maybe prefetch before the user reaches the **LoadMoreCellController**,

Fetch current items from cache when needed instead of keeping them in memory all the time

Right now we are keeping all the items in memory, and we keep appending into the items, like we see below:

```
private func makeRemoteLoadMoreLoader(items: [FeedImage], last: FeedImage?) ->
AnyPublisher<Paginated<FeedImage>, Error> {
    makeRemoteFeedLoader(after: last)
        .map { newItems in
            (items + newItems, newItems.last)
        }.map(makePage)
        .caching(to: localFeedLoader)
}
```

but what if I dont want to keep everything in memory as I load the pages?. We can prevent this by loading the current items from the cache when needed only. So for this, we will remove the **items** property from the **makeRemoteLoadMoreLoader** method argument, and instead load the items from the cache:

```
private func makeRemoteLoadMoreLoader(last: FeedImage?) ->
AnyPublisher<Paginated<FeedImage>, Error> {
    makeRemoteFeedLoader(after: last)
        .flatMap { [localFeedLoader] newItems in
            localFeedLoader.loadPublisher().map { cachedItems in
                (newItems, cachedItems)
            }
        }
        .map { (newItems, cachedItems) in
            (cachedItems + newItems, newItems.last)
        }.map(makePage)
        .caching(to: localFeedLoader)
}
```

So now, instead of keeping the items in memory we just request them from the cache.

By the way, using **flatMap** to combine two results into a tuple is the same thing as using a **zip** publisher, the result is exactly the same:

```
private func makeRemoteLoadMoreLoader(last: FeedImage?) ->
AnyPublisher<Paginated<FeedImage>, Error> {
    makeRemoteFeedLoader(after: last)
        .zip(localFeedLoader.loadPublisher())
        .map { (newItems, cachedItems) in
            (cachedItems + newItems, newItems.last)
        }.map(makePage)
        .caching(to: localFeedLoader)
}
```

We can also invert the order of the chain:

```
private func makeRemoteLoadMoreLoader(last: FeedImage?) ->
AnyPublisher<Paginated<FeedImage>, Error> {
    localFeedLoader.loadPublisher()
        .zip(makeRemoteFeedLoader(after: last))
        .map { (cachedItems, newItems) in
            (cachedItems + newItems, newItems.last)
        }.map(makePage)
        .caching(to: localFeedLoader)
}
```

and it still works.

Doing this or not depends on whether you want to keep the items in memory or not, keeping them in memory will be quicker, but it all depends on the results and on what you want/need. (if we are working with thousands of items it may not be the best to keep them in memory, and better to just load them when eneded)

This pagination with caching we developed are independant/ decoupled from the pagination method/algorithm, so we can change it from keyset to offset to whaterver without breaking other modules.

Logging/ Profiling/ Optimizing infrastructure Services

Live #006

- Logging and Profiling as Cross-Cutting Concerns
- Monitoring Debug and Release builds
- Null object pattern
- Optimizing data consumption

Logging

The idea is to do Logging in a clean way, without adding print statements everywhere.

"Log messages provide a continuous record of your app's runtime behaviour, and make it easier to identify problems that can't be caught easily using other techniques" - Apple Docs.

Common use cases

- Diagnosing problems without a debugger or when it's difficult to catch in the debugger (for example when the app is in production you cant debug it, or test it, but you can read the log messages)
- Tracing your app's behavior (e.g., when certain tasks start/end)
- Auditing
- Performance monitoring
- Recording unhandled Errors/Exceptions

Common Challenges

From Jeff Atwood (the co-creator of Stack Overflow)

- **Logging means more code** , which obscures your application code.
- **Logging isn't free**, and logging a lot means constantly writing to disk.
- **The more you log**, the less you can find.
- **If it's worth saving to a log file, it's worth showing in the user interface.** Many apps have use cases that do not handle the errors, and just print the unhandled error to a log on the console, without even showing the user what happened, this is terrible, because its like having a silent error that we can't track properly. Sometimes it is better that the app crashes altogether instead of **silently** failing,

since at least a crash sends us a crash report and we can see what happened, whereas an unhandled error printed to a log wont help us find and solve the problem, and the user will feel like the app is not working as intended.

Taking a look in our project we can see that in the SceneDelegate, when we are instantiating the CoreData `store` property we are doing a **try!** to instanciate it, but it could fail. This could happen due to many things, such as no more space left, a faulty migration by the developer, or some other developing error. Many programmers in this situation would choose not to have a throwing function in order for the app not to crash, and just ignore the errors or print them. It's also very commo to see a lot of code full of guards and returns that dont handle the errors.

Safe code isnt necessarily about the app never crashing, safe code doesn't allow the system to get into a weird state, if the system gets into a weird state, the safe thing to do is crash, because then we preserve the identity of the system.

We don't want to mask the errors with print statements!.

In our app, the CoreData store is not *critical*, since if it failed, our app would still work with internet, since the core data store is a fallback. In these cases, when a component is not crucial, we can replace a faulty component with another at runtime, for example we could replace a faulty coredata instance with our in-memory store (which would have to be made thread safe if we were to use it). Another strategy is to use the **Null-Object Pattern**.

Null Object Pattern

"A null object is an object with no referenced value or with a defined natural ("null") behaviour"

For example, we can create a NullStore that implements the protocols it needs to, and it will provide a default neutral behaviour, a behaviour that will not leave your system in a weird state but will do the minimum possible. For example

```
class NullStore: FeedStore & FeedImageDataStore {}
```

Which will implement all the methods needed by the protocol conformance:

```
class NullStore: FeedStore & FeedImageDataStore {
    func deleteCachedFeed(completion: @escaping DeletionCompletion) {

    }

    func insert(_ feed: [FeedFramework.LocalFeedImage], timestamp: Date, completion:
@escaping InsertionCompletion) {

    }
```

```
    func retrieve(completion: @escaping RetrievalCompletion) {

    }

    func insert(_ data: Data, for url: URL, completion: @escaping (InsertionResult) ->
Void) {

    }

    func retrieve(dataForURL url: URL, completion: @escaping (RetrievalResult) -> Void)
{

    }
}
```

So, what is the neutral behaviour for a **deleteCachedFeed**? It needs to at least complete the operation so the clients wont be hanging forever, to which we should get a .success(), because yes, there is no cache, but it didnt fail.

When **insert** ?, the same, we just complete with .success().

When **retrieve**? completion with .success(.none), it couldnt retrieve anything, but the operation completes successfully.

This **Neutral** behaviour depends from case to case, in our case we just need it to complete with success, so as not to break the flow of the app. For example if we have a method that does not take arguments and returns a void, we can simply do nothing.

Here is our finished **NullStore**:

```
class NullStore: FeedStore & FeedImageDataStore {
    func deleteCachedFeed(completion: @escaping DeletionCompletion) {
        completion(.success(()))
    }

    func insert(_ feed: [FeedFramework.LocalFeedImage], timestamp: Date, completion:
@escaping InsertionCompletion) {
        completion(.success(()))
    }

    func retrieve(completion: @escaping RetrievalCompletion) {
        completion(.success(.none))
    }

    func insert(_ data: Data, for url: URL, completion: @escaping (InsertionResult) ->
Void) {
        completion(.success(()))
    }
```

```
    func retrieve(dataForURL url: URL, completion: @escaping
(FeedImageDataStore.RetrievalResult) -> Void) {
        completion(.success(.none))
    }
}
```

Now we can change the declaration of our **store** in the SceneDelegate, not to force try, but with a Do-Catch, and in case it fails, we return a NullStore:

```
 private lazy var store: FeedStore & FeedImageDataStore = {
        do {
            return try CoreDataFeedStore(
                storeURL: NSPersistentContainer
                    .defaultDirectoryURL()
                    .appendingPathComponent("feed-store.sqlite"))
        } catch {
            return NullStore()
        }
    }()
```

If for example a migration fails on some devices, this way the app wont crash. But truly what we want now is to be notified when these problems happen, usually, we could add a **print** statement in the catch of the previous instantiation. But the more **log** messages we see on the console the less attention we will pay to them. But this is not nice, and we can't use it, before this at least if the store failed, the app would crash, this way the error gets buried in prints, so we want it to at least be able to crash on **debug** builds, so that when in testing, we can see it crash, so we are notified quickly of any programmer mistake.

What we can do for this is to add an **assertionFailure** to the catch block instead of a print, because **assertionFailure** will cause a crash on debug builds, but not on release builds. So the catch would look like this:

```
catch {
            assertionFailure("Failed to instantiate CoreDate store with error: \
(error.localizedDescription)")
            return NullStore()
        }
```

But what if a programmer mistake is released by mistake? For example a faulty CoreData migration that only happens on specific devices that we couldn't catch on debug?.

So, when we want to be notified of faulty behaviour in our release builds, we can use a logger library, there are many out there like for example **Firebase Crashalytics**, in our case we are going to be using the **Logger library** from **Apple's OS Framework** , simply by importing "os".

So we will now create a new property **logger: Logger** (Logger only works from iOS14+)

```
private lazy var logger = Logger(subsystem: "com.CompanyIdentifier.OurApp", category:
"main")
```

where the "subsystem" parameter makes reference to the **Bundle Identifier** set in **Signing & Capabilities** under the **Team**, and "category" is usually the module you are loading (which in this case doesnt make reference to anything in particular).

Now we can use this logger, or whatever logger library we are using to log the failure:

```
catch {
            assertionFailure("Failed to instantiate CoreDate store with error: \
(error.localizedDescription)")
            logger.fault("Failed to instantiate CoreDate store with error: \
(error.localizedDescription)")
            return NullStore()
        }
```

Now we will have the error also logged, so that if it happens in production we can check for it.

So, the main idea is :

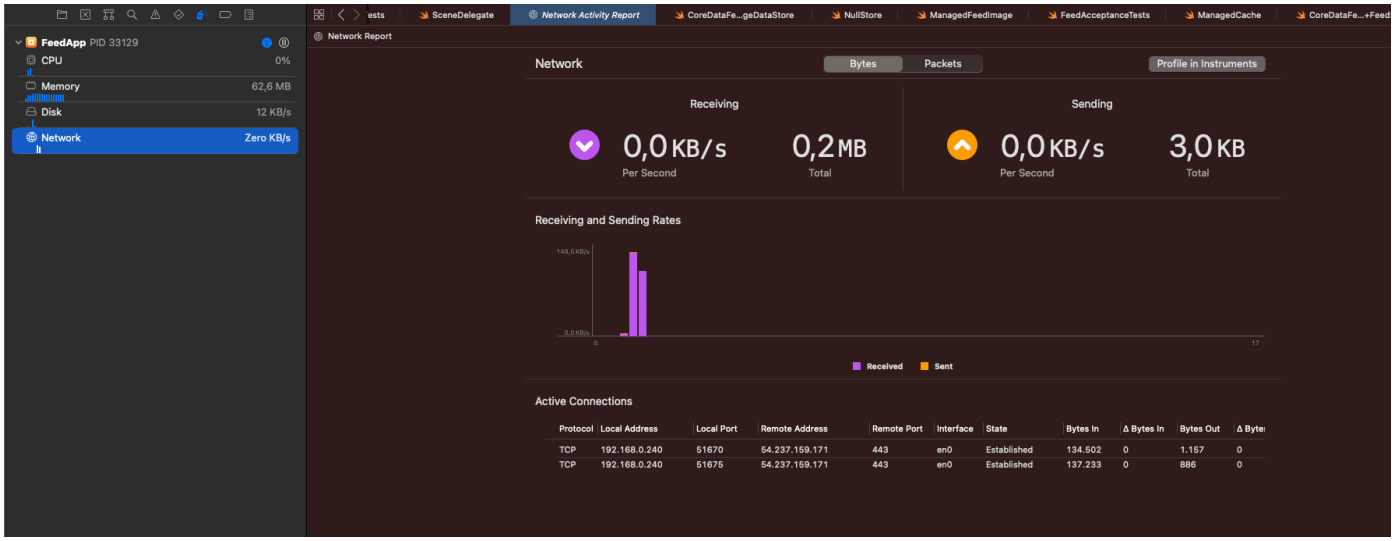
- **Don't ignore errors**
- **Don't let your app get into a weird state**
- **Provide a fallback strategy if you can** (and when you do, log the errors)
- **If the issue is really critical, it's better to crash the app than to get into a weird state where the App becomes unusable and must be reinstalled**

Another thing that is important that we can see here, is that we are adding the **log** in the error handling not within the instance that generated the error (**CoreDataFeedStore**), but from the main module in the **Composition Root**, and this is how you avoid having to inject loggers or even access global loggers from every module in your app, with a good application design you'll be able to apply logging without polluting your entire codebase. (We could of course have passed a logger to all of our objects such as **CoreDataFeedStore**, but that only pollutes the code, and its an extra dependency to which the objects don't need to know about, because they dont use it.)

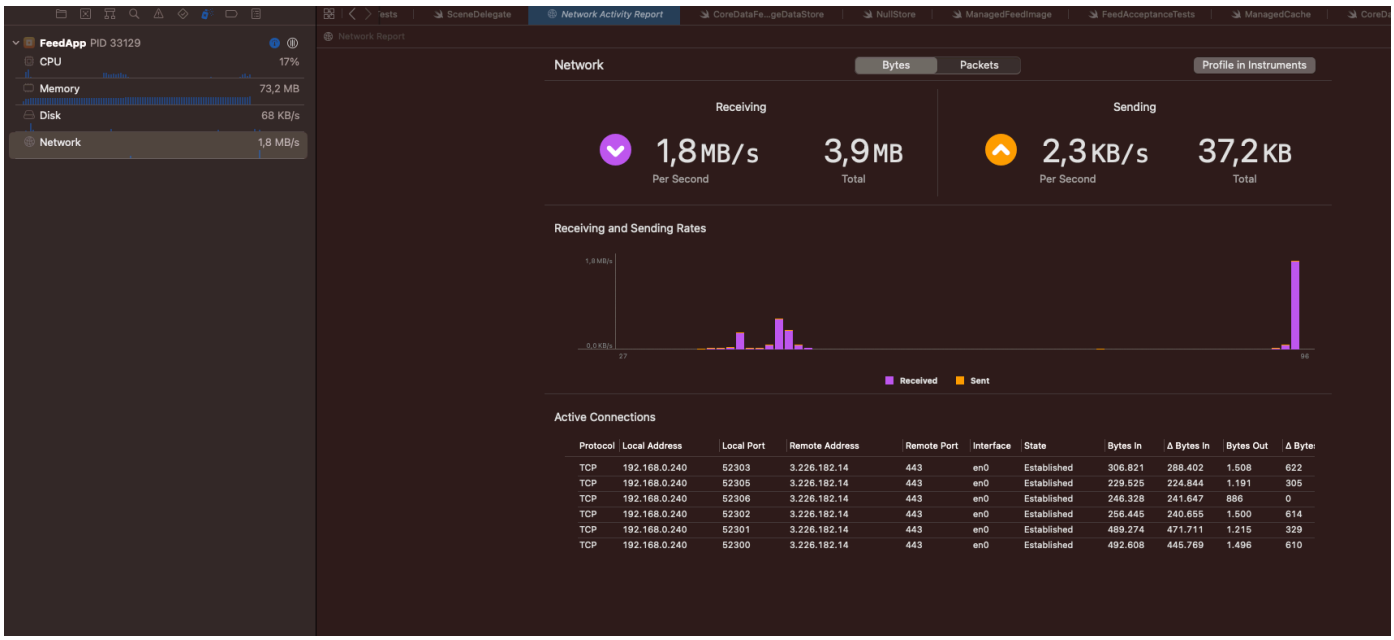
Logging is not free, it should be decided in the main module how to do it and when to do it and the more you log, the less you can find, so logs should be kept at a minimum for really critical errors and issues.

All of this said, is about tracing and monitoring errors in production, what about tracing debug logs to help find issues in debug?.

For example, lets profile the network request in our app:



We can see that as we scroll through the feed, new connections and consumptions start appearing in the network activity report:



As we pull to refresh or scroll through the feed to load more items, we can easily see how our Network Data Consumption rises, which may sometimes be quite inconvenient for users on Cellular Networks. We can see that everytime we do a pull to refresh, we are loading all the images again, so what if we want to trace those requests to help us reduce the data usage? Of course we could use **Instruments** , but what if we want to keep track of these requests, which ones were rejected, which ones completed?

