

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 implementation, 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"
      }
    },
    ...
  ]
}
```

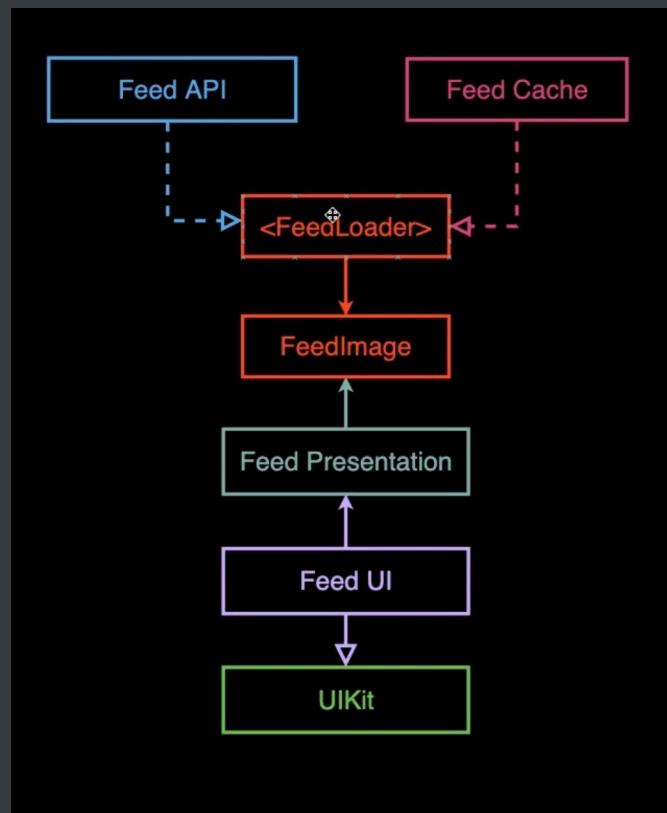
Feed Image Comment

| Property | Type |
|-------------------------|-----------------------|
| <code>id</code> | UUID |
| <code>message</code> | String |
| <code>created_at</code> | Date (ISO8601 String) |
| <code>author</code> | CommentAuthorObject |

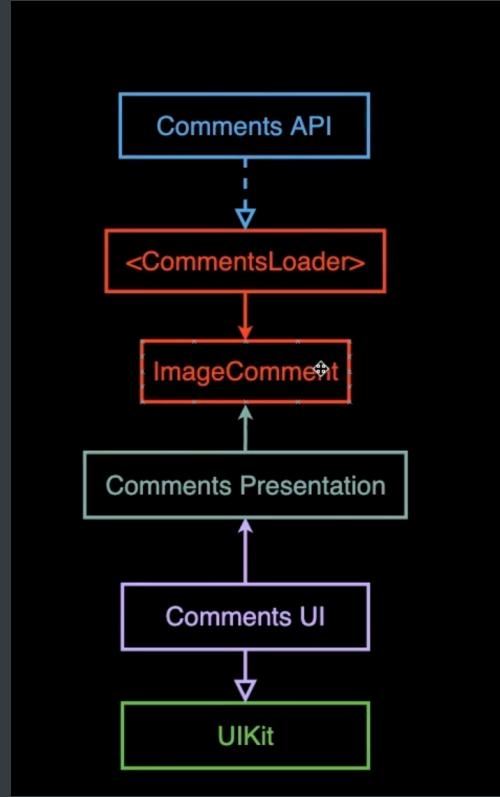
Feed Image Comment Author

| Property | Type |
|----------|--------|
| username | String |

Architecture for Base App

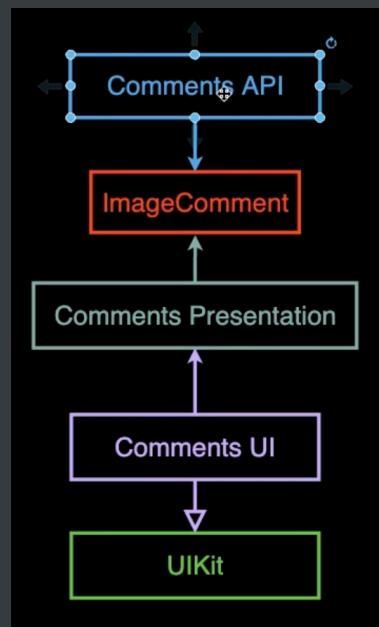


First Approach to the architecture



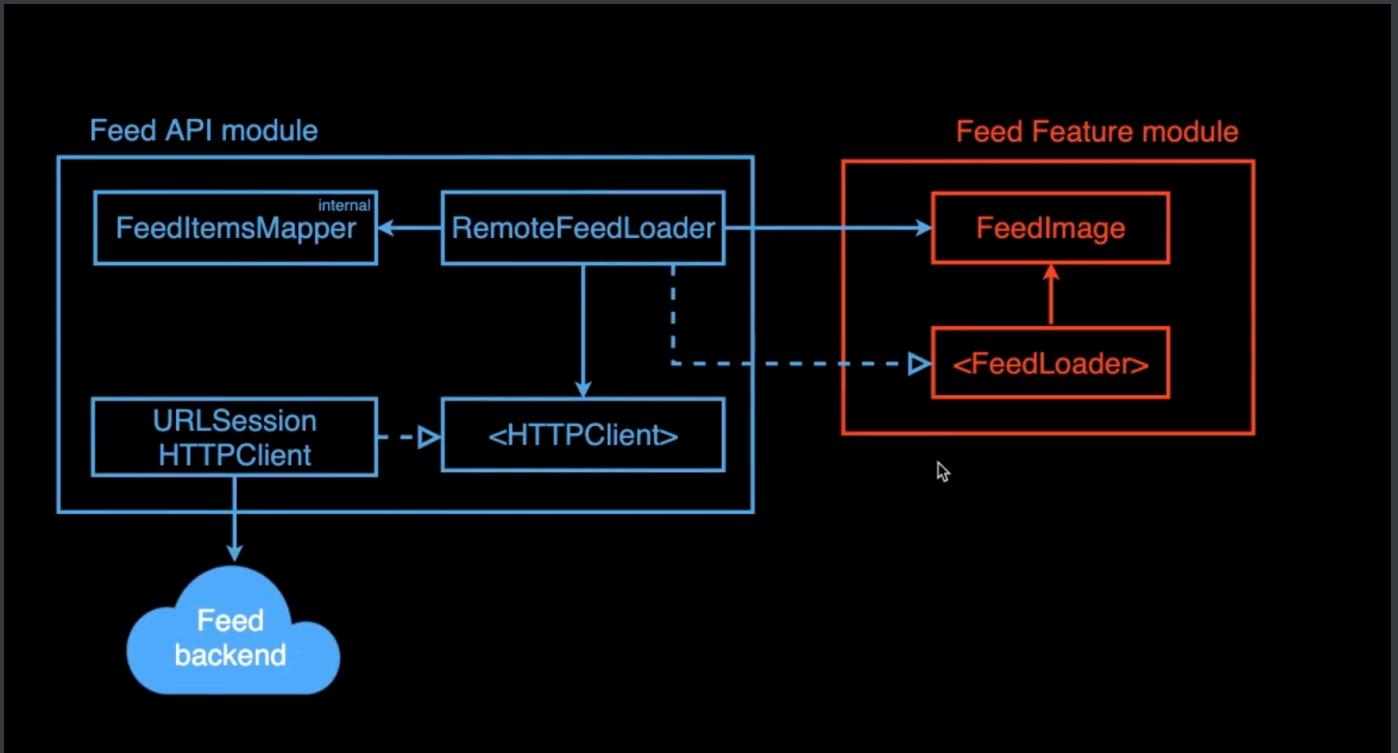
In Reality we dont need to have the **Comments Loader Protocol** since we wont be caching, usually protocols/interfaces are good for when we have more than one use of it. Its always good to simplify things when possible.

Final Diagram

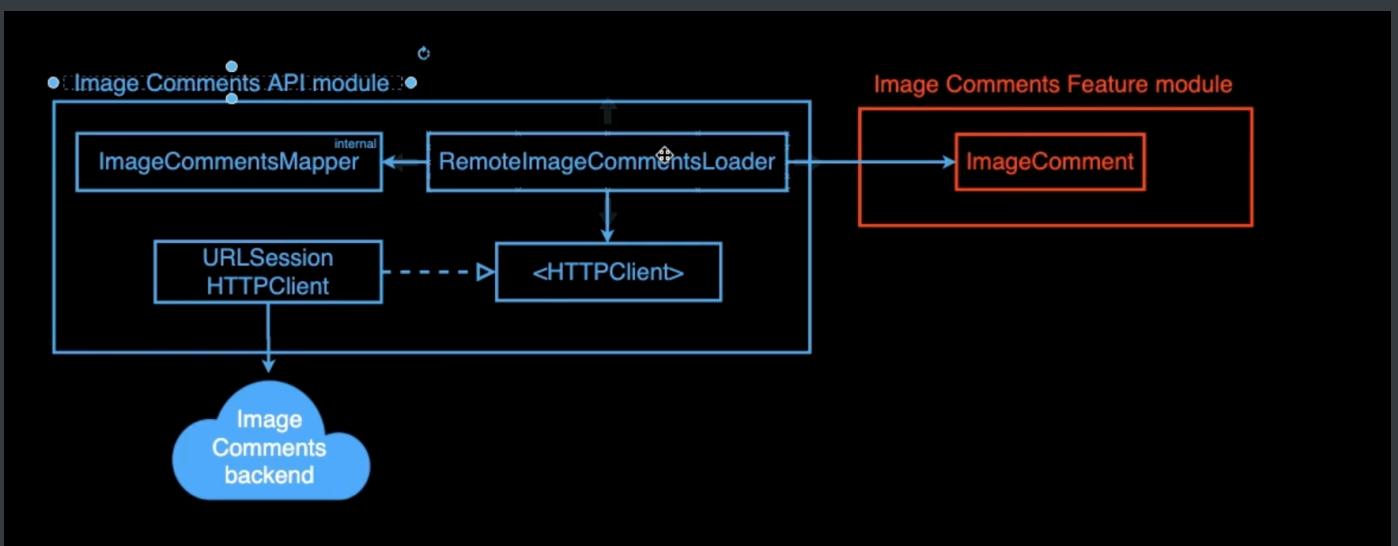


How can we create new features without duplicating code/ duplicating HTTPClients?