We couldn't write tests since its very hard to write tests to track these metrics/issues/performance improvements/data consumption, but what we can use are **logs**, but they are usually temporary trace logs, because we dont want to leave them in the app, we dont want them in release or in debug (we dont want to have thousands of log messages in the console), and we also want to be able to trace and add these logging behaviour without polluting our modules, because logging is a cross-cutting concern that can be injected from the Composition-Root, but we don't want to inject these modules everywhere or use a singleton that can lead to race conditions or other problems.

So, how do we deal with cross-cutting concerns? -> with the Decorator Pattern, we can inject the logging, by decorating an **httpClient**:

```
private class HTTPClientProfilingDecorator: HTTPClient {
    private let decoratee: HTTPClient
    private let logger: Logger

    internal init(decoratee: HTTPClient, logger: Logger) {
        self.decoratee = decoratee
        self.logger = logger
    }

    func get(from url: URL, completion: @escaping (HTTPClient.Result) -> Void) ->
HTTPClientTask {
        logger.trace("Started loading url: \(url)")
        return decoratee.get(from: url) { [logger] result in
            logger.trace("Finished loading url: \(url)")
            completion(result)
        }
    }
}
```

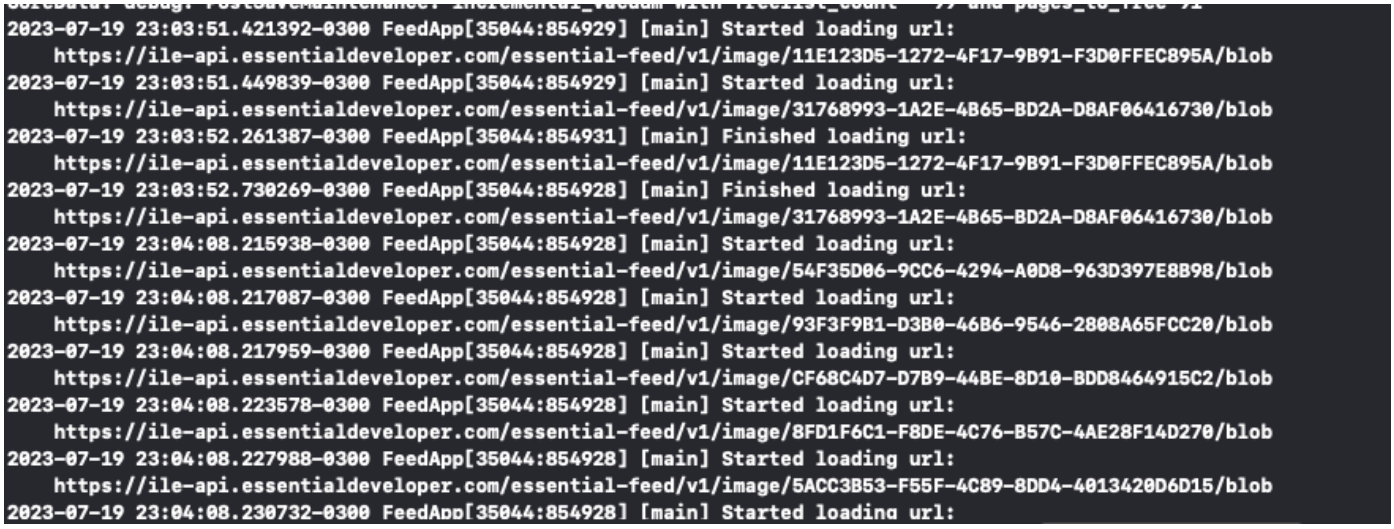
So all we do here is use a decorator, to *decorate* our HTTPClient, by adding two trace logs that log when a url started loading and when the url finished loading, and then we just simply forward the decoratee's message, this way we can easily add desired behaviour to an existing component without modifying it, satisfying the open-closed principle. Basically a Decorator is a wrapper, that wraps our existing object and adds functionality to it.

The idea is to track the loading of the images, so we will use the decorated client in the **makeLocalImageWithRemoteFallback(url:URL)** method in our SceneDelegate:

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
     let wrappedClient = HTTPClientProfilingDecorator(decoratee: httpClient, logger:
logger)

     let localImageLoader = LocalFeedImageDataLoader(store: store)

     return localImageLoader
         .loadImageDataPublisher(from: url)
         .fallback(to: {
             wrappedClient
                 .getPublisher(url: url)
                 .tryMap(FeedImageDataMapper.map)
                 .caching(to: localImageLoader, using: url)
         })
    }
}
```

Here we have replaced the already existing httpClient for the wrappedClient that has the logger.

Now we run the app, and in the console we can read:



We see that our logger is working properly, logging when the loading starts and when it finishes.

We could also log how long the request took and also add a log in case of failure:

```
private class HTTPClientProfilingDecorator: HTTPClient {
    private let decoratee: HTTPClient
    private let logger: Logger

    internal init(decoratee: HTTPClient, logger: Logger) {
        self.decoratee = decoratee
        self.logger = logger
    }

    func get(from url: URL, completion: @escaping (HTTPClient.Result) -> Void) ->
HTTPClientTask {
        logger.trace("Started loading url: \(url)")
        let startTime = CACurrentMediaTime()
        return decoratee.get(from: url) { [logger] result in
            if case let .failure(error) = result {
                logger.trace("Failer to load url: \(url), with error: \
(error.localizedDescription)")
            }
            let elapsedTime = CACurrentMediaTime() - startTime
            logger.trace("Finished loading url: \(url) in \(elapsedTime) seconds")
            completion(result)
        }
    }
}
```

Here we can see the console output of the logger:

```
2023-07-19 23:15:24.974195-0300 FeedApp[35400:864184] [main] Started loading url:
    https://ile-api.essentialdeveloper.com/essential-feed/v1/image/54F35D06-9CC6-4294-A0D8-963D397E8B98/blob
2023-07-19 23:15:25.181758-0300 FeedApp[35400:864182] [main] Started loading url:
    https://ile-api.essentialdeveloper.com/essential-feed/v1/image/93F3F9B1-D3B0-46B6-9546-2808A65FCC20/blob
2023-07-19 23:15:25.194450-0300 FeedApp[35400:864592] [main] Failer to load url:
    https://ile-api.essentialdeveloper.com/essential-feed/v1/image/93F3F9B1-D3B0-46B6-9546-2808A65FCC20/blob, with error:
    Error Domain=NSURLErrorDomain Code=-999 "cancelled"
    UserInfo={NSErrorFailingURLStringKey=https://ile-api.essentialdeveloper
    .com/essential-feed/v1/image/93F3F9B1-D3B0-46B6-9546-2808A65FCC20/blob,
    NSErrorFailingURLKey=https://ile-api.essentialdeveloper
    .com/essential-feed/v1/image/93F3F9B1-D3B0-46B6-9546-2808A65FCC20/blob, _NSURLErrorRelatedURLSessionTaskErrorKey=(
    "LocalDataTask <FB6CCE44-0597-49CA-B3FA-FE3C16A9657F>.<4>"
), _NSURLErrorFailingURLSessionTaskErrorKey=LocalDataTask <FB6CCE44-0597-49CA-B3FA-FE3C16A9657F>.<4>,
    NSLocalizedDescription=cancelled}
2023-07-19 23:15:25.218856-0300 FeedApp[35400:864592] [main] Finished loading url:
    https://ile-api.essentialdeveloper.com/essential-feed/v1/image/93F3F9B1-D3B0-46B6-9546-2808A65FCC20/blob in 0.265905
    seconds
2023-07-19 23:15:25.511943-0300 FeedApp[35400:864591] [main] Failer to load url:
    https://ile-api.essentialdeveloper.com/essential-feed/v1/image/CF68C4D7-D7B9-44BE-8D10-BDD8464915C2/blob, with error:
    Error Domain=NSURLErrorDomain Code=-999 "cancelled"
    UserInfo={NSErrorFailingURLStringKey=https://ile-api.essentialdeveloper
    .com/essential-feed/v1/image/CF68C4D7-D7B9-44BE-8D10-BDD8464915C2/blob,
    NSErrorFailingURLKey=https://ile-api.essentialdeveloper
```

we can see that we have the start logs, the finish logs, the elapsed times and the logged errors.

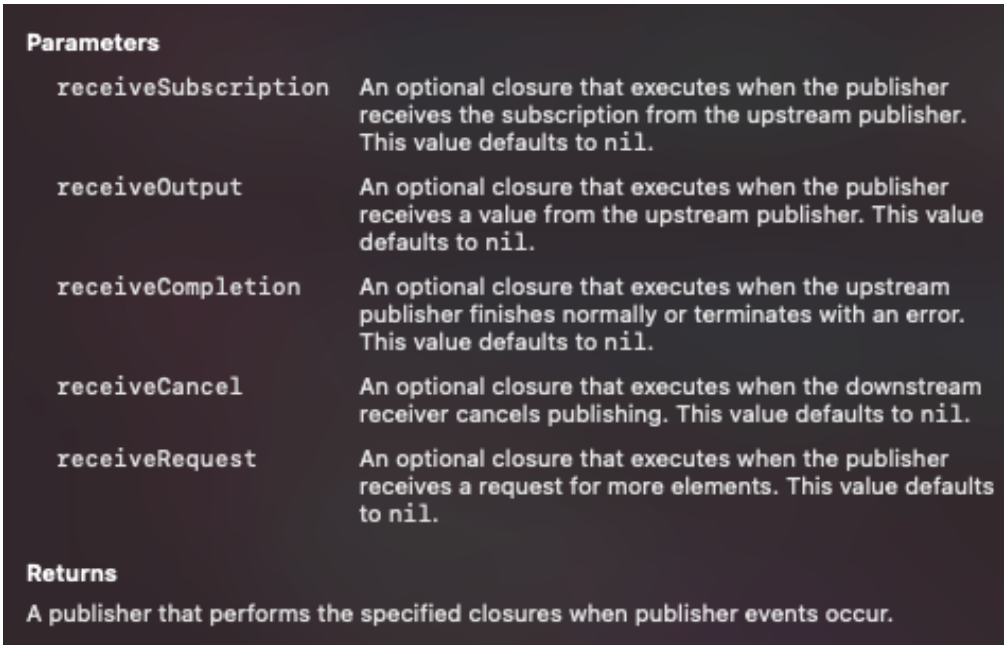
We only replaced a decorated for a decoratee for the fetch of images, but we could easily replace the main **httpClient** and log everything that goes through the client, but this could easily get out of hand, to do it universal we would do:

```
  private lazy var httpClient: HTTPClient = {
         HTTPClientProfilingDecorator(decoratee: URLSessionHTTPClient(session:
URLSession(configuration: .ephemeral)), logger: logger)
     }()
```

Wrapping our existing URLSessionHTTPCLient.

The conclusion, though, is that you do not need to pollute your entire codebase with logging, you can just log the infrastructure errors, because normally you want to log performance, and performance is usually talking to a database or to the network, or the ui, all infrastructure, and you do the infrastructure in the infrastructure layer, you dont need to pollute your business layer with a bunch of logs. No business rule should know about the logging, that is why we do all this in the composition root including the performance logging.

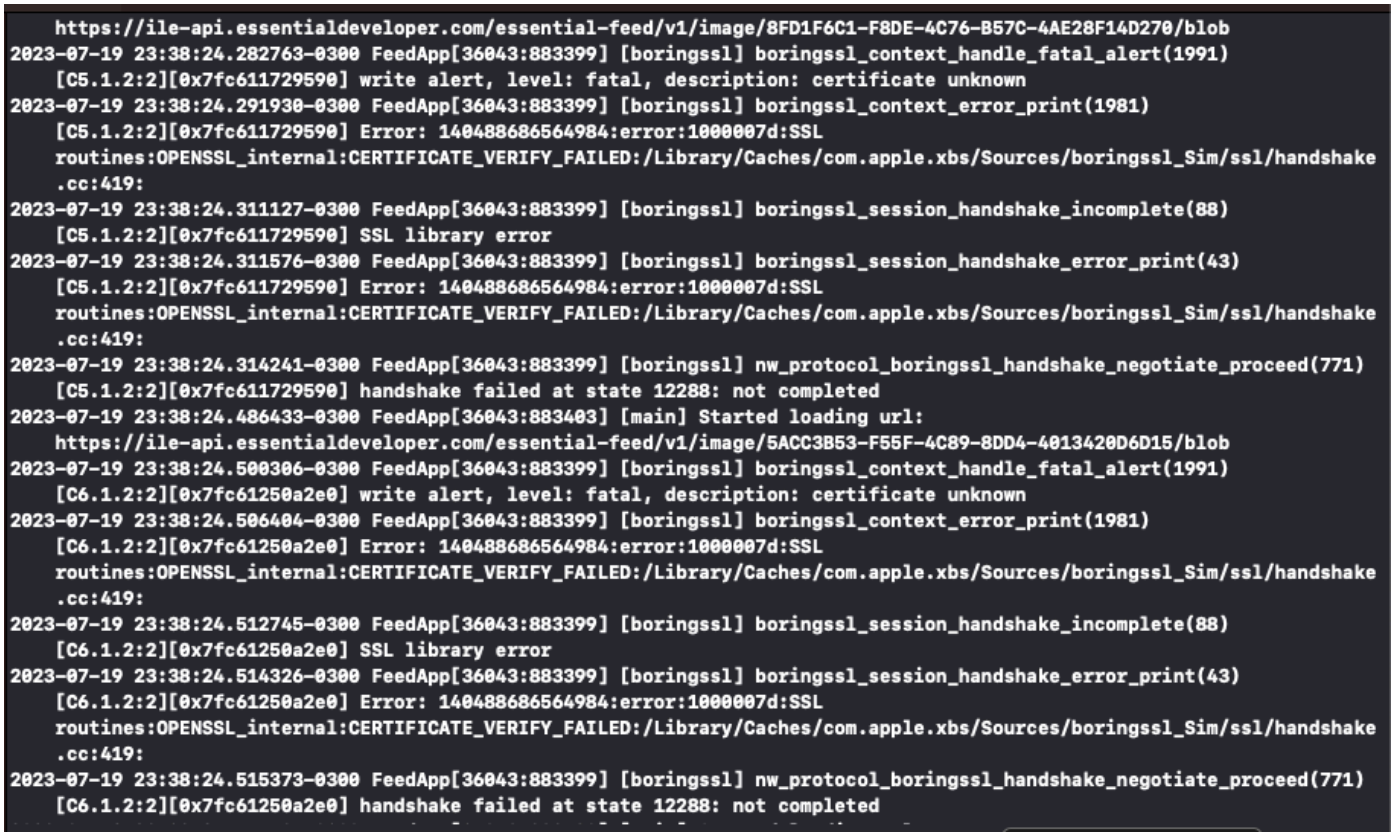
If we are using Combine, in fact, we dont even need to create the decorator we created, because with Combine, we can inject logging directly in to the publisher chain, using the **.handleEvents()** publisher that allows you to inject side effects according to the state of the subscription, since the handleEvents method can subscribe to the different stages of the subscription, we can inject code on subscription, on cancel, complete :



For example, we could do the exact same thing that we did with the decorator the following way:

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
    let localImageLoader = LocalFeedImageDataLoader(store: store)

    return localImageLoader
        .loadImageDataPublisher(from: url)
        .fallback(to: { [logger, httpClient] in
            var startTime = CACurrentMediaTime()
            return httpClient
                .getPublisher(url: url)
                .handleEvents(receiveSubscription: { [logger] _ in
                    logger.trace("Started loading url: \(url)")
                    startTime = CACurrentMediaTime()
                }, receiveCompletion: { [logger] result in
                    if case let .failure(error) = result {
                        logger.trace("Failed to load url: \(url), with error: \
(error.localizedDescription)")
                    }
                    let elapsedTime = CACurrentMediaTime() - startTime
                    logger.trace("Finished loading url: \(url) in \(elapsedTime)
seconds")
                })
                .tryMap(FeedImageDataMapper.map)
                .caching(to: localImageLoader, using: url)
        })
}
```

And the logging follows:



Which means we could do without the decorator we created earlier.