Starting with the architecture for the already existing FeedModule Feature:



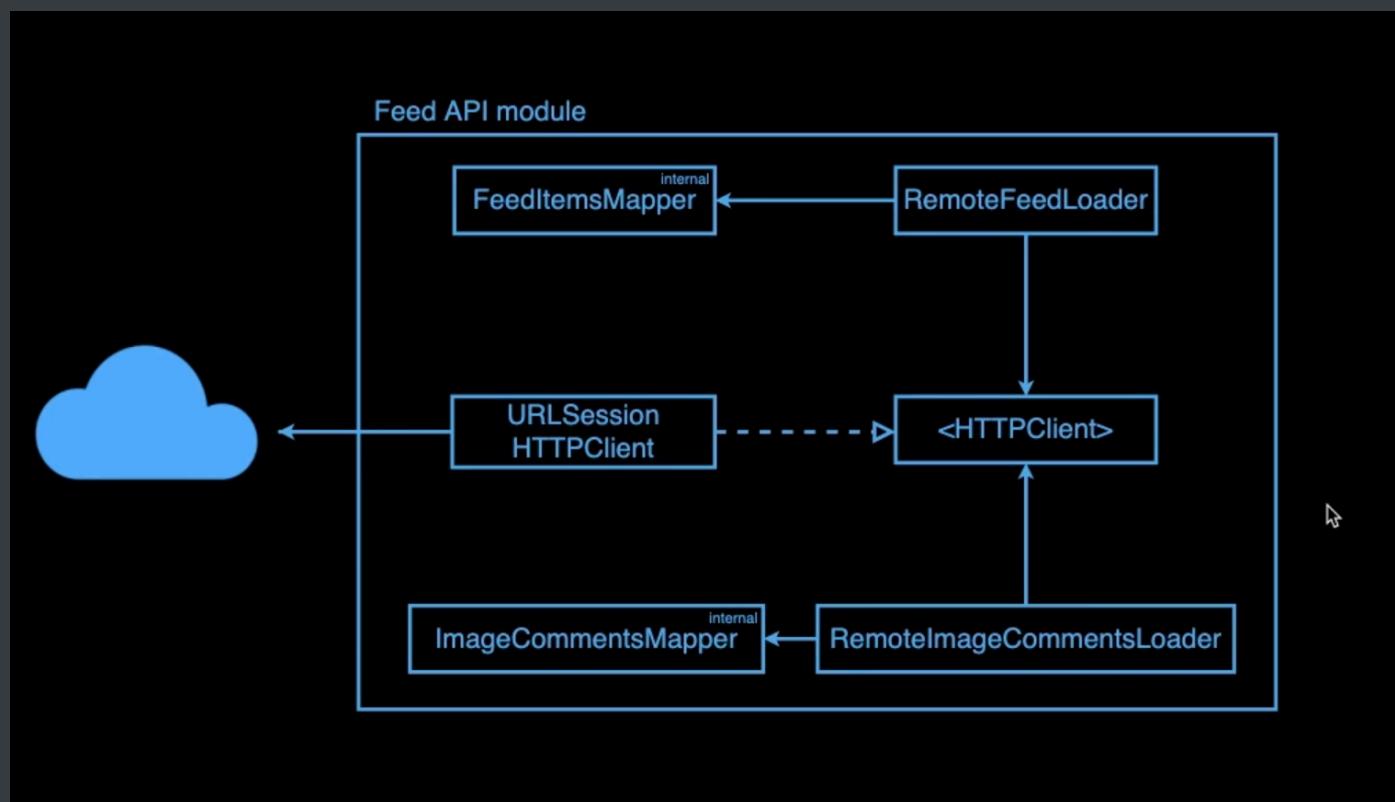
We can follow the same design to approach the Comments section:



Upsides: These both modules are decoupled. Since we don't need multiple strategies, we don't 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 they can therefore share the same types:

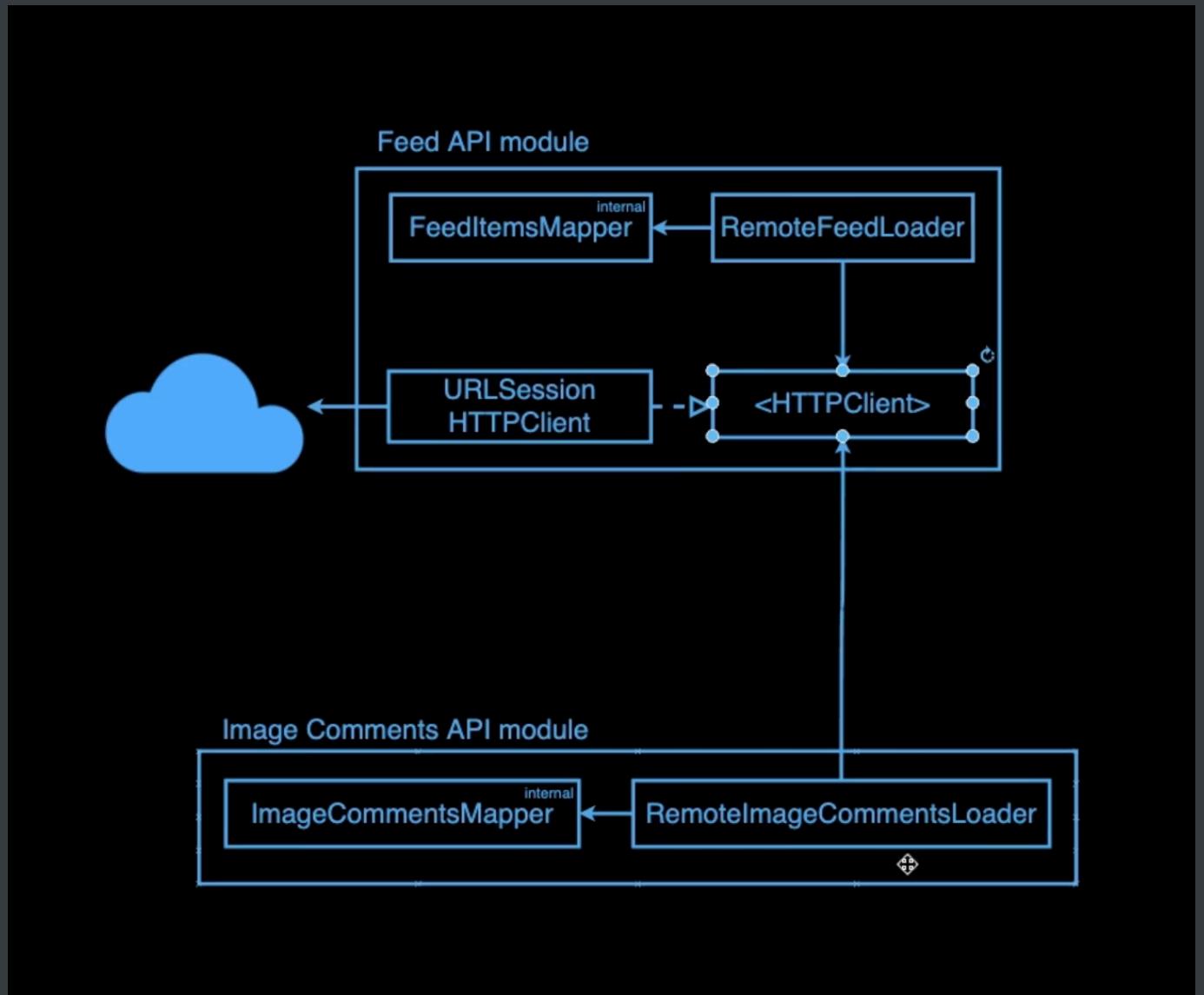


But then we lose the modularity that we want. Nonetheless it's a good starting point.

- 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 `ImageComments API` in some other place?

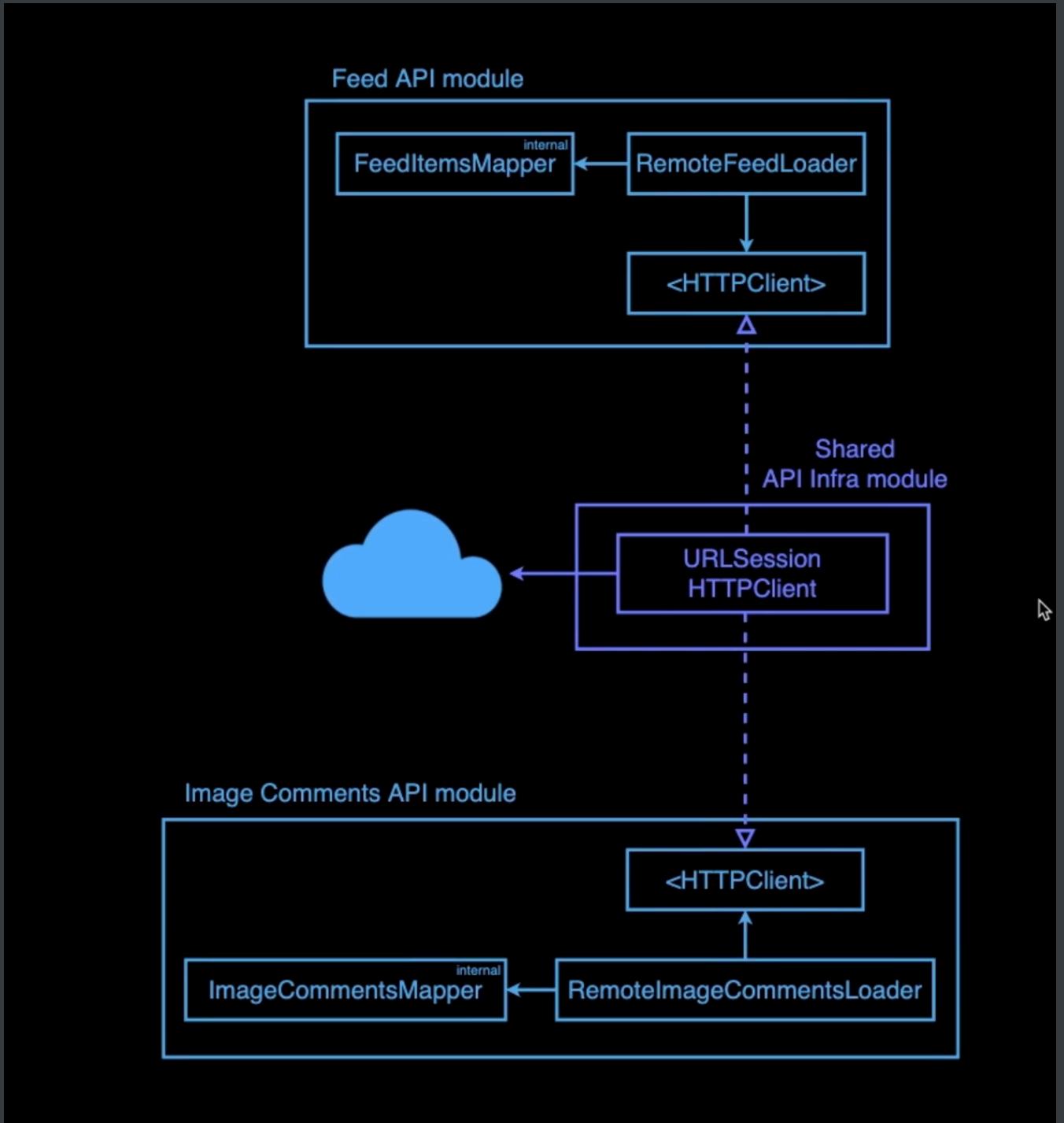
Well, we could move it into another module:



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

First idea:

We create **two** HTTP client protocols:

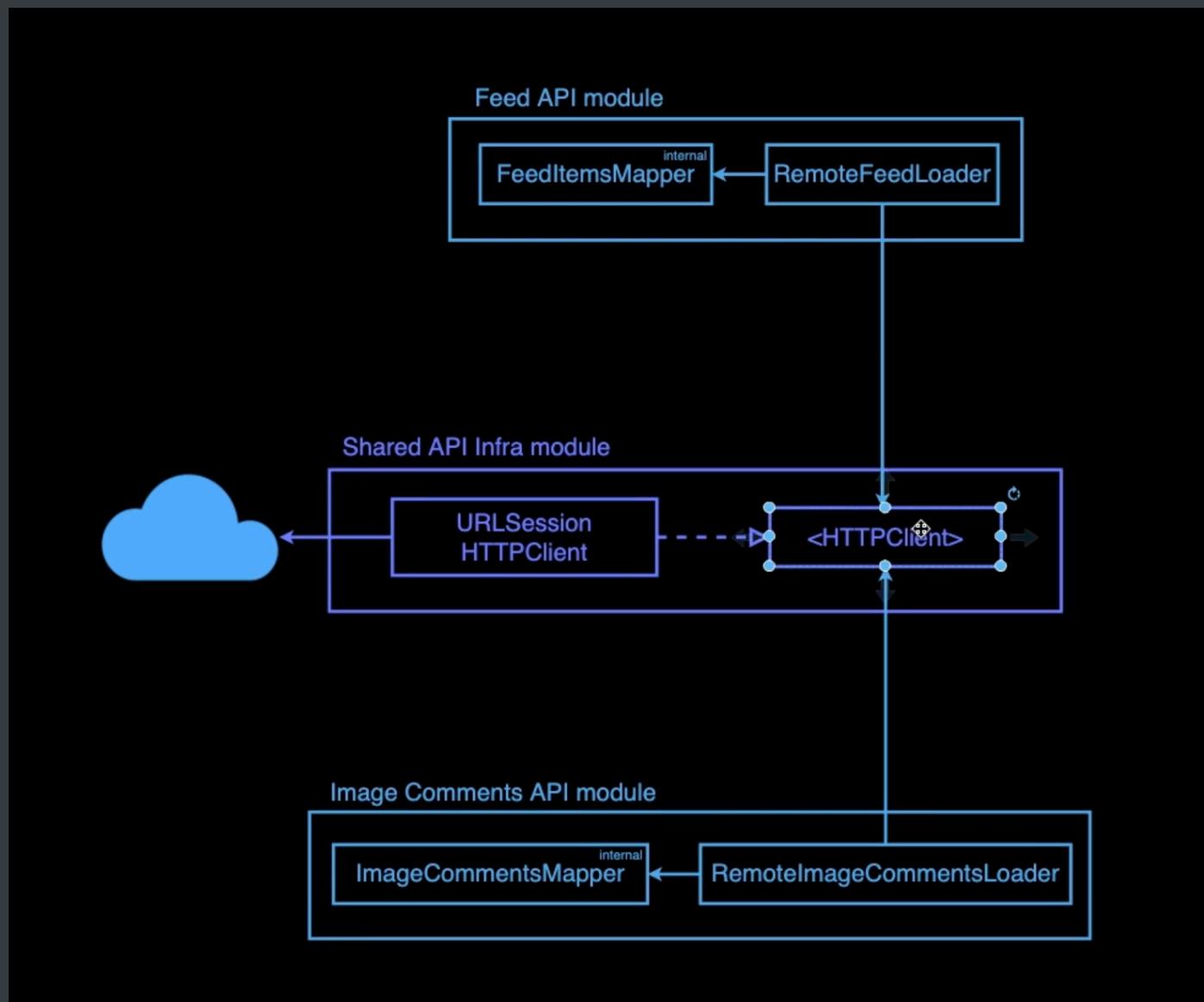


This is actually a great idea, since the `ImageComments API` could want to post comments in the future, whereas the `Feed API` module doesn't, therefore it would make sense to have two different `HTTPClients` that work for every use case. Otherwise, if we used the same `HTTPClient` for both use cases, 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 doesn't depend on any implementation, but on abstractions, in this case `<HTTPClient>`.

Second idea:

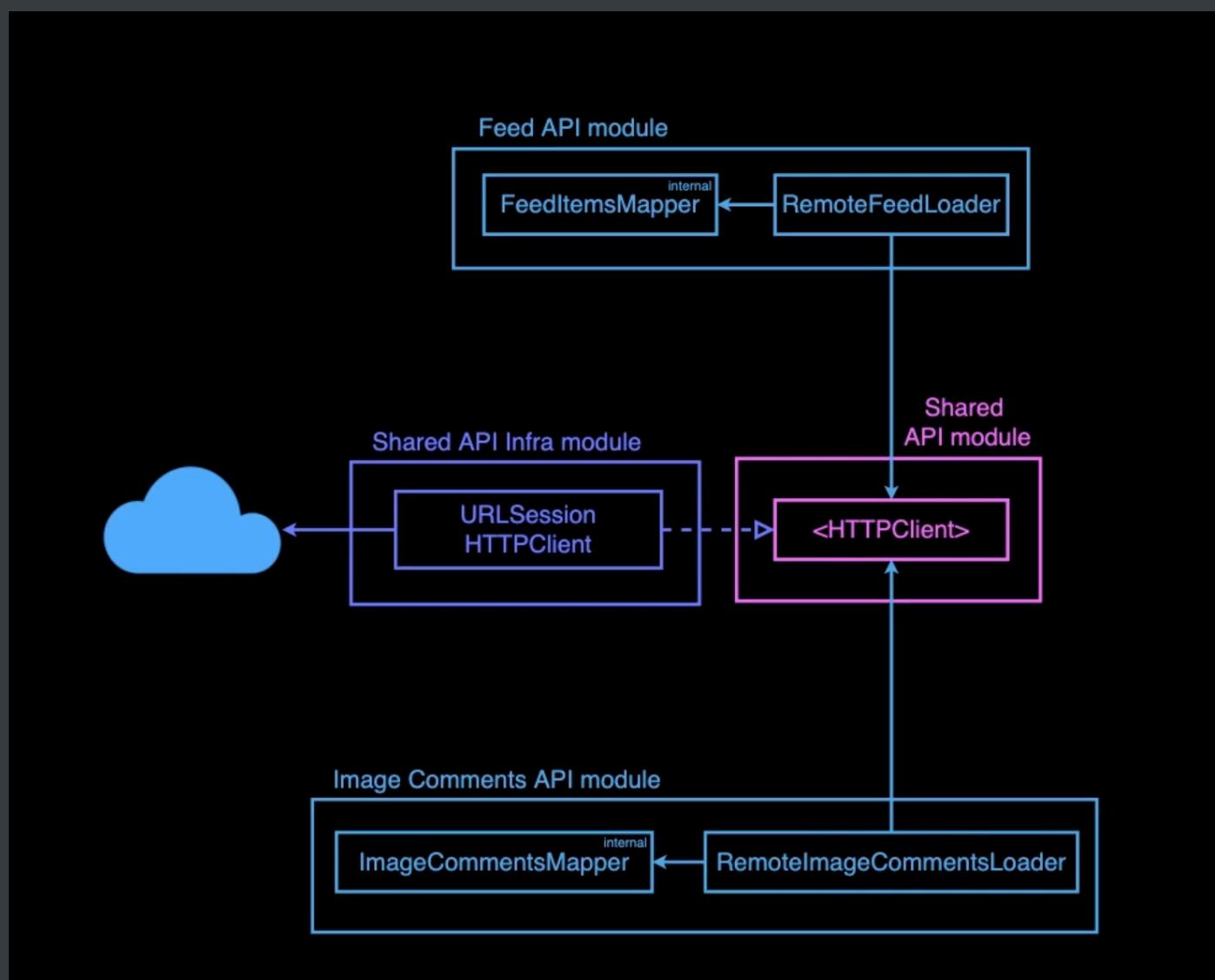
We might not want to duplicate the client, we may want to share it:



So what we would do is move the `<HTTPClient>` to the infrastructure module. This way they would share the client, but not depend on each other, 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 maintain. 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 doesn't 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 don't 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:

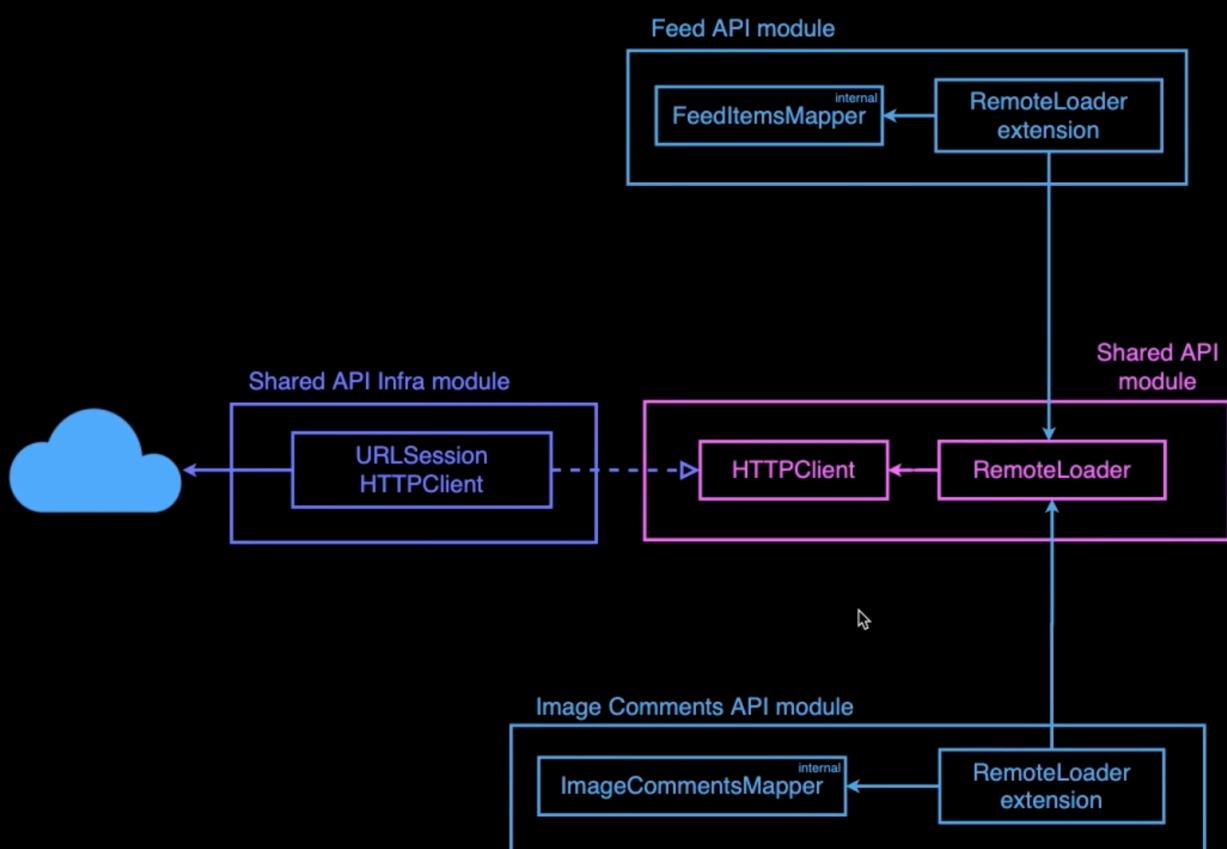
```

public typealias RemoteFeedLoader = RemoteLoader<[FeedImage]>

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

```

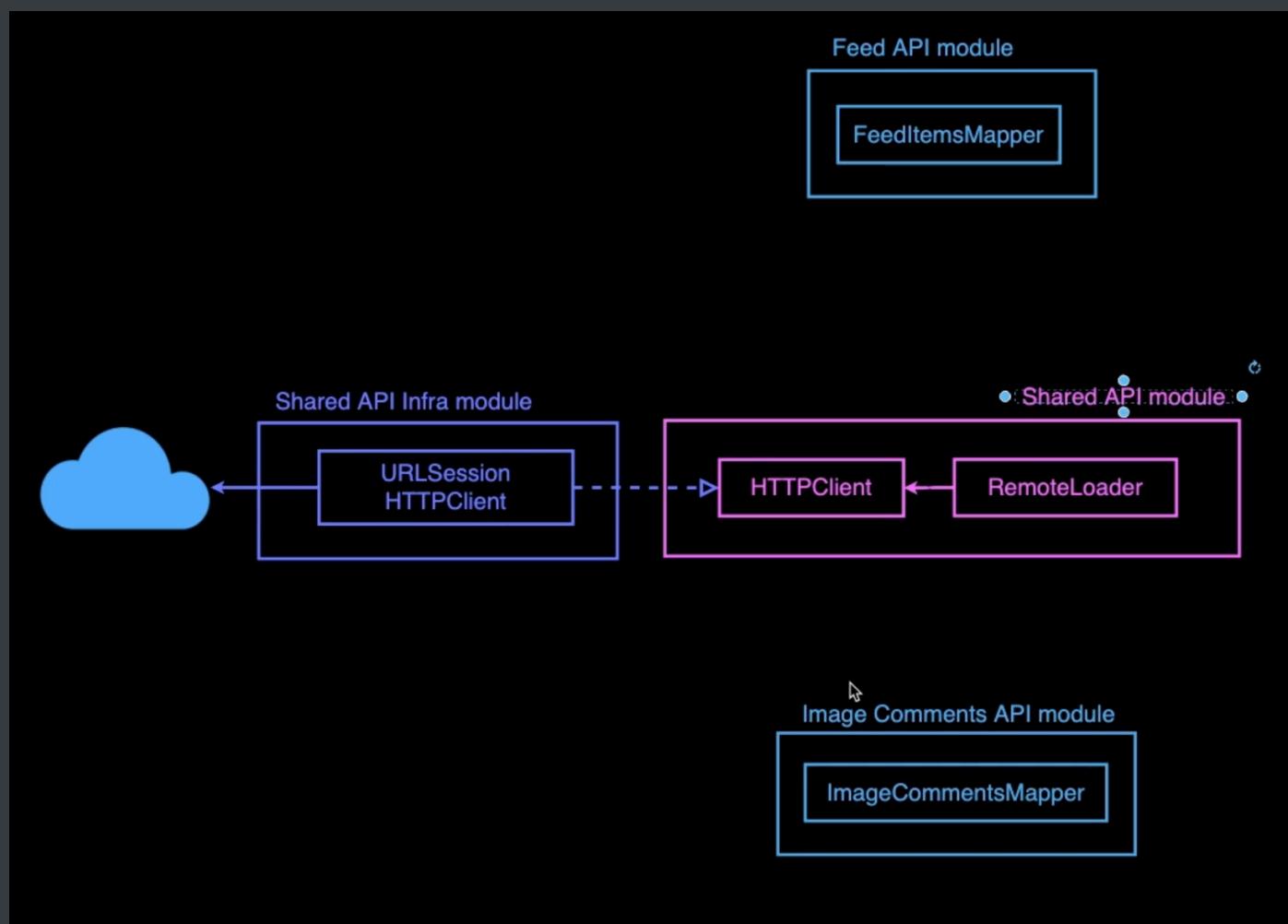
The new diagram is:



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:

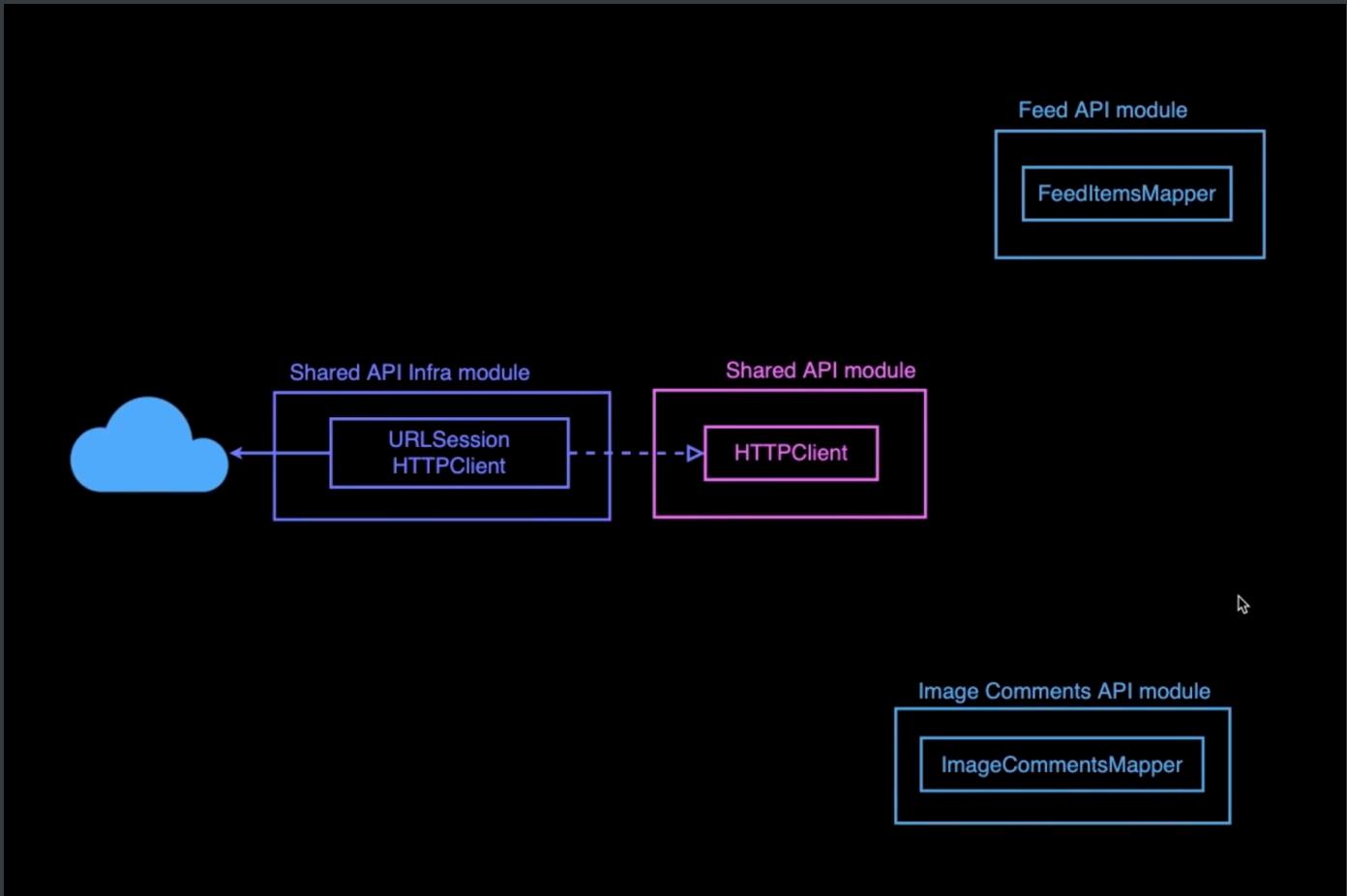
```
private func makeRemoteFeedLoaderWithLocalFallback() ->
FeedLoader.Publisher {
    let remoteURL = URL(string: "http://api-url.com")!