We can go one step further, so as not to pollute our existing code and create a **Combine** extension of **Publisher** and do:

```
extension Publisher {
    func logElapsedTime(url: URL, logger: Logger) -> AnyPublisher<Output, Failure> {
        var startTime = CACurrentMediaTime()

        return handleEvents(receiveSubscription: { _ in
            logger.trace("Started loading url: \(url)")
            startTime = CACurrentMediaTime()
        }, receiveCompletion: { result in
            if case let .failure(error) = result {
                logger.trace("Failed to load url: \(url), with error: \
(error.localizedDescription)")
            }
            let elapsedTime = CACurrentMediaTime() - startTime
            logger.trace("Finished loading url: \(url) in \(elapsedTime) seconds")
        })
        .eraseToAnyPublisher()
    }
}
```

And then chain it in our publisher stream:

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
        let localImageLoader = LocalFeedImageDataLoader(store: store)

        return localImageLoader
            .loadImageDataPublisher(from: url)
            .fallback(to: { [logger, httpClient] in
                return httpClient
                    .getPublisher(url: url)
                    .logElapsedTime(url: url, logger: logger)
                    .tryMap(FeedImageDataMapper.map)
                    .caching(to: localImageLoader, using: url)
            })
    }
```

And we get the same loggin results.

We could even break the login into elapsed time and log error:

```
extension Publisher {
    func logError(url: URL, logger: Logger) -> AnyPublisher<Output, Failure> {
        return handleEvents(receiveCompletion: { result in
            if case let .failure(error) = result {
                logger.trace("Failed to load url: \(url), with error: \
(error.localizedDescription)")
            }
        })
        .eraseToAnyPublisher()
    }

    func logElapsedTime(url: URL, logger: Logger) -> AnyPublisher<Output, Failure> {
        var startTime = CACurrentMediaTime()

        return handleEvents(receiveSubscription: { _ in
            logger.trace("Started loading url: \(url)")
            startTime = CACurrentMediaTime()
        }, receiveCompletion: { result in
            let elapsedTime = CACurrentMediaTime() - startTime
            logger.trace("Finished loading url: \(url) in \(elapsedTime) seconds")
        })
        .eraseToAnyPublisher()
    }
}
```

And now we can separately just log what we are interested in:

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
        let localImageLoader = LocalFeedImageDataLoader(store: store)

        return localImageLoader
            .loadImageDataPublisher(from: url)
            .fallback(to: { [logger, httpClient] in
                return httpClient
                    .getPublisher(url: url)
                    .logError(url: url, logger: logger)
                    .logElapsedTime(url: url, logger: logger)
                    .tryMap(FeedImageDataMapper.map)
                    .caching(to: localImageLoader, using: url)
            })
    }
```

We see how we can **inject** the logging behaviour into the **chain** , and if we are not using combine, we can use the method described with the decorator.

So with this logging we wound an issue: every time we reload, we reload all the images again. But, the cache should prevent reloading the same images again, so we can optimize the data consumption. We can also trace for cache misses (how many times it tries to find an image in the cache and it was empty).

So lets add a new logger in the same combine chain.

```
func logCacheMisses(url: URL, logger: Logger) -> AnyPublisher<Output, Failure> {
        return handleEvents(receiveCompletion: { result in
            if case let .failure(error) = result {
                logger.trace("Cache miss for url: \(url), with error: \
(error.localizedDescription)\n")
            }
        })
        .eraseToAnyPublisher()
    }
```

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
        let localImageLoader = LocalFeedImageDataLoader(store: store)

        return localImageLoader
            .loadImageDataPublisher(from: url)
            .logCacheMisses(url: url, logger: logger)
            .fallback(to: { [logger, httpClient] in
                return httpClient
```

```
                    .getPublisher(url: url)
                    .logError(url: url, logger: logger)
                    .logElapsedTime(url: url, logger: logger)
                    .tryMap(FeedImageDataMapper.map)
                    .caching(to: localImageLoader, using: url)
            })
    }
```

And now we can start logging the cache misses. We can see that every time we execute a reload, the number of cache misses increases constantly. It seems that the cache is failing for a lot of url's, if the cache worked properly we would avoid doing a lot of requests. So the next step is to optimize CoreData implementation to reduce cache misses.

Checking our existing code:

```
extension LocalFeedLoader: FeedCache {
    public typealias SaveResult = FeedCache.Result

    public func save(_ feed: [FeedImage], completion: @escaping (SaveResult) -> Void) {
        store.deleteCachedFeed { [weak self] deletionResult in
            guard let self = self else { return }

            switch deletionResult {
            case.success:
                self.cache(feed, with: completion)
            case .failure(let error):
                completion(.failure(error))
            }
        }
    }

    private func cache(_ feed: [FeedImage], with completion: @escaping (SaveResult) ->
Void) {
        store.insert(feed.toLocal(), timestamp: currentDate()) { [weak self] error in
            guard self != nil else { return }

            completion(error)
        }
    }
}
```

We can see, that when we save new images, we are deleting the existing cache to add the new one, and in doing so we are deleteing all the existing images, and if the new cache happens to contain the same images we have to load them again, which consumes resources unnecesarily, so we can change this implementation, we could handle **diffing** (seeing what is different and only removing what is different), but this would also complicate the service logic (the LocalFeedLoader). Deleting and Inserting is much simpler, it's usually its better to improve the performance in the infrastructure implementation so you dont pollute your services with performance improvements, because performance improvements are usually in the infrastructure, like if a database is too slow we can create an index for example.

Because if we start adding performance improvements in our services classes or our business logic its going to become very hard to mantain it, so what if instead we improved the performance in our infrastructure **CoreDataStore**, this way we wouldnt need to complicate our Loader service (LoadFeedLoader).

So we go to the infrastructure, to **ManagedFeedImage**, so can how we keep track of all the deleted FeedImages?

First, we can override `prepareForDeletion` which is called for each **CoreData** entity before its deletion. And we could store its image data in a temporary place. One way we can do it is to store it in the managedObjectContext?.userInfo dictionary which is not persisted in disk, its just a temporary lookup dictionary. So we can store here the data for that instance, if any for its **url** , before deleting. And when we are creating new images, we can try to find data for that url in the temporary cache:

```
static func images(from localFeed: [LocalFeedImage], in context:
NSManagedObjectContext) -> NSOrderedSet {
    let images =  NSOrderedSet(array: localFeed.map { local in
      let managed = ManagedFeedImage(context: context)
      managed.id = local.id
      managed.imageDescription = local.description
      managed.location = local.location
      managed.url = local.url
            managed.data = context.userInfo[local.url] as? Data
      return managed
    })
        context.userInfo.removeAllObjects()

        return images
  }

override func prepareForDeletion() {
    super.prepareForDeletion()

    managedObjectContext?.userInfo[url] = data
}
```

So, we are performing this optimization **IN THE INFRASTRUCTURE .**

Its important to note that we have to clear the temporary dictionary lookup table or we will be holding hundreds of images at some point , so in the same static method `images(from:...)` we remove all objects from the context like we can see in the previous code.

The idea with all this is that when we delete the existing cache to add a new cache, we will briefly store the active cache to then assign it to the new cache we are just creating, and at that moment we clean the dictionary. This dictionary secondary cache will have a very short lived life.

Another optimization we can do is in the static method `data(with url...)` we can first look up in the userInfo context dictionary to see if the data for the url exists in our temporary cache and if it does we return it immediately, so if we have the data in our temporary cache, we use it instead of making a database request which is much slower, because a dictionary lookup happens in constant time:

```
static func data(with url: URL, in context: NSManagedObjectContext) throws -> Data? {
        if let data = context.userInfo[url] as? Data { return data }

        return try first(with: url, in: context)?.data
    }
```

Now, if we run the app again we can see that we have 0 cache misses when reloading, and the only misses we see are the ones related to the image not being in the cache for the first time.

As we can see, we didn't even use the tests, because this performance issues are hard to test, thats why we used the network analyzer and logs.

If we pay close attention we can see that when we scroll really fast we are loading more than we should and therefore we are getting images that are being tried to be loaded more than two, three or even five times.

Taking a look at the FeedImageCellController we can see that we are calling the delegate method of `didRequestImage` many times and probably the requests are colliding since we are calling it from `cellForRowAt`, `onRetry`, `prefetchRowsAt` . For example, we start preloading a cell because it's about to become visible, and when it becomes visible if it's still loading, we start loading again, because we are requesting the image again. What we could do is avoid requesting again the image load if we are already loading it.

In this case, unlike the previous issue, we can write a test. So we are going to add a new test in the **FeedUIIntegrationTests** to test that a feedImageView already loading an image does not start loading again until previous request completes.

```
func test_feedImageView_doesNotLoadImageAgainUntilPreviousRequestCompletes() {
    let image = makeImage(url: URL(string: "http://url-0.com")!)
    let (sut, loader) = makeSUT()
    sut.loadViewIfNeeded()
    loader.completeFeedLoading(with: [image])
```

```
    sut.simulateFeedImageViewNearVisible(at: 0)
    XCTAssertEqual(loader.loadedImageURLs, [image.url], "Expected first request when
near visible")

    sut.simulateFeedImageViewVisible(at: 0)
    XCTAssertEqual(loader.loadedImageURLs, [image.url], "Expected no request until
previous completes")

    loader.completeImageLoading(at: 0)
    sut.simulateFeedImageViewVisible(at: 0)
    XCTAssertEqual(loader.loadedImageURLs, [image.url, image.url], "Expected second
request when visible after previous complete")

    sut.simulateFeedImageViewNotVisible(at: 0)
    sut.simulateFeedImageViewVisible(at: 0)
    XCTAssertEqual(loader.loadedImageURLs, [image.url, image.url, image.url], "Expected
third request when visible after canceling previous complete")
}
```

The idea is that we are testing that, everytime the **FeedImageCells** become either near visible, or visible, the **sut** fires a network request to load the content from that **image.url**, and it gets registered in the **loader.loadedImageURLS** array.

With this premise in mind, we first simulate that the desired Cell is about to become near visible, and thus executing a prefetch of that image url, this means that now, in our **loader.loadedImageURLs** array, we have captured the **image.url** that we are requesting. Now, since the premise is that while a Cell is being fetched, there shouldnt happen another request on the **image.url** of that same Cell, what we do is, without our previous request having finished, simulate that the cell is now visible, thus triggering another network request, on the same cell with the same url. However, this trigger should not happen, and thus, our **loader.loadedImageURLS** array shouldnt add this **image.url**, because the previous request has not yet finished, so what we expect to see is that the urls contained in our array of loaded image urls remains the same as before.

Next step is that we simulate that the image completed loading, and after that we execute, again that the FeedImageCell we are testing is visible, what this does is trigger another network request, meaning that our **loader.loadedImageURLS** array increases in size, adding a new element, which in this case, since we are testing the same cell, it will be another instance of **image.url** for the Cell we are analyzing and therefore our total array of loaded urls is now **[image.url, image.url]**.

The next step, is that before our previous trigger finishes loading, the Cell we were analyzing goes out of sight, meaning it no longer needs to be loaded. This means that this request must now be cancelled, and therefore, terminated. But immediately after, we simulate that the same Cell is now Visible, this means that, the initial request was cancelled but then we fired a new request, this means that our **loader.loadedImageURLs** that previously holded **[image.url, image.url]** will now add a new instance of that url, because the previous request, if well it did not complete, it was cancelled, and therefore finished, allowing us to begin a new request, therefore the new state of our loaded image urls is : **loader.loadedImageURLS = [image.url, image.url, image.url]**.

So, a way of preventing this is adding, in the **FeedImageCellController** an `isLoading` property, that checks, everytime that a fetch would be triggered, if it is already loading. But, preventing extra requests when one is already underway, is something we want to have everywhere, not just in the **FeedImageCellController** , so instead of doing it there, we could do it in one component above, one level higher in our design: in the shared **LoadResourcePresentationAdapter** that is used for loading any resource in the app. So if we implement this logic here, we effectively implement it in the whole app, since we don't ever want to use too much data.

So, in the presentation adapter, we are going to add a "**isLoading**" property that we will check, with a **guard** statement, before we begin the loading of any resource. If it's not loading, we will set the property to **true** and proceed to execute the loader **publisher** and **sink** it handling its **receiveCompletion** and its **receivedValue** as well as the **receivedCancel** events. In both the **receivedCancel** and the **receivedCompletion** event handlings, we will inject a closure where we will set the **isLoading** property to false to inform that the loading has finished.

```
func loadResource() {
    guard !isLoading else { return }

    presenter?.didStartLoading()
    isLoading = true

    cancellable = loader()
        .dispatchOnMainQueue()
        .handleEvents(receiveCancel: { [weak self] in
            self?.isLoading = false
        })
        .sink(
            receiveCompletion: { [weak self] completion in
                switch completion {
                case .finished: break

                case let .failure(error):
                    self?.presenter?.didFinishLoading(with: error)
                }

                self?.isLoading = false
            }, receiveValue: { [weak self] resource in
                self?.presenter?.didFinishLoading(with: resource)
            })
}
```

Its important to **weakify self** inside the publisher, otherwise we will create a retain cycle, since we are already holding the cancellation token.

It is important to set it to loading false on **receivedCancel** because otherwise on the cancelled requests we will have issues, and its also important that on our **tests**, we add a **send(completion: .finished)** event after we simulate the **completeFeedLoading** because otherweise the **receivedComplete** event wont get triggered. This last mentioned thing is neccessary in both our **FeedUIIntegrationTests** aswell as in the **CommentsUIIntegrationTests**, otherwise, when we run the tests we get crashes in runtime because it tries

to access out of bounds objects. So, we modify both the **completeFeedLoading** and the **completeCommentsLoading** methods in the following way:

```
// In the CommentsUIIntegrationTests.swift
func completeCommentsLoading(with comments: [ImageComment] = [], at index: Int = 0) {
    requests[index].send(comments)
    requests[index].send(completion: .finished)
}
// In the FeedUIIntegrationTests+LoaderSpy.swift
func completeFeedLoading(with feed: [FeedImage] = [], at index: Int = 0) {
    feedRequests[index].send(Paginated(items: feed, loadMorePublisher: { [weak self] in
        self?.loadMorePublisher() ?? Empty().eraseToAnyPublisher()
    }))
    feedRequests[index].send(completion: .finished)
}
```

A publisher can emit infinite values, but it will only stop sending events when it either fails or when you send a .**finish** event.

Ideally though, you shouldnt log everything. If there is a test for it, you should test it with a test and not a log.

We can still see that some images are still being loaded multiple times which is more than what we would want. We still have performance improvements we can make. It seems that everytime we load a new page, we repeat some image requests, thats probably because every time we load a new page we append the cache items into the new page and we recreate all the cell controllers in the **feedviewadapter**, so everytime we get a new page we dont get only the new items, we get all the items, the cached ones appended with the new ones, and the implementation currently simply iterates through that collection of paginated items and recreates all the cell controllers.

This is not efficient, it works, but its not efficient.

So in this case, in our FeedViewAdapter we could reuse the existing cellcontrollers from previous pages that were already created and allocated and just append new pages into it, so we avoid a bunch of allocations and we avoid recreating presentation adapters because then we would lose the **isLoading** state.

For this case we can write an extra assertion in our already existing test `test_feedImageView_doesNotLoadImageAgainUntilPreviousRequestCompletes` in the **FeedUIIntegrationTests** , at the end of the existing test we can add:

```
// Previous flow of the test
sut.simulateFeedImageViewNotVisible(at: 0)
sut.simulateFeedImageViewVisible(at: 0)
XCTAssertEqual(loader.loadedImageURLs, [image.url, image.url, image.url], "Expected
third request when visible after canceling previous complete")

// New assertions in the test
sut.simulateLoadMoreFeedAction()
loader.completeLoadMore(with: [image, makeImage()])
sut.simulateFeedImageViewVisible(at: 0)
XCTAssertEqual(loader.loadedImageURLs, [image.url, image.url, image.url], "Expected no
request until previous completes")
```

First of all we simulate the action of loading a new page and we complete the load more action with the image that was originally loaded and we append a new image. So at this moment we are creating two cell controllers, one for the image, we already had, and one for the new image, and if the first **image** was loading data which is the case here, because it is still loading (it still hasnt completed) it will recreate the cell controller and start loading again.

As we can see from the test code provided above, the "simulateFeedImageViewVisible(at: 0)" still has not finished loading, but in the middle of it we triggered a load more, to get a new page, and then, the cell becomes visible **AGAIN** because the load more ation, will trigger the re-creation of the cell controller, which will void the previous **loading** status and start loading again, but it shouldn't, it should still be loading 3 images, which is the state that the flow of our test had before triggering the load more action.

This is because the load more action doesnt complete the loading from the image, meaning the image from the cell already undergoing loading shouldnt re-trigger an image fetch request.

When we run the previous test, we will get a failure on the last assertion, it says that it has 4 loadedImageURL's when in fact they should be 3.

So we have to stop having cells re-allocated that already exist and avoid having our *isLoading* property voided.

How do we do this?

Somehow we need to keep track of the existing cellControllers , there are many ways we could do that, we could expose for example all the existing cellControllers from the user inteface from the ListViewController or we can deal with the optimization in the infrastructure directly, in the **FeedViewAdapter** and that would be ideal so that we dont pollute other components with optimizations.

So, we create a lookup table just like we did in the **CoreDataStore** but in this case we will have a **currentFeed** as our lookup table, a dictionary of **[FeedImage: CellController]** . We then initialize our class with the **currentFeed** , starting empty.