    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)
```

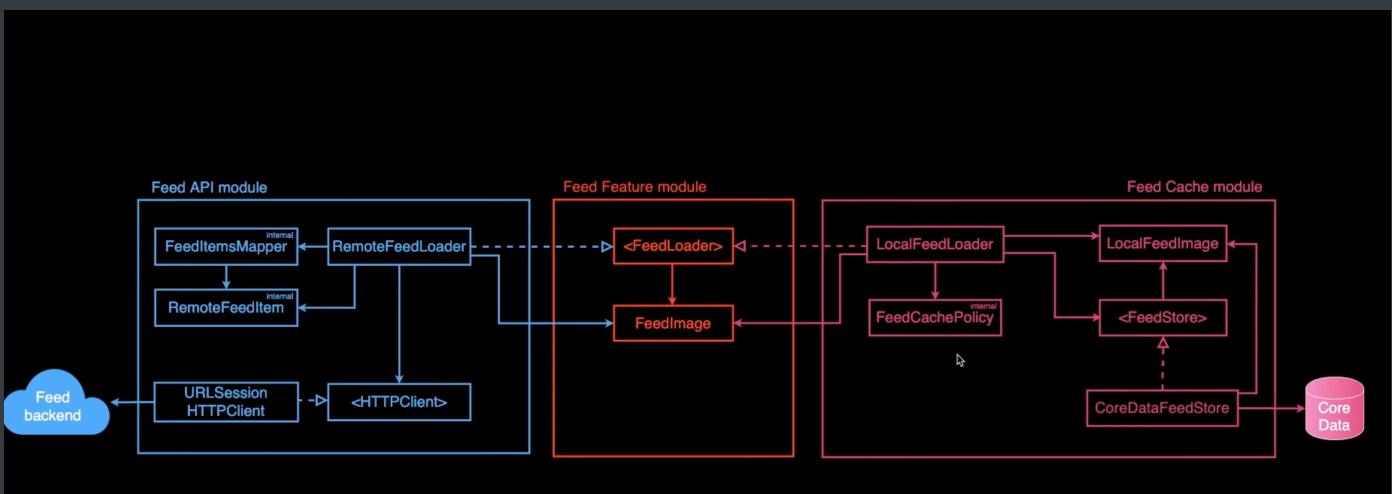
This way, our final design ends up looking like:



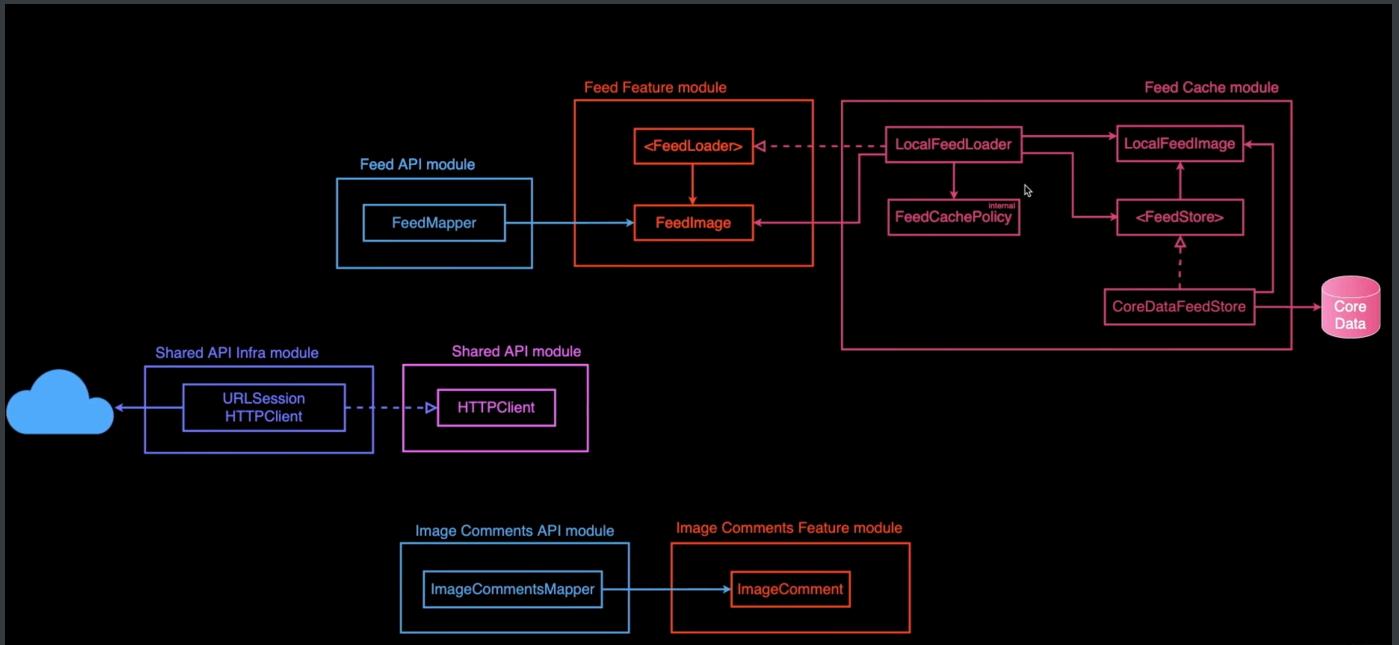
We don't need a loader, everything is composed with abstractions.

Old vs New Design

Old Design



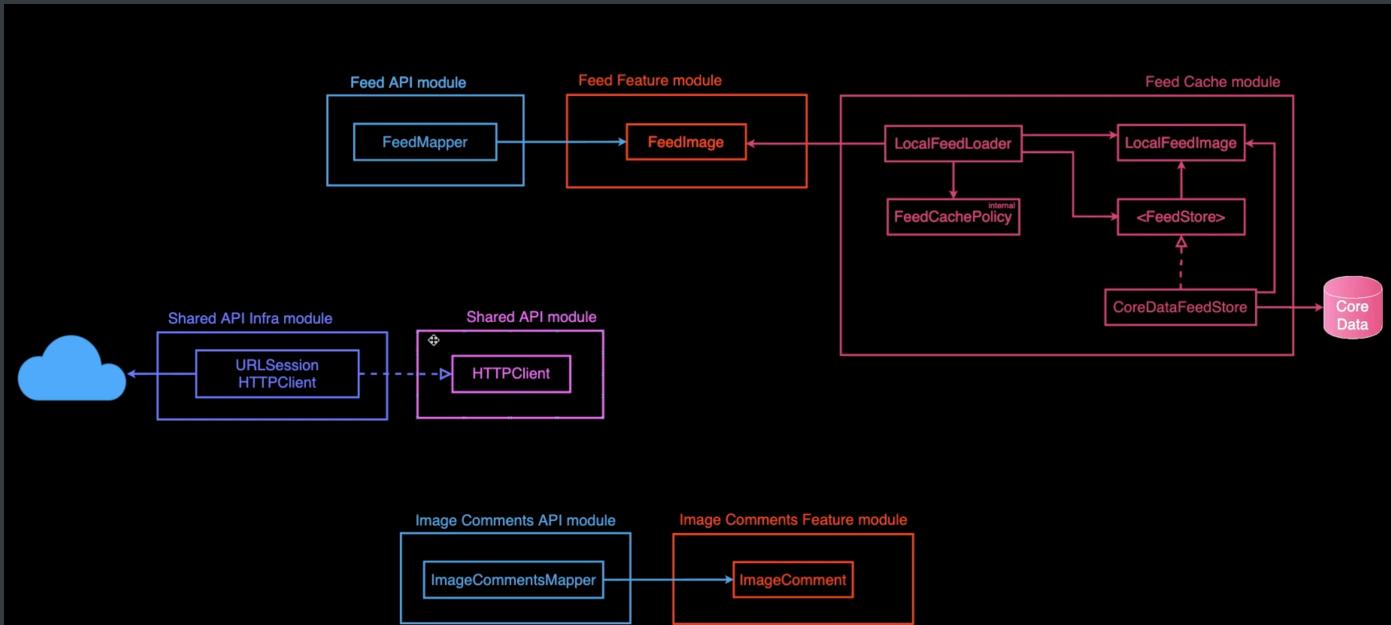
New Design



We don't have a **Loader** anymore, we just have the **FeedItemsMapper**, for the feed-use-case, everything composed in the Composition Root.