```
class FeedViewAdapter {
```

```
....
....
private let currentFeed: [FeedImage: CellController]
...
...
init(currentFeed: [FeedImage: CellController] = [:], controller: ListViewController,
imageLoader: @escaping (URL) -> FeedImageDataLoader.Publisher, selection: @escaping
(FeedImage) -> Void) {
    self.currentFeed = currentFeed
    self.controller = controller
    self.imageLoader = imageLoader
        self.selection = selection
  }
  ...
  ...
}
```

So now, with this lookup table, when we are creating a new CellController inside the `display(_ viewmodel: )` method, we can check the lookup table for a given model and if we have a value for it, we dont have to recreate it. Good thing about Dictionaries is that we get constant time lookup. So, inside the **display** method, inside the mapping method creating the **feed** (array of cellcontrollers), we check if a CellController already exists for the given **model** and if it does, we return it

```
if let controller = currentFeed[model] {
        return controller
}
```

But, since we need to populate the current feed, we create a mutable version of our currentFeed inside our **display** method, so that if we didn't find the CellController, we will create a new one, append it to the **currentFeed** lookup table for the given model and return it. So next time we look up the CellController in the currentFeed, it will be there. All of this logic happens inside the mapping logic that creates the feed array of CellControllers inside the **display** method.

```
let feed: [CellController] = viewModel.items.map { model in
            if let controller = currentFeed[model] {
                return controller
            }

            let adapter = ImageDataPresentationAdapter(loader: { [imageLoader] in
                imageLoader(model.url)
            })

            let view = FeedImageCellController(
                viewModel: FeedImagePresenter.map(model),
                delegate: adapter,
                selection: { [selection] in
```

```
                    selection(model)
                })

            adapter.presenter = LoadResourcePresenter(
                resourceView: WeakRefVirtualProxy(view),
                loadingView: WeakRefVirtualProxy(view),
                errorView: WeakRefVirtualProxy(view),
                mapper: UIImage.tryMake)

            let controller = CellController(id: model, view)
            currentFeed[model] = controller
            return controller
        }
```

And, in the following part of the **display** method, when creating a new page instead of using `self` as the **resourceView** for the new **LoadResourceAdapter**, we can create a new **FeedViewAdapter** passing in the **currentFeed** that we just updated/populated, so we just keep passing the **currentFeed** forward, so we never lose track of the pages and we dont need a mutable property. For this we will need to guard that our **controller: ListViewController** property that the **FeedViewAdapter** uses isnt nil, since its initializer requires a non nil controller, since without having a ListViewController we cannot present a new page. (Its important to remember that we are currently holding a weak reference to the **controller: ListViewController**) in our FeedViewAdapter, because we do not want to have a retain cycle, so adding these changes, our final **display(_ viewModel:)** method looks like:

```
func display(_ viewModel: Paginated<FeedImage>) {
    guard let controller = controller else { return }

    var currentFeed = self.currentFeed
    let feed: [CellController] = viewModel.items.map { model in
        if let controller = currentFeed[model] {
            return controller
        }

        let adapter = ImageDataPresentationAdapter(loader: { [imageLoader] in
            imageLoader(model.url)
        })

        let view = FeedImageCellController(
            viewModel: FeedImagePresenter.map(model),
            delegate: adapter,
            selection: { [selection] in
                selection(model)
            })

        adapter.presenter = LoadResourcePresenter(
            resourceView: WeakRefVirtualProxy(view),
            loadingView: WeakRefVirtualProxy(view),
```

```
            errorView: WeakRefVirtualProxy(view),
            mapper: UIImage.tryMake)

        let controller = CellController(id: model, view)
        currentFeed[model] = controller
        return controller
    }

    guard let loadMorePublisher = viewModel.loadMorePublisher else {
        controller.display(feed)
        return
    }

    let loadMoreAdapter = LoadMorePresentationAdapter(loader: loadMorePublisher)
    let loadMore = LoadMoreCellController(callback: loadMoreAdapter.loadResource)

    loadMoreAdapter.presenter = LoadResourcePresenter(
        resourceView: FeedViewAdapter(
            currentFeed: currentFeed,
            controller: controller,
            imageLoader: imageLoader,
            selection: selection
        ),
        loadingView: WeakRefVirtualProxy(loadMore),
        errorView: WeakRefVirtualProxy(loadMore))

    let loadMoreSection = [CellController(id: UUID(), loadMore)]

    controller.display(feed, loadMoreSection)
}
```

We run the tests again and we see that they pass: the lookup table has solved our problem. This way we avoid reallocating already existing instances since our instances are somewhat expensive, and also, if we reallocate an existing already allocated instance we are losing our **isLoading** state which we need to avoid unnecesary network fetches.

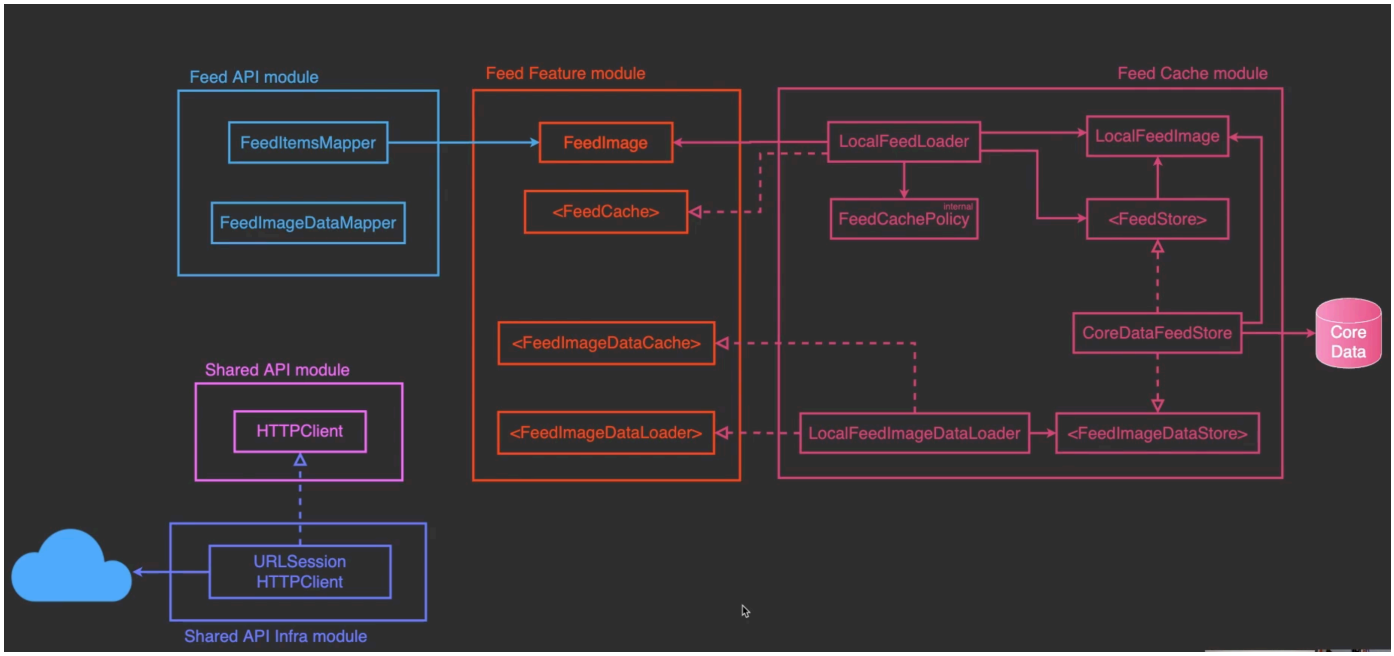
Async Injection: Decoupling the Domain from Infra Details

- [ ] Eliminating Async Boilerplate & Nested Callbacks (Arrow Code/Pyramid of Doom anti-pattern)
- [ ] Threading as a Cross-Cutting Concern (Decorator Pattern & Universan Abstractions)

- [ ] Combine Schedulers
- [ ] Type Erasure

Quick Recap Live Session #001:

It was shown how to decouple modules from infrastructure details, for example, we refactor the FeedAPI Module and we decoupled it from the HTTPClient as we can see in the following image, no arrows are going from the FeeedAPI Module to the Client.



The Client lives in its own Infrastructure Module, and we compose the FeedAPI Logic with the infrastructure in the Composition Root (SceneDelegate). So, our FeedAPI Module doesn't know about the Infrastructure, and the infrastructure doesn't know about the FeedAPI Module.

```
  private func makeRemoteFeedLoader(after: FeedImage? = nil) ->
AnyPublisher<[FeedImage], Error> {
        let url = FeedEndpoint.get(after: after).url(baseURL: baseURL)

        return httpClient
            .getPublisher(url: url)
            .tryMap(FeedItemsMapper.map)
            .eraseToAnyPublisher()
    }
```

They are composed as we see above, in the SceneDelegate.

As a result we eliminated a lot of boilerplate code, we deleted the FeedLoader protocol which was async with the completion block, and we also made the FeedAPI Logic, the mapper, synchronous (input-output).

Going from asynchronous code to synchronous code makes everything simpler, since we dont have completion blocks, closures etc.

Its important to bear in mind that the infrastructure, the HTTPClient, is still asynchronous and it makes the request and at somepoint its going to complete, and then we map that response with our mapper, which is syncrhonous, effectively decoupling the synchrony details in the infrastructure details from the FeedAPI module.

The question now is: can we make other modules synchronous aswell and move asynchrony always to the composition root? Yes, we can, we can follow the same process as followed in Live #001 to decouple the modules from asynchrony.

So, the steps were:

- We decouple the module from the infrastructure details
- We make the module functions synchronous
- Then we compose it in the infrastructure either using Decorators, Composites or Combine/RxSwift.

Today we are going to show a different way of doing it, we are going to convert another async API into a sync API to decouple modules from Async details, which will simplify the development, testing and maintenance of those components. Of course we still want to mantain the same benefits of async excecution such as running network and database operations concurrently without blocking the main thread.

Migrating Async Modules to Sync

For example, the protocol, which is defined in the **Domain Module**, is asynchronous, it has a completion block. So the service abstraction, which is defined in the domain layer, is only asynchronous because of the infrastructure details. Same thing happens with the , which is also async and has a completion block. Also the , it's a domain abstraction which is async with a completion block.

All of these abstractions in the **domain module** are only asynchronous because the infrastructure implementation needs to be asynchronous, because we don't want to block the main thread.

Same thing is happening with infrastructure abstractions, the aswell as the which, along the implementation, are running async since we dont want to run the DB queries synchronously and block the clients (same as we wouldn't want to block them when making a network request), but then, we are leaking infrastructure details into the **Domain**.

This is pretty common in Swift Async API's, it's common to see a bunch of Domain abstractions with either completion blocks or even RxSwift/Combine publishers, because you need to deal with asynchrony. This is not a big problem but its a leaky abstraction, and if you can eliminate the Leaky abstraction, you can make your domain simpler to develop/maintain and test.

This infrastructure being async "problem", becomes a chain effect that leaks into the rest of our domains, forcing us to make every API async. We don't want to leak infrastructure details.

A solution to avoid leaking infrastructure details into everywhere is to make the infrastructure synchronous, so we don't leak these async details everywhere. And we move asynchrony into the composition root. (threading is a cross cutting concern, and we can deal with cross-cutting concerns in the composition root so we avoid coupling modules with unnecesary details).

Eliminating Completion Blocks (avoiding pyramid of doom)

Starting with , we analyze the `insert` and `retrieve` methods to see how we can get rid of the completion code.

We propose synchronous method signatures that are completely synchronous, but that can fail, therefore they **throw**:

```
func insert(_ data: Data, for url: URL) throws
func retrieve(dataForURL url: URL) throws -> Data?
```

We are making these synchronous and eliminating the completion blocks we can then make every component synchronous and deal with asyncrony somewhere else (Composition Root).

Nevertheless, changing this interface will break clients, so temporarily we will add an extension with default implementations that will use the async API's that we already have, but respecting the synchronous signature. For this we will create a **DispatchGroup**, enter the group, and then inside the async API, when the closure is executed, we assign the result, and leave the group. Outside the closure, we wait for the completion async completion to finish (in that time, the client is locked in the meanwhile, which is why it becomes synchronous). Finally we try to get the result and return it.

```
func retrieve(dataForURL url: URL) throws -> Data? {
    let group = DispatchGroup()

    group.enter()
    var result: RetrievalResult!
    retrieve(dataForURL: url, completion: {
        result = $0
        group.leave()
    })
    group.wait()

    return try result.get()
}
```

We do the same thing for the insertion:

```
func insert(_ data: Data, for url: URL) throws {
    let group = DispatchGroup()

    group.enter()
    var result: InsertionResult!
    insert(data, for: url) {
        result = $0
        group.leave()
    }
    group.wait()

    return try result.get()
}
```

So if we fail this will throw an error, and if it succeeds it will return the **Data**.

This is just a temporary implementation so that we dont break the clients.

Next step is to deprecate the original asynch API's with an **@available(*, deprecated)** statement before the protocol signatures, and also provide default empty implementations for the deprecated implementations:

```
func insert(_ data: Data, for url: URL, completion: @escaping (InsertionResult) ->
Void) {}
func retrieve(dataForURL url: URL, completion: @escaping (RetrievalResult) -> Void) {}
```

After we are done, we are going to remove the new implementations of the async-sync methods.

Migrating the deprecated APIs to the new API's

Now we are going to follow the compiler to see where the warnings regarding our deprecated methods are and migrate to the new sync API's.

Save Images Synchronously

First we will migrate the **LocalFeedImageDataLoader: FeedImageDataCache** extension that conforms to the protocol API for saving, the current code is:

```
extension LocalFeedImageDataLoader: FeedImageDataCache {
    public typealias SaveResult = FeedImageDataCache.Result

    public enum SaveError: Error {
        case failed
    }

    public func save(_ data: Data, for url: URL, completion: @escaping (SaveResult) ->
Void) {
        store.insert(data, for: url) { [weak self] result in
            guard self != nil else { return }
```

```
                completion(result.mapError { _ in SaveError.failed })
            }
        }
    }
```

Where the compiler tells us that **insert** is deprecated.

For this migration, we will start with a test in the **CacheFeedImageUseCaseTests**. Inspecting the test helpers such as **expect**, we see how we depend on asynchrony, using the expectations, we end up having to wait, but we can simplify that logic now with the async API's.

What we are going to do is convert the **FeedImageDataStoreSpy**'s async methods into sync. At the moment what our spies' methods are doing is capturing the completion blocks so that we can call them in the future, but if the spy's implementation were synchronous, we wouldnt need to capture, we would need to stub the result upfront, to return something, either error or void, for which we create a new **private var insertionResult: Result<Void, Error>?** property that we will use for stubbing:

```
  private var insertionResult: Result<Void, Error>?

    func insert(_ data: Data, for url: URL) throws {
        receivedMessages.append(.insert(data: data, for: url))
        try insertionResult?.get()
    }

//We also modify the helpers
func completeInsertion(with error: Error) {
    insertionResult = .failure(error)
}

func completeInsertionSuccessfully() {
    insertionResult = .success(())
}
```

its important to understand that in the sync API's we need to stub the value before the method is invoked before we need to return a value immediately, synchronously.

This can be seen clearly in our **expect** method helper:

```
private func expect(_ sut: LocalFeedImageDataLoader, toCompleteWith expectedResult: LocalFeedImageDataLoader.SaveResult, when action: () -> Void, file: StaticString =
    #filePath, line: UInt = #line) {
    let exp = expectation(description: "Wait for load completion")

    sut.save(anyData(), for: anyURL()) { receivedResult in
        switch (receivedResult, expectedResult) {
        case (.success, .success):
            break

        case (.failure(let receivedError as LocalFeedImageDataLoader.SaveError),
              .failure(let expectedError as LocalFeedImageDataLoader.SaveError)):
            XCTAssertEqual(receivedError, expectedError, file: file, line: line)

        default:
            XCTFail("Expected result \(expectedResult), got \(receivedResult) instead", file: file, line: line)
        }

        exp.fulfill()
    }

    action()
    wait(for: [exp], timeout: 1.0)
}
```

the **action** closure, which in the previous image is shown to be executed at the end, in the new synch API's it needs to be executed at the begginning:

```
private func expect(_ sut: LocalFeedImageDataLoader, toCompleteWith expectedResult: LocalFeedImageDataLoader.SaveResult, when action: () -> Void, file: StaticString =
    #filePath, line: UInt = #line) {
    let exp = expectation(description: "Wait for save completion")
    action()

    sut.save(anyData(), for: anyURL()) { receivedResult in
        switch (receivedResult, expectedResult) {
        case (.success, .success):
            break

        case (.failure(let receivedError as LocalFeedImageDataLoader.SaveError),
              .failure(let expectedError as LocalFeedImageDataLoader.SaveError)):
            XCTAssertEqual(receivedError, expectedError, file: file, line: line)

        default:
            XCTFail("Expected result \(expectedResult), got \(receivedResult) instead", file: file, line: line)
        }

        exp.fulfill()
    }

    action()
    wait(for: [exp], timeout: 1.0)
}
```

because the Stub NEEDS to be configured before executing the sync method. As expected, these changes ake our tests fail, which is the cue to go fix them in the **LocalFeedImageDataLoader: FeedImageDataCache** extension. So in our original async code we had:

```
public func save(_ data: Data, for url: URL, completion: @escaping (SaveResult) ->
Void) {
    store.insert(data, for: url) { [weak self] result in
        guard self != nil else { return }
        completion(result.mapError { _ in SaveError.failed })
    }
}
```

Where our async api was waiting for the completion to then call the completion block. We can now do that synchronously by wrapping the result and calling the synchronous API without the completion block:

```
public func save(_ data: Data, for url: URL, completion: @escaping (SaveResult) ->
Void) {
        completion(SaveResult {
            try store.insert(data, for: url)
        }.mapError {_ in SaveError.failed })
}
```

This will be executed synchronously. Now the tests run without error, except for a deallocation test that has no reason to exist anymore since we are not working async anymore, so we delete it.

Load images Syncronously

Now we are going to attend to the warning regarding the loading of images for the **retrieve** method.

In the same fashion as in the previous case, we will modify our Spy's result switching from a completion array, to a stub, returning either optional data or error.

```
private var retrievalResult: Result<Data?, Error>?
```

```
func retrieve(dataForURL url: URL) throws -> Data? {
    receivedMessages.append(.retrieve(dataFor: url))
    return try retrievalResult?.get()
}

func completeRetrieval(with error: Error) {
    retrievalResult = .failure(error)
}

func completeRetrieval(with data: Data?) {
    retrievalResult = .success(data)
}
```

Same as before, we use Stubs in our Spy, and now all our spy methods are synchronous. The same procedure we applied to the **expect** helper from **save** applies here, we need to run the **action** closure at the begginning of the **expect** helper method instead of at the end. Same as with the **save** method, the tests regarding asynchrony and the correct results expected from deallocation can be removed since we are now working with synchronous tasks. (Regarding the cancel task test: you cant cancel a synchronous process because it happens instantly, and all the cancelling regarding the asynchronous tasks are handled in the Composition Root).

We run the test and they fail, which leads us to modifying our **load** code in the **LocalFeedImageDataLoader: FeedImageDataLoader** extension, in the method **loadImageData(from url:)**

Previous code:

```
public func loadImageData(from url: URL, completion: @escaping (LoadResult) -> Void) ->
FeedImageDataLoaderTask {
    let task = LoadImageDataTask(completion)
    store.retrieve(dataForURL: url) { [weak self] result in
        guard self != nil else { return }

        task.complete(with: result
            .mapError { _ in LoadError.failed }
            .flatMap { data in
                data.map { .success($0) } ?? .failure(LoadError.notFound)
            })
    }
    return task
}
```

```
public func loadImageData(from url: URL, completion: @escaping (LoadResult) -> Void) ->
FeedImageDataLoaderTask {
    let task = LoadImageDataTask(completion)

    task.complete(
```

```
        with: Swift.Result {
        try store.retrieve(dataForURL: url)
        }
        .mapError { _ in LoadError.failed }
        .flatMap { data in
            data.map { .success($0) } ?? .failure(LoadError.notFound)
        })

    return task
}
```

Now the tests pass, and we have removed one level of indentation since we dont have completion blocks.

Making the Domain Abstractions synchronous

Make FeedImageDataCache sync

Now we are going to look at our domain abstractions to make them synchronous, we start with **FeedImageDataCache** and we will go from this:

```
public protocol FeedImageDataCache {
    typealias Result = Swift.Result<Void, Error>

    func save(_ data: Data, for url: URL, completion: @escaping (Result) -> Void)
}
```

To this:

```
public protocol FeedImageDataCache {
    func save(_ data: Data, for url: URL) throws
}
```

Since the API is now synchronous we do not need for completion results because it either saves or throws an error that we will later catch.

We now follow the compiler and fix the errors we got with this change in the **LocalFeedImageDataLoader: FeedImageDataCache** extension. For starters we dont need the result type, since its now synchronous and we modify our **save** method from:

```
public func save(_ data: Data, for url: URL, completion: @escaping (SaveResult) ->
Void) {
    completion(SaveResult {
        try store.insert(data, for: url)
    }.mapError {_ in SaveError.failed })
}
```

to:

```
public func save(_ data: Data, for url: URL) throws {
    do {
        try store.insert(data, for: url)
    } catch {
        throw SaveError.failed
    }
}
```

So, we try to save the data, and if it fails we catch the error and throw a personalized error.

Now we have to fix the tests, which were still configured for the async API. Since we don't have completion closures anymore, we reduce one indentation and we have less code.

Finally, we have to modify our **LocalFeedLoader** extension method **saveIgnoringResult**.

Make FeedImageDataLoader sync

Now we will apply the same process to make our **FeedImageDataLoader**, we want to convert it from Async, to Sync. This conversion is a bit more complex, since we have a cancel task

```
public protocol FeedImageDataLoaderTask {
    func cancel()
}
public protocol FeedImageDataLoader {
    typealias Result = Swift.Result<Data, Error>
    func loadImageData(from url: URL, completion: @escaping (Result) -> Void) ->
FeedImageDataLoaderTask
}
```

But as we said before, we get this for free in the composition root, so we can get rid of it and refactor our **loadImageData** signature, which will now return just Data and if it fails it will throw, we also get rid of the result:

```
public protocol FeedImageDataLoader {
    func loadImageData(from url: URL) throws -> Data
}
```

which is the new sync API.

We follow the compiler and go fix the **LocalFeedImageDataLoader: FeedImageDataLoader** extension, for starters we dont need the **LoadResult** type alias anymore or the **LoadImageDataTask** , our **loadImageData** method goes from this:

```
public func loadImageData(from url: URL, completion: @escaping (LoadResult) -> Void) ->
FeedImageDataLoaderTask {
        let task = LoadImageDataTask(completion)

        task.complete(
            with: Swift.Result {
```

```
            try store.retrieve(dataForURL: url)
            }
            .mapError { _ in LoadError.failed }
            .flatMap { data in
                data.map { .success($0) } ?? .failure(LoadError.notFound)
            })

        return task
    }
```

to this:

```
public func loadImageData(from url: URL) throws -> Data {
        do {
            if let imageData = try store.retrieve(dataForURL: url) {
                return imageData
            }
        } catch {
            throw LoadError.failed
        }

        throw LoadError.notFound
    }
```

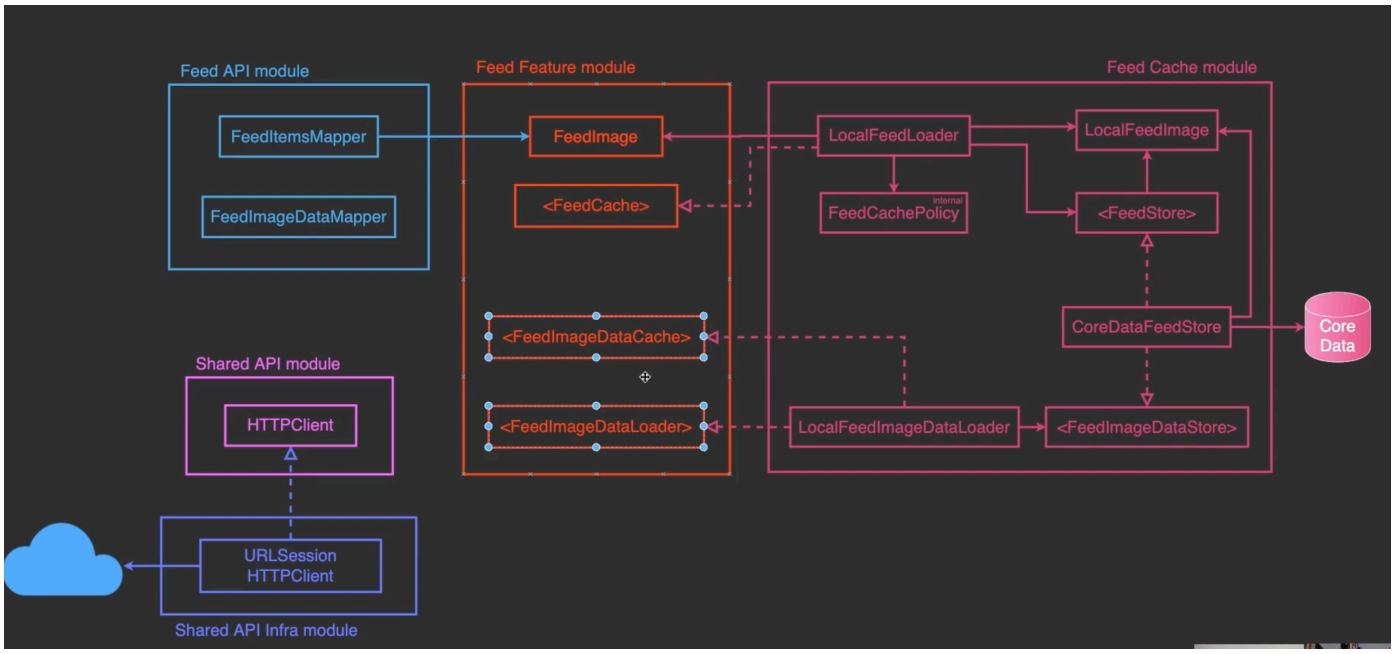
In the same fashion we can start modifying our tests that required expectations for tests without them, since we dont need them now. We also get rid of one level of indentation, as said before.

Finally we also have to modify our CombineHelpers file, the **FeedImageDataLoader** extension, since we went Sync, the **FeedImageDataLoaderTask** doesn't exist anymore, and every publisher is cancellable by default, so we go from this:

```
public extension FeedImageDataLoader {
    typealias Publisher = AnyPublisher<Data, Error>

    func loadImageDataPublisher(from url: URL) -> Publisher {
        var task: FeedImageDataLoaderTask?

        return Deferred {
            Future { completion in
                task = self.loadImageData(from: url, completion: completion)
            }
        }
        .handleEvents(receiveCancel: { task?.cancel() })
        .eraseToAnyPublisher()
    }
}
```

to this:

```
public extension FeedImageDataLoader {
    typealias Publisher = AnyPublisher<Data, Error>

    func loadImageDataPublisher(from url: URL) -> Publisher {
        return Deferred {
            Future { completion in
                completion(Result {
                    try self.loadImageData(from: url)
                })
            }
        }
        .eraseToAnyPublisher()
    }
}
```



Now we've made both the Cache and the synchronous, and we can do the same for the . But first we'll see to make the synchronous.

So, we go to **CoreDataFeedStore** and we see that the **perform** method provided by core data is async. So we will rename it to show that to **performAsync**.

Next step is to make CoreData implementation syncronous. For this we will need to create a sync version of it.

We will start with the CoreData tests in the CoreDataFeedImageDataStoreTests. We need to modify our **expect** method and stop using expectations, since those were to test async api's and now we are working with sync apis. We will also modify the other helper methods such as **insert** in the same way. What we wont change, at least not now, is that our FeedStore still is asynchronous, so in our test code, in the **insert** helper

method we will keep that.

Initially the tests will pass, since in the **FeedImageDataStore** we have created our sync versions of the methods **insert** and **retrieve** which internally make use of a **DispatchGroup** to execute async code in an sync mode, making the clients wait. This will be changed in the future, but for now the tests pass.

Going back to the **CoreDataFeedStore+FeedImageDataStore** we examine our **insert** and **retrieve** methods, and we see they are being used with **performAsync** method we created to work with **CoreData**. Our next goal is to create a **performSync** method that allows us to work synchronously. We will create this method below the **performAsync** method, in the **CoreDataFeedStore** .

In synchronous operations, closures dont need to be escaping, since we are not holding a reference to it, it's not escaping the scope of the function, and they will execute immediately, and when it returns it doesn't hold any reference to the closure. Therefore the **action** closure of our **performSync** wont be escaping.

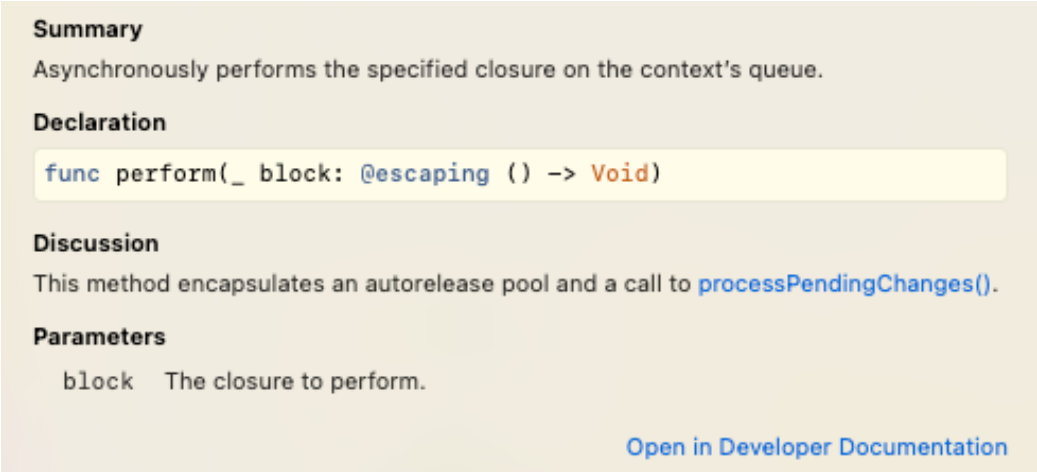
So, a sync API needs to return the result immediately, not in the future, so we need to return a result here, but what should we return? Somes we return **Data?** for the retrieve and sometimes we return **Void**, meaning the result changes depending on the operation.

So we can create a generic result type and we can then returl the generic result. And also our closure returning **action** can also return a Result<R, Error>, and if it fails it should throw an error:

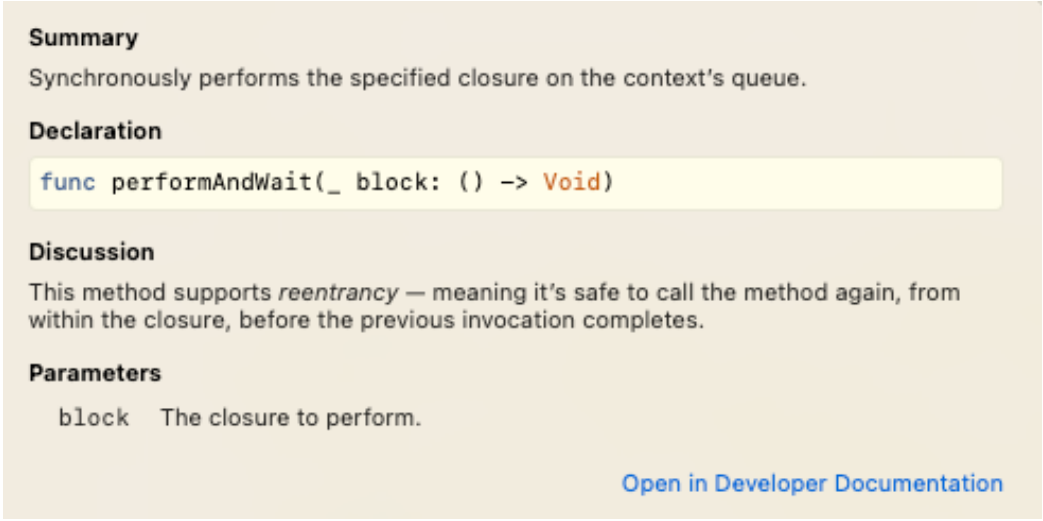
```
func performSync<R>(_ action: (NSManagedObjectContext) -> Result<R, Error>) throws -> R
```