Since we don't 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 don't need a strategy anymore, so we can remove it.

Removing the `<FeedLoader>` :



We get a much simpler design.

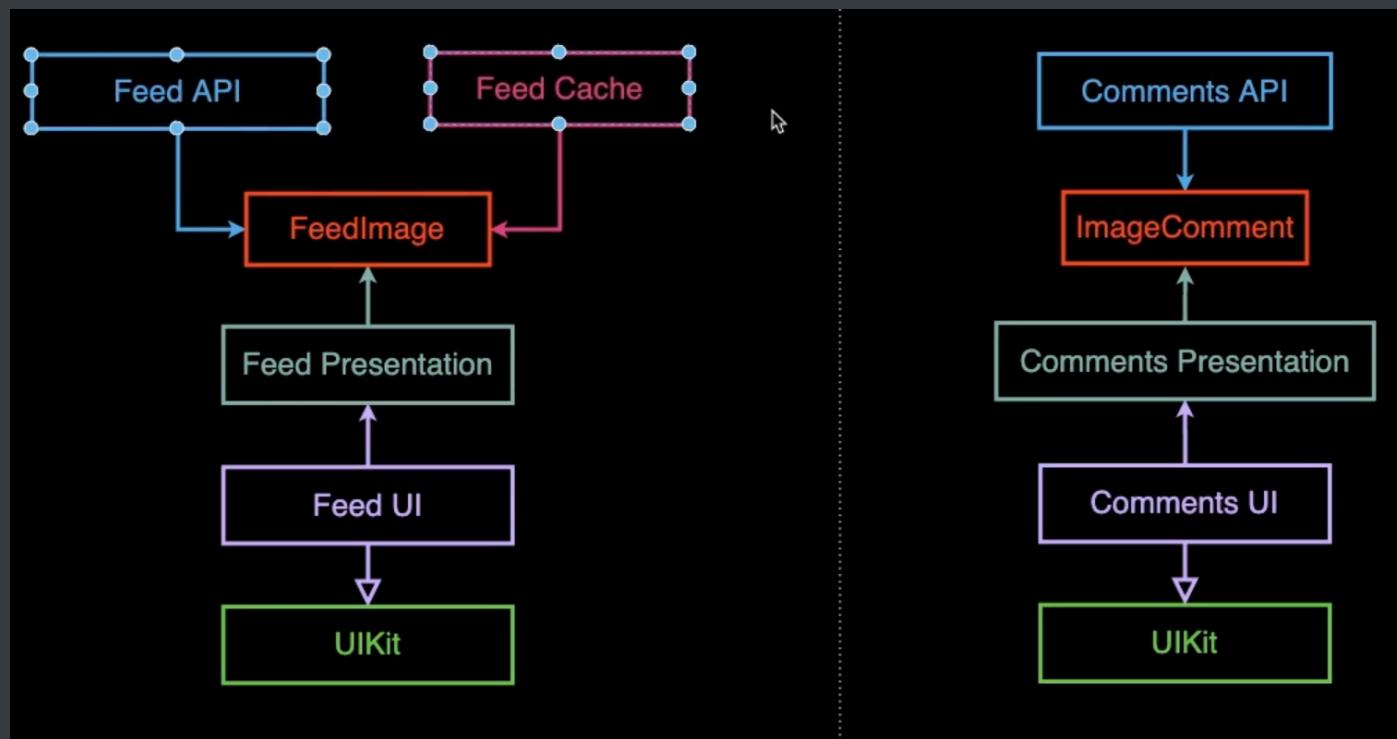
So, **Functional Style**: you don't *inject* dependencies, you *compose* them with the functional sandwich.

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).

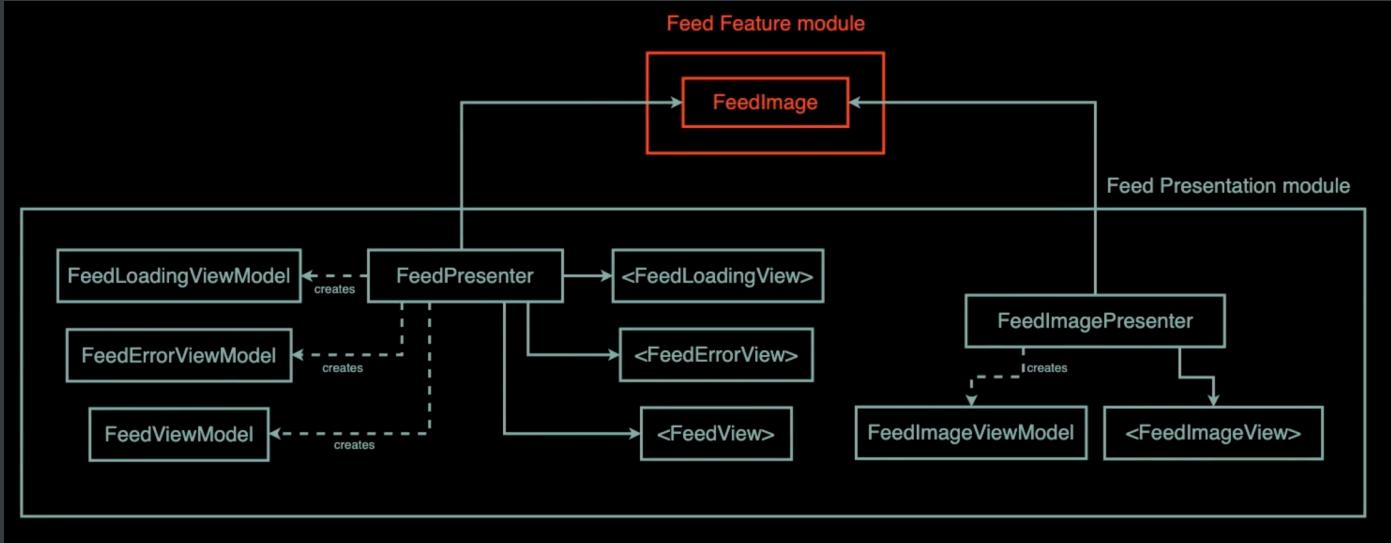
So we have to see how to reuse this code, by making some sort of abstraction, and not duplicate. This way everytime we need to load a resource we can use the esame presentation

Current Design



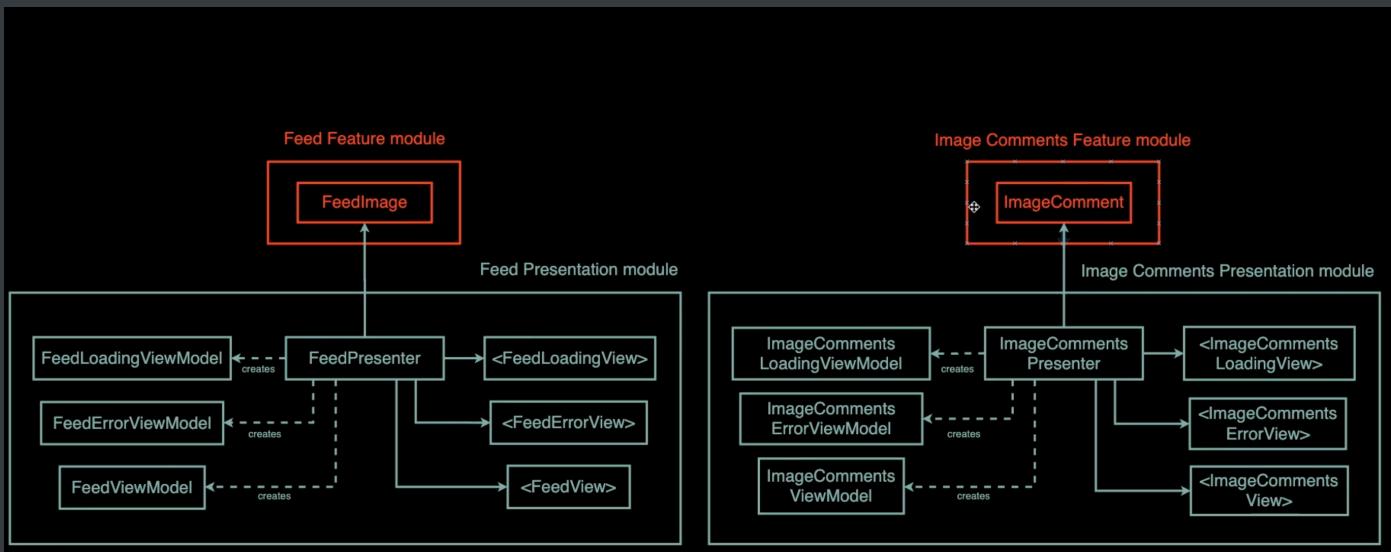
We can see that the features are decoupled from eachother, and we want to keep it that way, even though they are going share presentation logic and may share infrastructure for loading details.