```
func performSync<R>(_ action: (NSManagedObjectContext) -> Result<R, Error>) throws -> R {
    let context = self.context
    var result: Result<R, Error>!
    context.perform {                                   Escaping closure captures non-escaping parameter 'action'
        result = action(context)
    }
    return try result.get()
}
```

We mentioned earlier that we do not need an escaping closure, since we are executing the task synchronously, but we can see that CoreData's **perform** method



So what we can do is use the **performAndWait** method, which is synchronous and will wait until the task is completed to continue



finally our **performSync** method looks like:

```
 func performSync<R>(_ action: (NSManagedObjectContext) -> Result<R, Error>) throws ->
R {
    let context = self.context
    var result: Result<R, Error>!
    context.performAndWait {
        result = action(context)
    }
    return try result.get()
}
```

We either got a result or it will throw an error.

Now going back to the API's in the **CoreDataFeedStore: FeedImageDataStore** extension, we can perform our **insert** and **retrieve** using the new sync API's, by using the **performSync** method:

Going from this:

```
extension CoreDataFeedStore: FeedImageDataStore {
    public func insert(_ data: Data, for url: URL, completion: @escaping
(FeedImageDataStore.InsertionResult) -> Void) {
        performAsync { context in
            completion(Result {
                try ManagedFeedImage.first(with: url, in: context)
                    .map { $0.data = data }
                    .map(context.save)
            })
        }
    }

    public func retrieve(dataForURL url: URL, completion: @escaping
(FeedImageDataStore.RetrievalResult) -> Void) {
        performAsync { context in
            completion(Result {
                try ManagedFeedImage.data(with: url, in: context)
```

```
            })
        }
    }
}
```

To this:

```
extension CoreDataFeedStore: FeedImageDataStore {
    public func insert(_ data: Data, for url: URL) throws {
        try performSync { context in
            Result {
                try ManagedFeedImage.first(with: url, in: context)
                    .map { $0.data = data }
                    .map(context.save)
            }
        }
    }

    public func retrieve(dataForURL url: URL) throws -> Data? {
        try performSync { context in
            Result {
                try ManagedFeedImage.data(with: url, in: context)
            }
        }
    }
}
```

We see how we have eliminated the completion blocks.

Removing the deprecated async API's

So, now we have everything we need regarding the new sync API's, and we can therefore remove the async, deprecated api's. We are going to remove the RetrievalResult and the InsertionResult aswell as the async methods of **retrieve** and **insert** , we will also remove the helper extension, and our new **FeedImageDataStore** looks like:

```
public protocol FeedImageDataStore {
    func insert(_ data: Data, for url: URL) throws
    func retrieve(dataForURL url: URL) throws -> Data?
}
```

Some changes must be done to the CoreDateFeedImageDataStoreTests too, since we removed the result types aswell. We will be getting rid of the test_sideEffects_runSerially, since all the sync processes run serially.

We must also modify our **NullStore: FeedImageDataStore** extension that provides the default **insert** and **retrieve** methods, to reflect the changes we made. We must modify too, the **InMemoryFeedStore: FeedImageDataStore** that also was making use of the async apis, and change them to work with the sync apis.

So, all the API's are now synchronous: FeedImageDataCache, FeedImageDataLoader, etc. So the abstractions are syncronous, which makes the implementations much simpler as well and we are not leaking the asynchrony details everywhere.

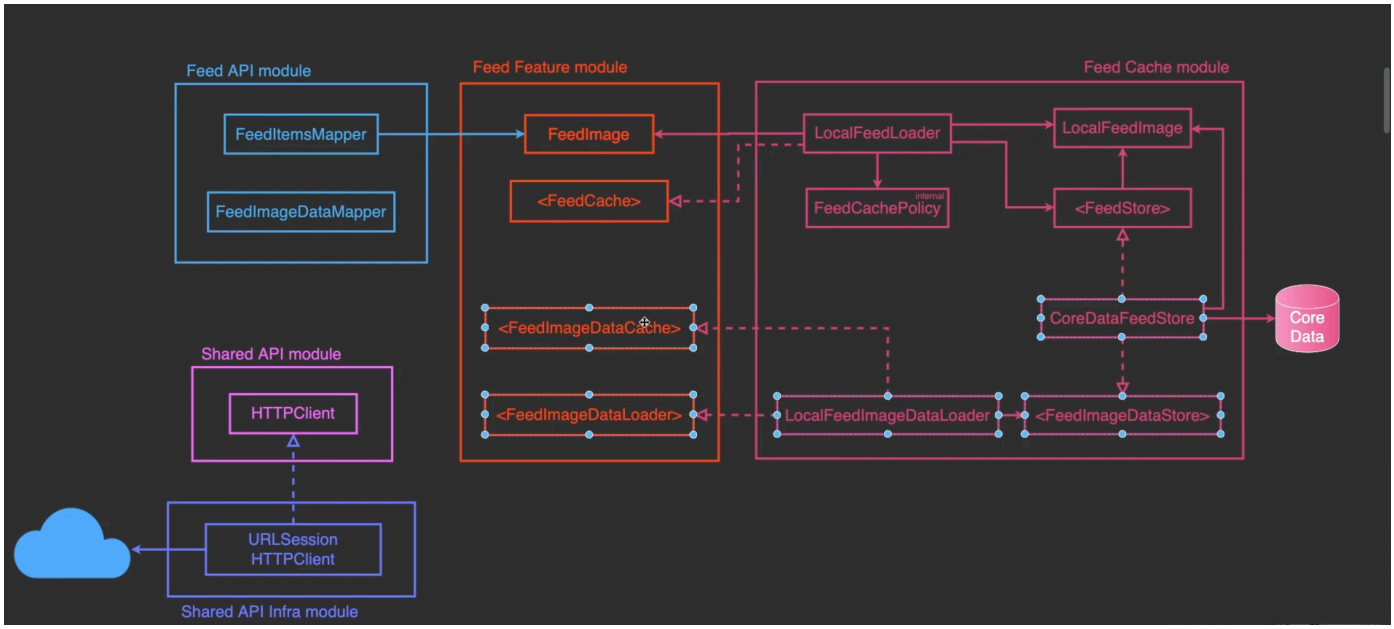
If we run the app again, and add a breakpoint on our newly created method **performSync**, we can see that its being executed in the Main Thread, and, that the method **performAndWait** is indeed blocking the Main Thread/queue, until the CoreData query finishes. This can make the UI unresponsive for a while depending on how long it takes to execute the query and we dont want that.

How can we maintain te benefits of an async API with a sync api? -> **By moving asynchrony to the Composition Root** . Where we could either use a Decorator, or in our case, Combine's **subscribe(on:...)** method.

We will do that next.

Inject asynchrony in the Composition Root so as not to block the Main Queue

So now, we've made all the following implementations synchronous:



But we are blocking the Main Queue, as we pointed out earlier, by using the **performAndWait** method from CoreData's framework.

Going to our Composition Root (SceneDelegate), and analyzing our **makeLocalImageLoaderWithRemoteFallback(url: URL) -> FeedImageDataLoader.Publisher** method:

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
    let localImageLoader = LocalFeedImageDataLoader(store: store)

    return localImageLoader
        .loadImageDataPublisher(from: url)
        .logCacheMisses(url: url, logger: logger)
        .fallback(to: { [httpClient, logger] in
            httpClient
                .getPublisher(url: url)
                .logError(url: url, logger: logger)
                .logElapsedTime(url: url, logger: logger)
                .tryMap(FeedImageDataMapper.map)
                .caching(to: localImageLoader, using: url)
        })
}
```

We can see that first we try to load from the cache, if we dont have anything there we try an API request, so the UI requests for the image in the main queue and we were dispatching it in the background queue, we need to to the same here. This can be done with a Decorator or using Combine/RxSwift using the **subscribe(on: Scheduler)** and subscribing to **DispatchQueue.global()** which implements the scheduler protocol.

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
    let localImageLoader = LocalFeedImageDataLoader(store: store)

    return localImageLoader
        .loadImageDataPublisher(from: url)
        .logCacheMisses(url: url, logger: logger)
        .fallback(to: { [httpClient, logger] in
            httpClient
                .getPublisher(url: url)
                .logError(url: url, logger: logger)
                .logElapsedTime(url: url, logger: logger)
                .tryMap(FeedImageDataMapper.map)
                .caching(to: localImageLoader, using: url)
        })
        .subscribe(on: DispatchQueue.global())
        .eraseToAnyPublisher()
}
```

Now if we run the app with the same checkpoint in the **performSync** method in our CoreDataFeedStore, we can see that we are now indeed in a background thread and not in the main thread:

so, the subscribe(on:) publisher, allows us to execute work in any provided scheduler, in this case we are using the **DispatchQueue.global()** which is a concurrente queue. Everytime we jump into the breakpoint we might see we are on a different thread, this is normal, this is because the global queue will choose whichever queue/thread is idle and run the operation concurrently. So if your starting implementation is thread safe, the global dispatch queue will do just fine, but if its not thread safe you can create your own background queue to run the operations.

Creating the Scheduler

So, we'll start by creating a scheduler in our Composition Root:

```
private lazy var scheduler = DispatchQueue(label:
"com.jmt.wot.FeedApp.FeedApp.infra.queue", qos: .userInitiated)
```

so now, we can use this scheduler, which is a background queue but serial, because, by default, DispatchQueue is serial.

We run again the app, and we can see that we are running in our infra queue, which is serial,

so we dont block the main thread, because we subscribe the upstream of operations with our own scheduler, so we are controlling all the threading from the composition root aswell.

So, if our components are not thread-safe, we should always execute our operations in a serial scheduler/queue. Therefore, where we cache the feed to the database, we also need to perform it in the scheduler, using the same scheduler (**Every operation in the store should be excecuted in the same serial scheduler if its not thread safe**)

```
private func makeLocalImageLoaderWithRemoteFallback(url: URL) ->
FeedImageDataLoader.Publisher {
    let localImageLoader = LocalFeedImageDataLoader(store: store)

    return localImageLoader
        .loadImageDataPublisher(from: url)
        .logCacheMisses(url: url, logger: logger)
        .fallback(to: { [httpClient, logger, scheduler] in
            httpClient
                .getPublisher(url: url)
                .logError(url: url, logger: logger)
                .logElapsedTime(url: url, logger: logger)
                .tryMap(FeedImageDataMapper.map)
                .caching(to: localImageLoader, using: url)
                .subscribe(on: scheduler)
                .eraseToAnyPublisher()
        })
        .subscribe(on: scheduler)
        .eraseToAnyPublisher()
}
```

Its not necessary to make the components thread safe if you control threading as a cross-cutting concern in the composition root, this way the implementations can be much simpler, **but since CoreData is thread-safe already (if we are using the `perform` API's (either `perform`(async) or `performAndWait`(sync))) , we can make a concurrent queue** :

```
private lazy var scheduler = DispatchQueue(label:
"com.jmt.wot.FeedApp.FeedApp.infra.queue",
                                           qos: .userInitiated,
                                           attributes: .concurrent)
```

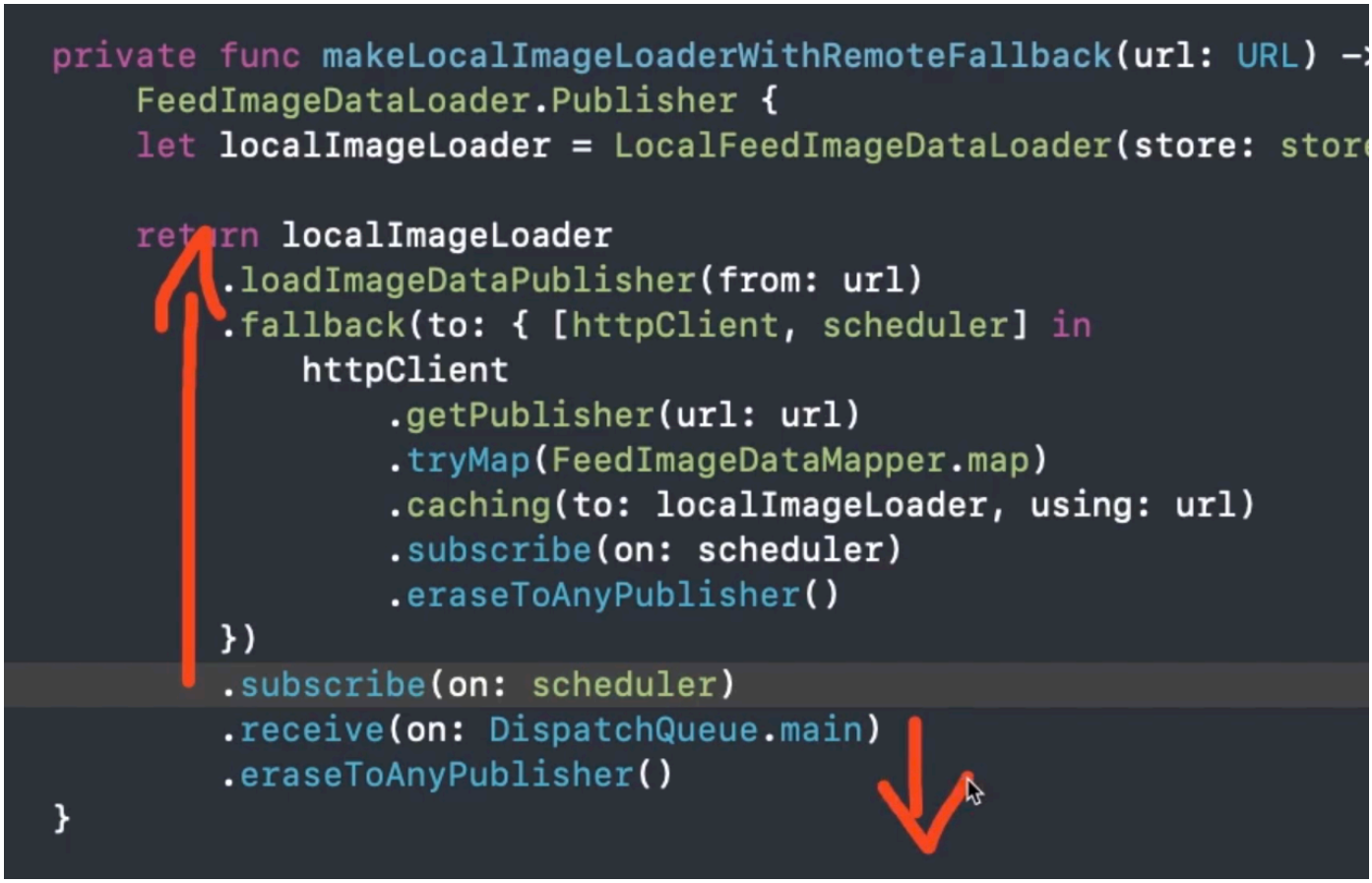
we run again, and we can clearly see that we are running concurrently in our infra queue

which concurrently will choose whatever thread is idle to run the operations.

So, the **subscribe(on:)** publisher has a counterpart, the **receive(on:)**, which we used already to dispatch to the main queue. So what we will do next is **receive(on:)** the main queue.

Subscribe(on:), affects the upstream messages, while **receive(on:)** the downstream messages:

So when we use the **subscribe(on: upstreamScheduler)** publisher, the upstream will be executed with that **upstreamScheduler** (arrow up), and with the **receive(on: downstreamScheduler)** publisher, the downstream (arrow down) will be affected by that **downstreamScheduler**



So all the operations pointed by the **up arrow** will be excecuted in the **scheduler** (background concurrence scheduler) but the results pointed by the **down arrow** will be dispatched to the **DispatchQueue.main** scheduler (main queue scheduler).

This way, we can subscribe on the background scheduler, with everything upwards from the subscribe(on:) will be excecuted in our concurrent **scheduler** and the results will be passed down and received in the **DispatchQueue.main** scheduler.

This was shown here for explanation purposes, but in this particular case we do not need the **receive(on:)** method here, because the **LoadResourcePresentationAdapter** already does it, it already dispatches on the main queue before updating the UI, with the **dispatchOnMainQueue() -> AnyPublisher<Output, Failure>** that we created in the **CombineHelpers** , that dispatches using the **receive(on:)** that affects the downstream.

We can see that the **dispatchOnMainQueue()** will affect all of the downstream of events that will all be on the main queue, regardless of what the upstream did. This is how we can control threading in the Composition Root. (We could use Decorators but using Combine we get this for free as an universal abstraction)

So, we can decouple all modules from the infra details, from async details so they are easier to develop mantain and test.

We run the App and it works. But there is a problem, the acceptance tests expect the code to run synchronously with the In-memory stub, but since we are dispatching this work in a background queue the acceptance test will fail because they dont wait for operations to finish, specifically the tests that were testing the rendered images with the data acquired from the requests vs the dummy image data.

We could also make the test wait, and thus making it slow, there exist some third party test frameworks able to wait for operations to finish asynchronously. It works but it adds friction/delay, makes the test slower.

Another way we could deal with that is to inject the scheduler, instead of always using the **concurrent scheduler** in our SceneDelegate, we can replace with an immediate scheduler during tests, so we use the **background scheduler** in production and during tests we use an **immediate scheduler**. So the idea is to inject a **scheduler** into the **SceneDelegate**, like we inject the **HttpClient** infrastructure details, because asynchrony is also an infrastructure detail!.

So, we pass an scheduler into the SceneDelegate init, that conforms to the protocol

```
convenience init(httpClient: HTTPClient, store: FeedStore & FeedImageDataStore,
    scheduler: Scheduler) {
    self.init()
    self.httpClient = httpClient
    self.store = store
}
```

the problem is that the protocol defines two associated types:

```
public protocol Scheduler {

    /// Describes an instant in time for this scheduler.
    associatedtype SchedulerTimeType : Strideable where Self.SchedulerTimeType.Stride :
        SchedulerTimeIntervalConvertible

    /// A type that defines options accepted by the scheduler.
    ///
    /// This type is freely definable by each `Scheduler`. Typically, operations that take a `Scheduler` parameter will
        also take `SchedulerOptions`.
    associatedtype SchedulerOptions

    /// This scheduler's definition of the current moment in time.
    var now: Self.SchedulerTimeType { get }
```

so we cannot pass it as an argument directly, or even hold it as a property, because the compiler needs to know the associated types.

```
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?

    private lazy var scheduler: Scheduler = DispatchQueue(   Protocol 'Scheduler' can only be used as...
        label: "com.essentialdeveloper.infra.queue",
        qos: .userInitiated,
        attributes: .concurrent)

    private lazy var httpClient: HTTPClient = {
        URLSessionHTTPClient(session: URLSession(configuration: .ephemeral))
    }()
```

So, a protocol with associated types is an incomplete protocol, you cannot use it directly, you cannot specify the associated types of a protocol in a property.

So, we could define the associated types into the **init** by using generics, but we cant hold a reference to it in a property. We could make the SceneDelegate define the associated types for the Scheduler, the problem is that SceneDelegate cannot have associated types because it is an NSObject Class and it wouldn't be instantiated properly by UIKit.

So, we cannot pass the scheduler as a protocol, but we can pass the concrete protocol implementations like a **DispatchQueue** as a scheduler (because it implements Scheduler), but by passing the concrete DispatchQueue scheduler, we are coupling the client with the DispatchQueue Scheduler only, but the immediate scheduler we created earlier, is not a DispatchQueue, so because we are coupled with DispatchQueue in this case, we cannot pass another type that is also a Scheduler. So we would like to pass AnyScheduler and we cannot do that with a protocol.

This problem is similar to the Publisher protocol, which has associated types, but we cannot pass publishers around like protocols due to the associated types, but we can pass concrete implementations of the protocol like the Publishers.Map , because we can define the generic types with the concrete type, but again, by passing the map publisher we couple clients with the map publisher. What if we want to perform extra mappings like tryMap or subscribe or so on?.

So the solution to this problem is **Type Erasure**. For example to decouple the clients from these concrete publishers Apple provides us with the **AnyPublisher** type erasure

```
private func makeRemoteCommentsLoader(url: URL) -> () -> AnyPublisher<[ImageComment],
    Error> {
    return { [httpClient] in
        return httpClient
            .getPublisher(url: url)
            .tryMap(ImageCommentsMapper.map)
            .eraseToAnyPublisher()
    }
}
```

so here we create a chain of publishers but then we erase that chain, the types, with the AnyPublisher type erasure, so we can decouple the clients from the concrete types and still hold it passed as a parameter and hold it as a property.

Implementing an AnyScheduler Type Erasure

However, for some reason Apple doesnt provide us with **AnyScheduler** , which makes it very hard to pass schedulers as parameters or decouple the clients from concrete schedulers. But we can create our own **AnyScheduler** type erasure. To do this, lets do it following the same idea as in Apple's implementation of **AnyPublisher** :

```
///
/// You can use Combine's ``Publisher/eraseToAnyPublisher()`` operator to wrap a publisher with
    ``AnyPublisher``.
@available(macOS 10.15, iOS 13.0, tvOS 13.0, watchOS 6.0, *)
@frozen public struct AnyPublisher<Output, Failure> : CustomStringConvertible,
    CustomPlaygroundDisplayConvertible where Failure : Error {

    /// A textual representation of this instance.
    ///
    /// Calling this property directly is discouraged. Instead, convert an
    /// instance of any type to a string by using the `String(describing:)`
    /// initializer. This initializer works with any type, and uses the custom
    /// `description` property for types that conform to
    /// `CustomStringConvertible`:
    ///
    ///     struct Point: CustomStringConvertible {
    ///         let x: Int, y: Int
    ///
    ///         var description: String {
    ///             return "(\(x), \(y))"
    ///         }
    ///     }
    ///
    ///     let p = Point(x: 21, y: 30)
    ///     let s = String(describing: p)
    ///     print(s)
    ///     // Prints "(21, 30)"
    ///
    /// The conversion of `p` to a string in the assignment to `s` uses the
    /// `Point` type's `description` property.
    public var description: String { get }

    /// A custom playground description for this instance.
    public var playgroundDescription: Any { get }
```

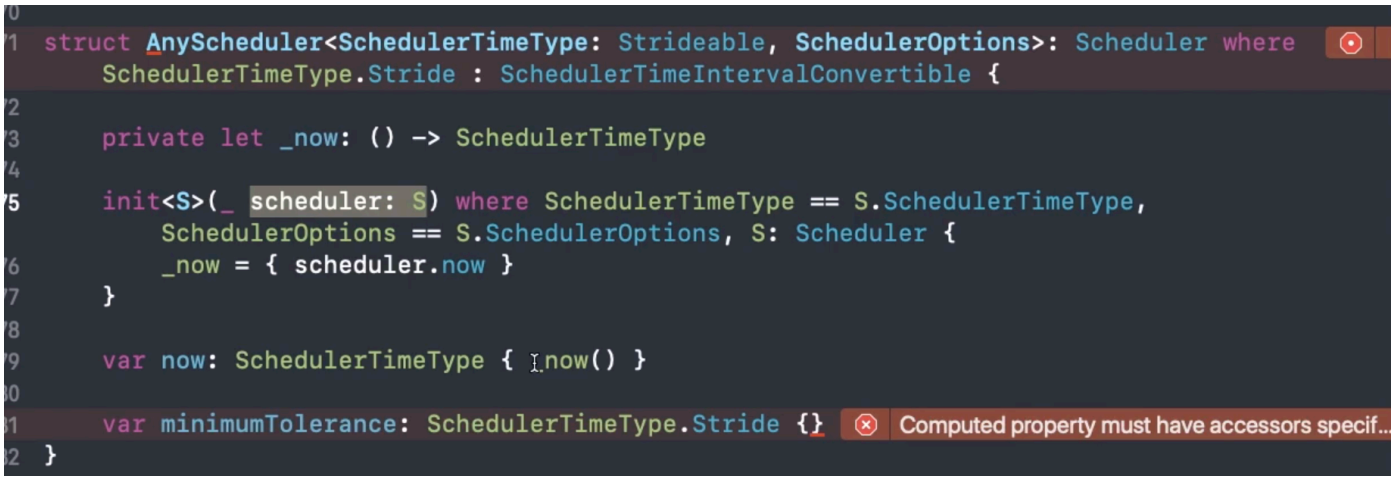
So in AnyPublisher, it defines the generic associated types, Output, Failure that matches the associated types in the protocol, and the AnyPublisher also implements the Publisher protocol, because the AnyPublisher will wrap a Publisher and AnyPublisher is also a Publisher thats why you can pass it around and chain it with other publishers

```
@available(macOS 10.15, iOS 13.0, tvOS 13.0, watchOS 6.0, *)
extension AnyPublisher : Publisher {

    /// Attaches the specified subscriber to this publisher.
    ///
    /// Implementations of ``Publisher`` must implement this method.
    ///
    /// The provided implementation of ``Publisher/subscribe(_:)-4u8kn`` calls this method.
    ///
    /// - Parameter subscriber: The subscriber to attach to this ``Publisher``, after which it can receive values.
    @inlinable public func receive<S>(subscriber: S) where Output == S.Input, Failure ==
        S.Failure, S : Subscriber
```

So, we will follow the same idea here, we will start by literally copying and pasting the AnyPublisher type erasure from apple's docs and start modifying it.

The idea is that we are going to wrap any scheduler provided in the init and it will erase its type behind the AnyScheduler, but internally it will delegate messages to the provided scheduler. This pattern is very similar to the decorator pattern.

So one by one we are going to add all the properties required by the protocol. We have a problem, though that is that we cannot hold any references to the Scheduler passed in the init, since the compiler doesnt know about its type, but we can delegate those messages using closures. For example to delegate the **now SchedulerTimeType**, we would do it like this:



This is a challenge with strongly typed languages, you have to satisfy the compiler, and if the compiler doesnt understand the types you need to come up with these solutions.

Doing the same logic for all the required properties we have:

```
struct AnyScheduler<SchedulerTimeType: Strideable, SchedulerOptions>: Scheduler where
SchedulerTimeType.Stride: SchedulerTimeIntervalConvertible {
    private let _now: () -> SchedulerTimeType
    private let _minimumTolerance: () -> SchedulerTimeType.Stride
    private let _schedule: (SchedulerOptions?, @escaping () -> Void) -> Void
    private let _scheduleAfter: (SchedulerTimeType, SchedulerTimeType.Stride,
SchedulerOptions?, @escaping () -> Void) -> Void
    private let _scheduleAfterInterval: (SchedulerTimeType, SchedulerTimeType.Stride,
SchedulerTimeType.Stride, SchedulerOptions?, @escaping () -> Void) -> Cancellable

    init<S>(_ scheduler: S) where SchedulerTimeType == S.SchedulerTimeType,
SchedulerOptions == S.SchedulerOptions, S: Scheduler {
        _now = { scheduler.now }
        _minimumTolerance = { scheduler.minimumTolerance }
        _schedule = scheduler.schedule(options:_:)
        _scheduleAfter = scheduler.schedule(after:tolerance:options:_:)
        _scheduleAfterInterval =
scheduler.schedule(after:interval:tolerance:options:_:)
    }

    var now: SchedulerTimeType { _now() }

    var minimumTolerance: SchedulerTimeType.Stride { _minimumTolerance() }

    func schedule(options: SchedulerOptions?, _ action: @escaping () -> Void) {
        _schedule(options, action)
    }
```

```
    func schedule(after date: SchedulerTimeType, tolerance: SchedulerTimeType.Stride,
options: SchedulerOptions?, _ action: @escaping () -> Void) {
        _scheduleAfter(date, tolerance, options, action)
    }

    func schedule(after date: SchedulerTimeType, interval: SchedulerTimeType.Stride,
tolerance: SchedulerTimeType.Stride, options: SchedulerOptions?, _ action: @escaping ()
-> Void) -> Cancellable {
        _scheduleAfterInterval(date, interval, tolerance, options, action)
    }
}
```

In the same way as we eraseToAnyPublisher, when we use type Erasure for AnyPublisher, we are going to create an API to be able to eraseToAnyScheduler, when we use type erasure for AnyScheduler, which will return an AnyScheduler, but mantaining the generic types.

```
extension Scheduler {
    func eraseToAnyScheduler() -> AnyScheduler<SchedulerTimeType, SchedulerOptions> {
        AnyScheduler(self)
    }
}
```

Which basically is wrapping the scheduler into an AnyScheduler.

So now we can erase our scheduler in the Composition Root:

```
  private lazy var scheduler: AnyDispatchQueueScheduler = DispatchQueue(
    label: "com.jmt.wot.FeedApp.FeedApp.infra.queue",
    qos: .userInitiated,
    attributes: .concurrent
  ).eraseToAnyScheduler()
```

So that its type is now **AnyScheduler<DispatchQueue.SchedulerTimeType, DispatchQueue.SchedulerOptions>**

which is an AnyScheduler with the DispatchQueue associated types (since the type was originally a DispatchQueue object).

It also happens that our **ImmediateWhenOnMainQueueScheduler: Scheduler** uses the same types:

```
struct ImmediateWhenOnMainQueueScheduler: Scheduler {
    typealias SchedulerTimeType = DispatchQueue.SchedulerTimeType
    typealias SchedulerOptions = DispatchQueue.SchedulerOptions

    var now: SchedulerTimeType {
        DispatchQueue.main.now
    }

    var minimumTolerance: SchedulerTimeType.Stride {
        DispatchQueue.main.minimumTolerance
    }

    static let shared = Self()

    private static let key = DispatchSpecificKey<UInt8>()
    private static let value = UInt8.max

    private init() {
        DispatchQueue.main.setSpecific(key: Self.key, value: Self.value)
```

thus, we can inject now a scheduler into the Composition Root's init, and override it during tests:

```
convenience init(httpClient: HTTPClient, store: FeedStore & FeedImageDataStore,
scheduler: AnyDispatchQueueScheduler) {
        self.init()
        self.httpClient = httpClient
        self.store = store
        self.scheduler = scheduler
    }
```

Where the **AnyDispatchQueueScheduler** is a typealias we made to simplify the reading

```
typealias AnyDispatchQueueScheduler = AnyScheduler<DispatchQueue.SchedulerTimeType,
DispatchQueue.SchedulerOptions>
```

And since as we noted earlier,

```
extension AnyDispatchQueueScheduler {
    static var immediateOnMainQueue: Self {
        DispatchQueue.immediateWhenOnMainQueueScheduler.eraseToAnyScheduler()
    }
}
```

It also happens that our **ImmediateWhenOnMainQueueScheduler: Scheduler** uses the same types, therefore we create an extension to **AnyDispatchQueueScheduler** typealias that provides a static property that provides us with our **immediateOnMainQueue** erased to any scheduler. This immediate scheduler is the one we will provide during tests.

All our tests are running.

The next step is to make the rest of our async API abstractions synchronous like the and the .

We dont need to make all our API's synchronous but it helps that they don't leak implementations that way, and we can always compose, in our Composition Root. It also helps that with less completion blocks it means we have to capture self much less avoiding potential memory retention cycles and arrow code.

Synchronous FeedStore and FeedCache

Applying the same logic that we applied for the previous API abstractions, we are going to make both FeedStore and FeedCache API's synchronous.

Same as before, the first thing we will do is add the desired synchronous API methods to our abstraction and at the same time deprecate the old asynchronous API methods.

The next step is to add default implementations in our new Sync API's that use the pre-existing async apis, in the meanwhile of our migration, aswell as default empty async implementations of our old async apis.