Feed Presentation Logic



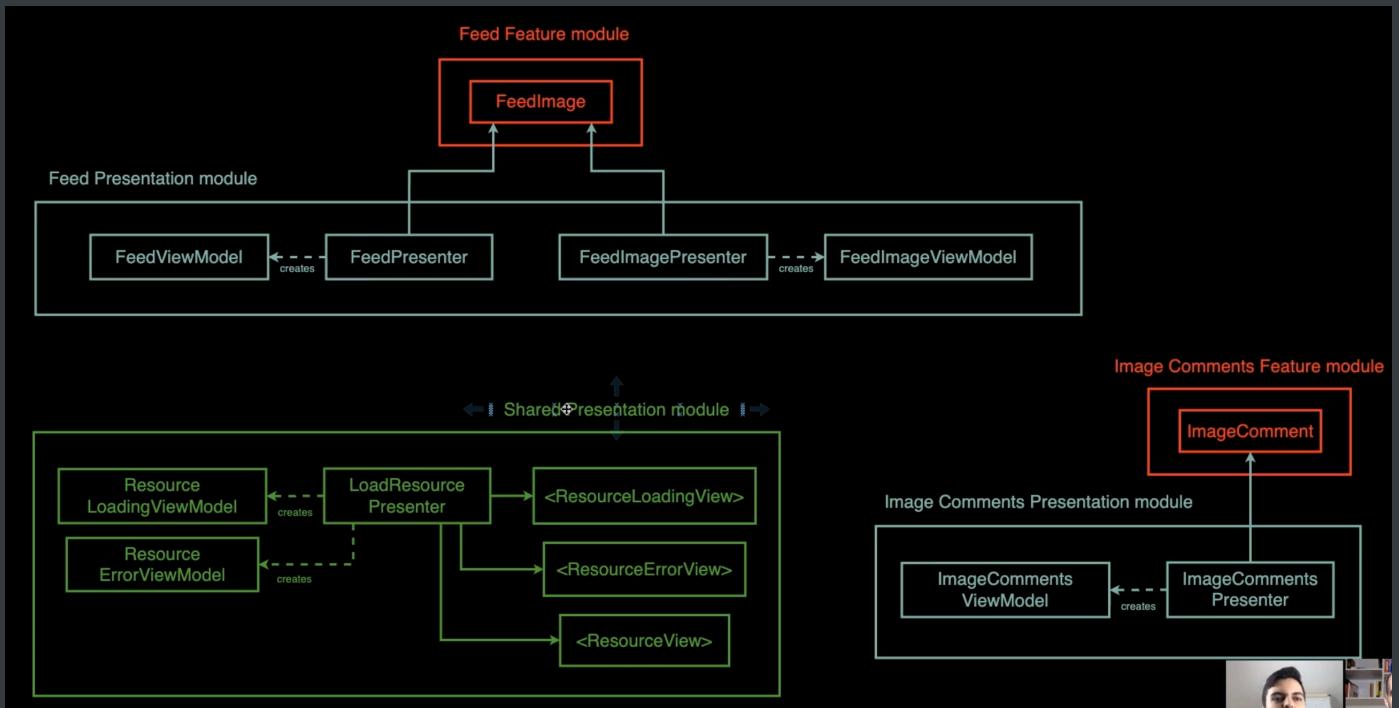
We have the **FeedPresenter** that we implemented in the presentation module, that creates ViewModels and passes them to the `<View>` protocols. So the **FeedPresenter** controls the loading of the resource, and the resource is the *feed* which is an array of images.

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 don'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 each other, 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 it's 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

Therefore the 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<FeedImageCell
Controller>, 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 come 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 `UITableViewDataSource` and `UITableViewDelegate` 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 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 necessary.

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:

- API (Live 001)
- 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)

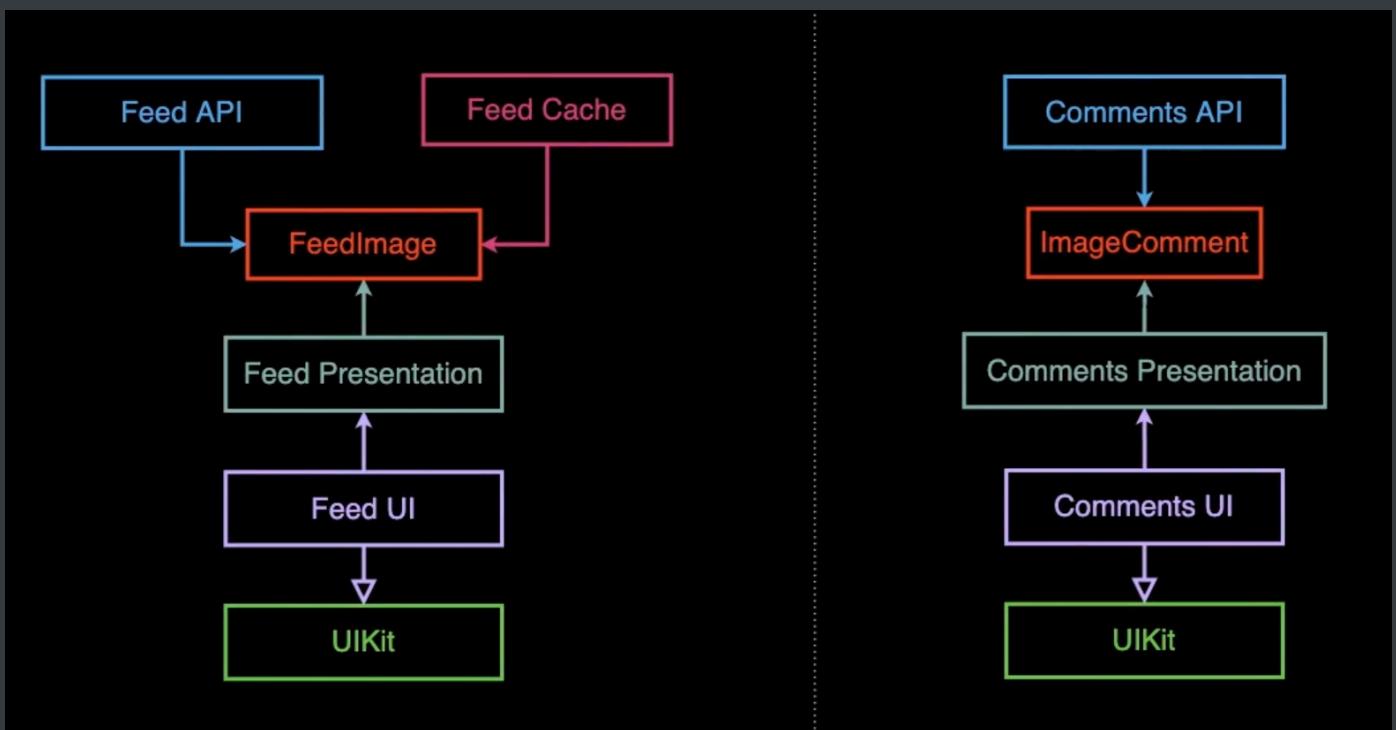
First Analysis

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.

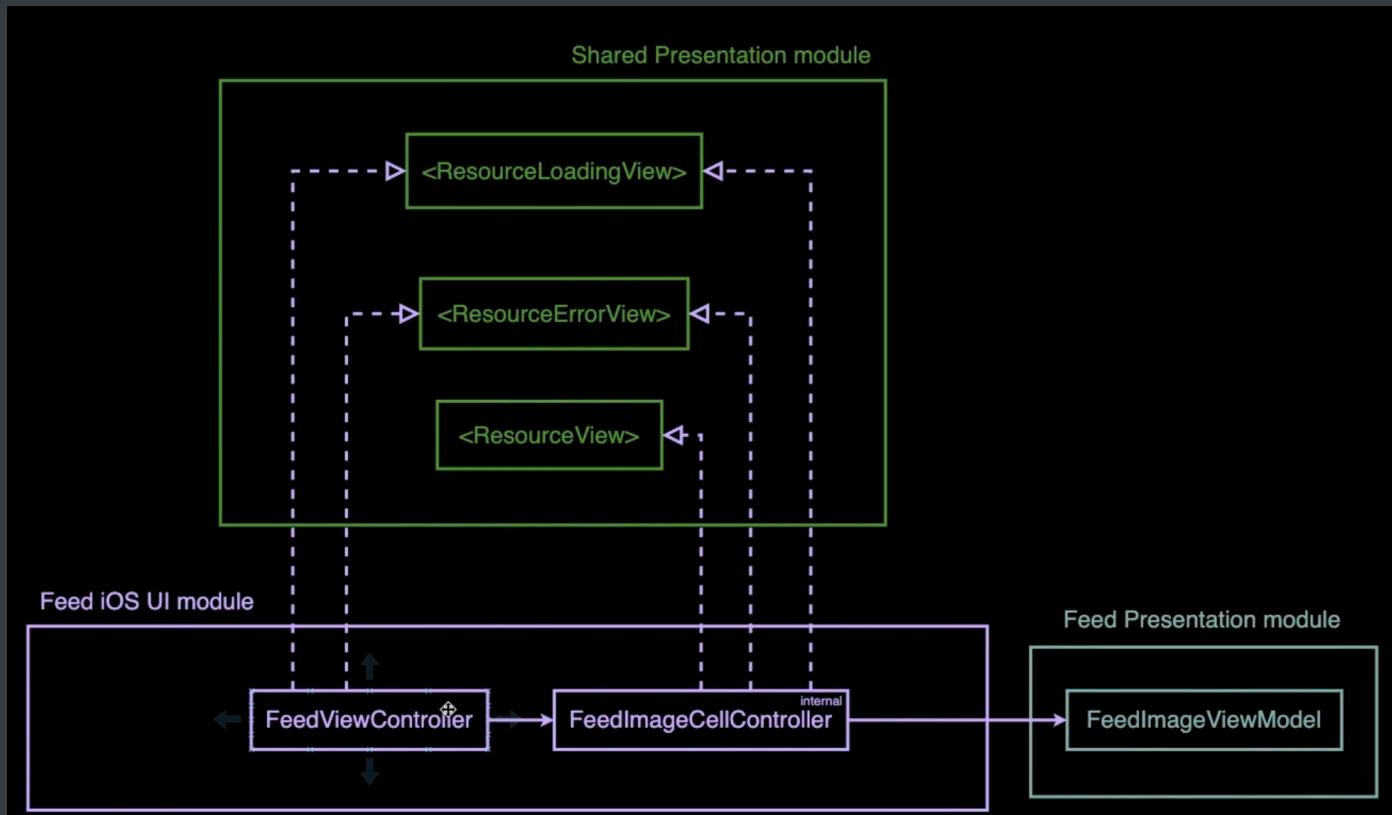
We are going to genericize our existing ui and just inject the different changes when needed, which in this case is the cell.