```
public protocol FeedStore {
    func deleteCachedFeed() throws
    func insert(_ feed: [LocalFeedImage], timestamp: Date) throws
    func retrieve() throws -> CachedFeed?

    ...
    ...
    ...

    @available(*, deprecated)
    func deleteCachedFeed(completion: @escaping DeletionCompletion)

    @available(*, deprecated)
    func insert(_ feed: [LocalFeedImage], timestamp: Date, completion: @escaping
InsertionCompletion)

    @available(*, deprecated)
    func retrieve(completion: @escaping RetrievalCompletion)
}
```

```
public extension FeedStore {
```

```
    func deleteCachedFeed() throws {
        let group = DispatchGroup()
        group.enter()
        var result: DeletionResult!
        deleteCachedFeed {
            result = $0
            group.leave()
        }
        group.wait()
        return try result.get()
    }

    func insert(_ feed: [LocalFeedImage], timestamp: Date) throws {
        let group = DispatchGroup()
        group.enter()
        var result: InsertionResult!
        insert(feed, timestamp: timestamp) {
            result = $0
            group.leave()
        }
        group.wait()
        return try result.get()
    }

    func retrieve() throws -> CachedFeed? {
        let group = DispatchGroup()
        group.enter()
        var result: RetrievalResult!
        retrieve {
            result = $0
            group.leave()
        }
        group.wait()
        return try result.get()
    }

    func deleteCachedFeed(completion: @escaping DeletionCompletion) {}
    func insert(_ feed: [LocalFeedImage], timestamp: Date, completion: @escaping
InsertionCompletion) {}
    func retrieve(completion: @escaping RetrievalCompletion) {}
}
```

In the exact same way as we did with the previous API's, we modify the existing tests to take into account synchronicity instead of asynchronicity.

Making the LocalFeedLoader: FeedCache synchronous

Now we have to migrate our **LocalFeedLoader: FeedCache** extension to conform to the new synchronous API's which means going from :

```
extension LocalFeedLoader: FeedCache {
    public typealias SaveResult = FeedCache.Result

    public func save(_ feed: [FeedImage], completion: @escaping (SaveResult) -> Void) {
        store.deleteCachedFeed { [weak self] deletionResult in
            guard let self = self else { return }

            switch deletionResult {
            case.success:
                self.cache(feed, with: completion)
            case .failure(let error):
                completion(.failure(error))
            }
        }
    }

    private func cache(_ feed: [FeedImage], with completion: @escaping (SaveResult) ->
Void) {
        store.insert(feed.toLocal(), timestamp: currentDate()) { [weak self] error in
            guard self != nil else { return }

            completion(error)
        }
    }
}
```

To :

```
extension LocalFeedLoader: FeedCache {
    public typealias SaveResult = FeedCache.Result

    public func save(_ feed: [FeedImage], completion: @escaping (SaveResult) -> Void) {
        completion(SaveResult {
            try store.deleteCachedFeed()
            try store.insert(feed.toLocal(), timestamp: currentDate())
        })
    }
}
```

As we can see we have drastically simplified our code by making it synchronous. Its possible to see we have reduced our code one indentation level and its much more easy to read. We have removed the switch statement, and the closure will simply return a .**failure** if it finds any of its **try** statements throwing, if it doesnt, then it completes with **success** .

Making the LocalFeedLoader 'load(completion:)' method asynchronous

Now we will change our **load(completion:)** method so that it uses our new sync api's. Changing it from:

```
extension LocalFeedLoader {
    public typealias LoadResult = Swift.Result<[FeedImage], Error>

    public func load(completion: @escaping (LoadResult) -> Void) {
        store.retrieve { [weak self] result in
            guard let self = self else { return }

            switch result {
            case .failure(let error):
                completion(.failure(error))

            case .success(.some(let cache)) where
FeedCachePolicy.validate(cache.timestamp, against: self.currentDate()):
                completion(.success(cache.feed.toModels()))

            case .success:
                completion(.success([]))
            }
        }
    }
}
```

To:

```
extension LocalFeedLoader {
    public typealias LoadResult = Swift.Result<[FeedImage], Error>

    public func load(completion: @escaping (LoadResult) -> Void) {
        completion(LoadResult {
            if let cache = try store.retrieve(),
FeedCachePolicy.validate(cache.timestamp, against: currentDate()) {
                return cache.feed.toModels()
            }
            return []
        })
    }
}
```

Same note as before, we are getting rid of our arrow code and switches, in favour of cleaner code with less completion blocks. In the same fashion, we wrap our result into a **Result** block where we check if the info is cached and then we validate that it respects the cache policy regarding its date. In the case of the **store.retrieve()** throwing, the Result block simply transforms it into a **.failure** result.

The next step is to migrate our **validateCache(completion:)** method to use our new api's. This method is has similarities to the previous load method in the way that it's modified. Since our new api's dont have completion blocks, we dont need to switch the result, and thus getting rid of levels of arrow code.

Previous:

```
extension LocalFeedLoader {
    public typealias ValidationResult = Result<Void, Error>

    public func validateCache(completion: @escaping (ValidationResult) -> Void) {
        store.retrieve { [weak self] result in
            guard let self = self else { return }

            switch result {
            case .failure:
                self.store.deleteCachedFeed(completion: completion)

            case let .success(.some(cache)) where
!FeedCachePolicy.validate(cache.timestamp, against: self.currentDate()):
                self.store.deleteCachedFeed(completion: completion)

            case .success:
                completion(.success(()))
            }
        }
    }
}
```

New Api:

```
extension LocalFeedLoader {
    public typealias ValidationResult = Result<Void, Error>

    private struct InvalidCache: Error {}

    public func validateCache(completion: @escaping (ValidationResult) -> Void) {
        completion(ValidationResult {
            do {
                if let cache = try store.retrieve(),
!FeedCachePolicy.validate(cache.timestamp, against: currentDate()) {
                    throw InvalidCache()
                }
            } catch {
                try store.deleteCachedFeed()
```

```
            }
        })
    }
}
```

As we analyze the previous code we can also see that we have one less **self** to capture, meaning we are actively reducing the chances of missing a memory cycle by mistake. We can see that the idea here is that we are going to check if the cache exists, and if it respects the cache policy. in case of success nothing happens, but in case of failure we throw an **InvalidCache** error, that is caught by our **catch** block, and which attempts to delete the outdated cache. We are prepared to have this invalid cache state, thats why if it happens we dont get a **.failure** as a result, but a **.success**. The way to get a **.failure** is if the deletion of an outdated cache fails.

Make FeedCache Sync

Next step is to make our api abstraction synchronous. This abstraction only has one method, **save(_feed: completion)** and a Result typealias.

```
//Previously
public protocol FeedCache {
    typealias Result = Swift.Result<Void, Error>

    func save(_ feed: [FeedImage], completion: @escaping (Result) -> Void)
}

//New sync api
public protocol FeedCache {
    func save(_ feed: [FeedImage]) throws
}
```

Since we dont have a completion closure anymore, we dont need to have the Result typealias.

Now we have to modify our previously modified **LocalFeedLoader: FeedCache** extension from this:

```
extension LocalFeedLoader: FeedCache {
    public typealias SaveResult = FeedCache.Result

    public func save(_ feed: [FeedImage], completion: @escaping (SaveResult) -> Void) {
        completion(SaveResult {
            try store.deleteCachedFeed()
            try store.insert(feed.toLocal(), timestamp: currentDate())
        })
    }

}
```

to this:

```
extension LocalFeedLoader: FeedCache {
    public func save(_ feed: [FeedImage]) throws {
        try store.deleteCachedFeed()
        try store.insert(feed.toLocal(), timestamp: currentDate())
    }
}
```

As we can see, we dont need to have a **SaveResult** typealias anymore since we dont have a completion closure anymore, we also dont need to wrap our body in a Result block. Our new **save(_feed:) throws** methods just throws when there is an exception, meaning we can simply call the body of our method, containing the **try** statements, without further worries.

We also change all the tests in the same way we did for our previous case adding necessary do-catch blocks where needed and making the logic synchronous.

Next step is to refactor the **LocalFeedLoader load** method :

```
//Previous (using async)
extension LocalFeedLoader {
    public typealias LoadResult = Swift.Result<[FeedImage], Error>

    public func load(completion: @escaping (LoadResult) -> Void) {
        completion(LoadResult {
            if let cache = try store.retrieve(),
FeedCachePolicy.validate(cache.timestamp, against: currentDate()) {
                return cache.feed.toModels()
            }
            return []
        })
    }
}

//Actual (sync)

extension LocalFeedLoader {
    public func load() throws -> [FeedImage] {
        if let cache = try store.retrieve(), FeedCachePolicy.validate(cache.timestamp,
against: currentDate()) {
            return cache.feed.toModels()
        }
        return []
    }
}
```

With this change, we will always either get an array of FeedImage (either empty or with the cached feedimages) or it will throw an error. This way we get rid of the completion blocks and the arrow code.

Its also necessary that we make a small change in our **CombineHelpers**, we need to change our **loadPublisher() -> Publisher** method because it uses the old version (async) of our **LocalFeedLoader.load** method.

We go from:

```
public extension LocalFeedLoader {
    typealias Publisher = AnyPublisher<[FeedImage], Error>

    func loadPublisher() -> Publisher {
        Deferred {
            Future(self.load)
        }
        .eraseToAnyPublisher()
    }
}
```

to:

```
public extension LocalFeedLoader {
    typealias Publisher = AnyPublisher<[FeedImage], Error>

    func loadPublisher() -> Publisher {
        Deferred {
            Future { completion in
                completion(Result{ try self.load() })
            }
        }
        .eraseToAnyPublisher()
    }
}
```

Since our load used to have a completion closure, but not it doesnt, we will have to wrap our new **load** function in a completion block with a Result, type.

Making LocalFeedLoader.validateCache synchronous

Continuing the migration from async api's to the new sync ones, its time to migrate LocalFeedLoader's **validateCache** method.

Originally the method returned a completion closure containing a **ValidationResult** that consisted in a Result<Void, Error>, the new api, however, doesn't need a completion closure since its synchronous, it will either succeed or throw.

The original **validateCache** was this:

```
extension LocalFeedLoader {
    public typealias ValidationResult = Result<Void, Error>
```

```
    private struct InvalidCache: Error {}

    public func validateCache(completion: @escaping (ValidationResult) -> Void) {
        completion(ValidationResult {
            do {
                if let cache = try store.retrieve(),
!FeedCachePolicy.validate(cache.timestamp, against: currentDate()) {
                    throw InvalidCache()
                }
            } catch {
                try store.deleteCachedFeed()
            }
        })
    }
}
```

The synchronous version is:

```
extension LocalFeedLoader {
    private struct InvalidCache: Error {}

    public func validateCache() throws {
        do {
            if let cache = try store.retrieve(),
!FeedCachePolicy.validate(cache.timestamp, against: currentDate()) {
                throw InvalidCache()
            }
        } catch {
            try store.deleteCachedFeed()
        }
    }
}
```

Previously, we had updated this method to comply with our new synch apis from the Cache, this time we are doing another round of refactoring to bring this method completely synchronous, since we dont need completion handlers anymore because we have migrated our apis to sync. As stated earlier, the **ValidationResult** is no longer needed, and we can see that we are losing one more level of indentation and arrow code by getting rid of the completion wrapper.

This change must also be accompanied by a refactor to the **SceneDelegate**, in the **sceneWillResignActive** delegate method, where we were executing a cache validation before the app was closed killed or sent to background. An adjustment is added to reflect the api changes:

```
//Before
```

```
func sceneWillResignActive(_ scene: UIScene) {
        localFeedLoader.validateCache { _ in }
}

//After

func sceneWillResignActive(_ scene: UIScene) {
    do {
        try localFeedLoader.validateCache()
    } catch {
        logger.error("Failed to validate cache with error: \
(error.localizedDescription)")
    }
}
```

Make CoreData FeedStore implementation sync

Following step is to migrate our **CoreData FeedStore** implementation from async api, to sync api. For this, we are going to migrate the three **FeedStore** Api implementations made on the **CoreDataFeedStore**, **retrieve, insert and deleteCachedFeed**:

Retrieve

```
//Async API
public func retrieve(completion: @escaping RetrievalCompletion) {
    performAsync { context in
        completion(Result {
            try ManagedCache.find(in: context).map {
                CachedFeed(feed: $0.localFeed, timestamp: $0.timestamp)
            }
        })
    }
}

//Sync API
public func retrieve() throws -> CachedFeed? {
    try performSync { context in
        Result {
            try ManagedCache.find(in: context).map {
                CachedFeed(feed: $0.localFeed, timestamp: $0.timestamp)
            }
        }
    }
}
```

As we did previously with the migration of our previous API's, we are going to migrate from using the **performAsync** to the **performSync** CoreData context methods. Since we don't have completion handlers, we can reduce our code indentation one level, getting rid of the completion block. If this operation succeeds we will return the CachedFeed, and if it fails we will return nil.

Insert

```
//Async API
public func insert(_ feed: [LocalFeedImage], timestamp: Date, completion: @escaping
InsertionCompletion) {
    performAsync { context in
        completion(Result {
            let managedCache = try ManagedCache.newUniqueInstance(in: context)
            managedCache.timestamp = timestamp
            managedCache.feed = ManagedFeedImage.images(from: feed, in: context)
            try context.save()
        })
    }
}

//Sync
public func insert(_ feed: [LocalFeedImage], timestamp: Date) throws {
    try performSync { context in
        Result {
            let managedCache = try ManagedCache.newUniqueInstance(in: context)
            managedCache.timestamp = timestamp
            managedCache.feed = ManagedFeedImage.images(from: feed, in: context)
            try context.save()
        }
    }
}
```

DeleteCachedFeed

```
//Async
public func deleteCachedFeed(completion: @escaping DeletionCompletion) {
    performAsync { context in
        completion(Result {
            try ManagedCache.deleteCache(in: context)
        })
    }
}
//Sync
public func deleteCachedFeed() throws {
    try performSync { context in
        Result {
            try ManagedCache.deleteCache(in: context)
```

```
        }
    }
}
```

NullStore

We will also adapt our NullStore to work with the new sync apis, which since we dont have completion blocks anymore, can just be empty methods or return nil:

```
extension NullStore: FeedStore {
    func deleteCachedFeed() throws {}

    func insert(_ feed: [LocalFeedImage], timestamp: Date) throws {}

    func retrieve() throws -> CachedFeed? { .none }
}
```

InMemoryFeedStore

```
extension InMemoryFeedStore: FeedStore {
    func deleteCachedFeed() throws {
        feedCache = nil
    }

    func insert(_ feed: [LocalFeedImage], timestamp: Date) throws {
        feedCache = CachedFeed(feed: feed, timestamp: timestamp)
    }

    func retrieve() throws -> CachedFeed? {
        feedCache
    }
}
```

Cleaning up

Now that the new sync api's are in play, its time to remove the old API's and result types that arent used anymore from the , the final code is:

```
public typealias CachedFeed = (feed: [LocalFeedImage], timestamp: Date)

public protocol FeedStore {
    func deleteCachedFeed() throws
    func insert(_ feed: [LocalFeedImage], timestamp: Date) throws
    func retrieve() throws -> CachedFeed?
}
```

We delete the old api's aswell as the default implementations and the deprecated ones.

Subscribe upstream store subscriptions in a background queue

In the same way as we did earlier in other parts of the project, we will make the **makeRemoteFeedLoaderWithFallback** , and **makeRemoteLoadMoreLoader** subscribe upstream in a background thread, in the **scheduler** we created previously in this project.

```
private func makeRemoteFeedLoaderWithLocalFallback() ->
AnyPublisher<Paginated<FeedImage>, Error> {
    makeRemoteFeedLoader()
        .caching(to: localFeedLoader)
        .fallback(to: localFeedLoader.loadPublisher)
        .map(makeFirstPage)
        .subscribe(on: scheduler)
        .eraseToAnyPublisher()
}

private func makeRemoteLoadMoreLoader(last: FeedImage?) ->
AnyPublisher<Paginated<FeedImage>, Error> {
    localFeedLoader.loadPublisher()
        .zip(makeRemoteFeedLoader(after: last))
        .map { (cachedItems, newItems) in
            (cachedItems + newItems, newItems.last)
        }
        .map(makePage)
        .caching(to: localFeedLoader)
        .subscribe(on: scheduler)
        .eraseToAnyPublisher()
}
```