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:

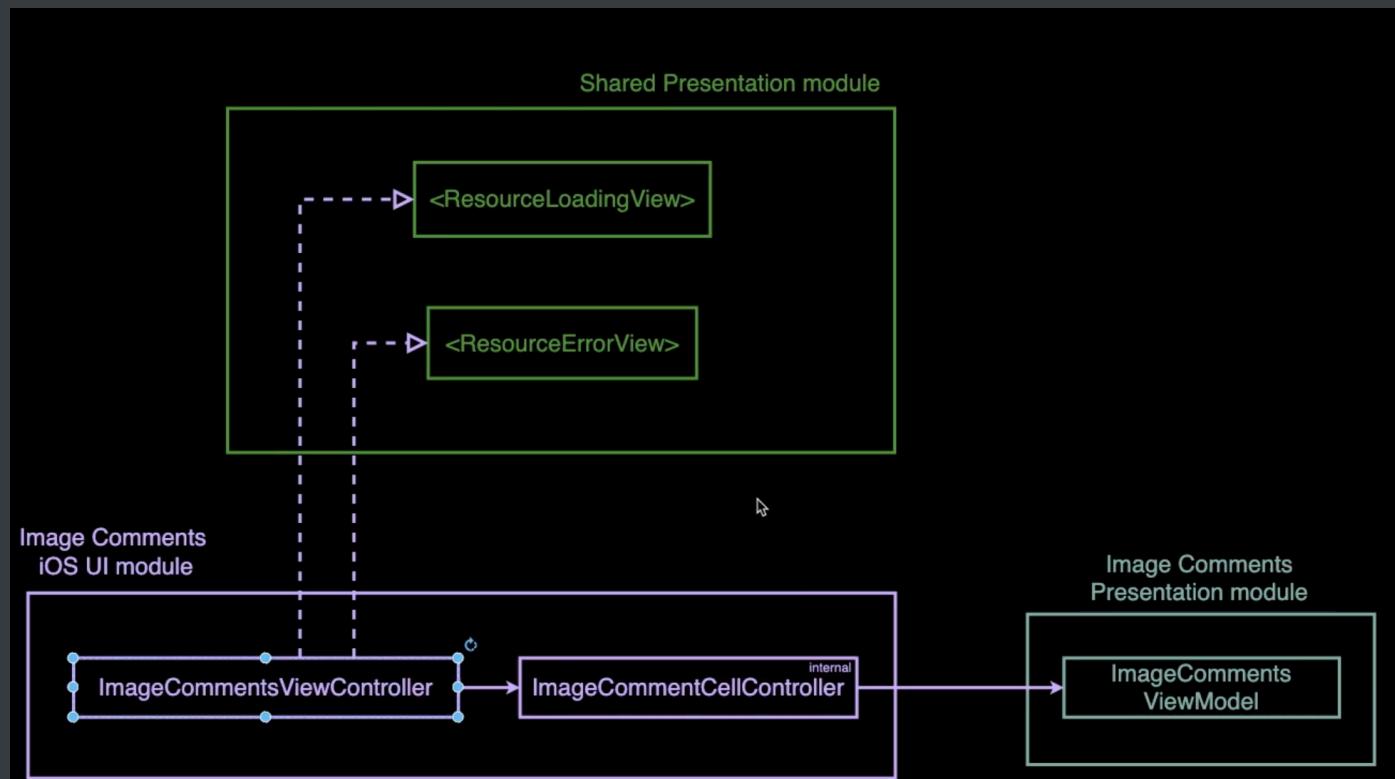
```
public final class FeedViewController: UITableViewcontroller,  
UITableViewDataSourcePrefetching, ResourceLoadingView, ResourceErrorView  
{  
    @IBOutlet private(set) public var errorView: ErrorView?  
  
    private var loadingControllers = [IndexPath:  
FeedImageCellController]()  
  
    private var tableViewModel = [FeedImageCellController](){  
        didSet { tableView.reloadData() }  
    }  
    ....  
    ....  
}
```

These **FeedImageCellControllers** 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  
    }  
    ...  
    ...  
}
```

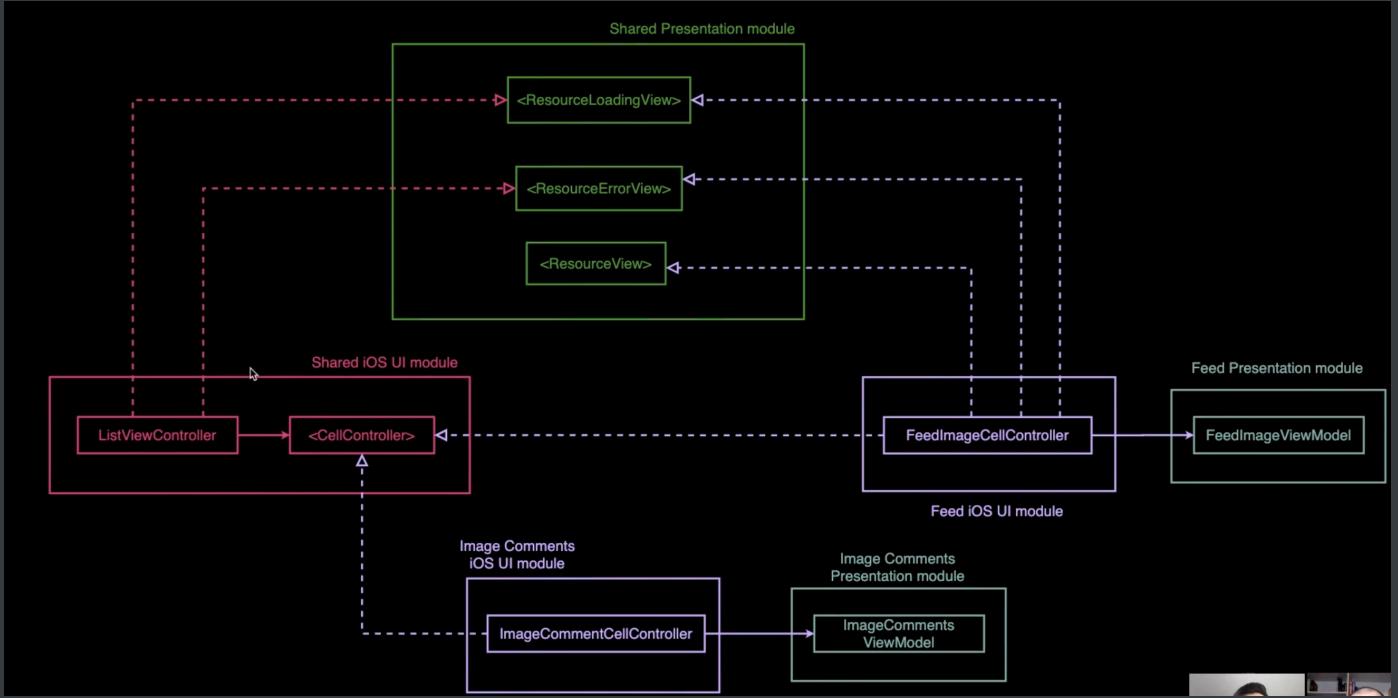
Where the ViewModel is the model, the Cell is the View and then the Controller.

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 `CellController` 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  
{}
```

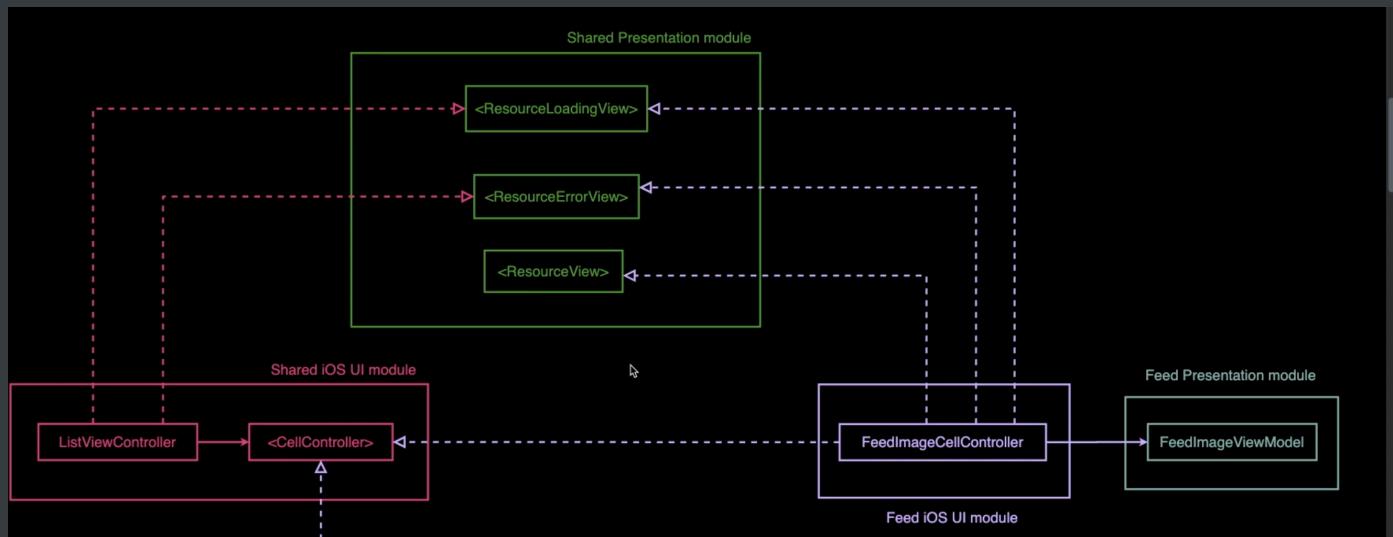
Following, we get rid of the `,`, and we replace it with a closure, since this protocol only had one method. So we replace the **delegate** property with an **onRefresh** closure. We assign this closure in the **CompositionRoot**, in the **FeedUIComposer**.

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.

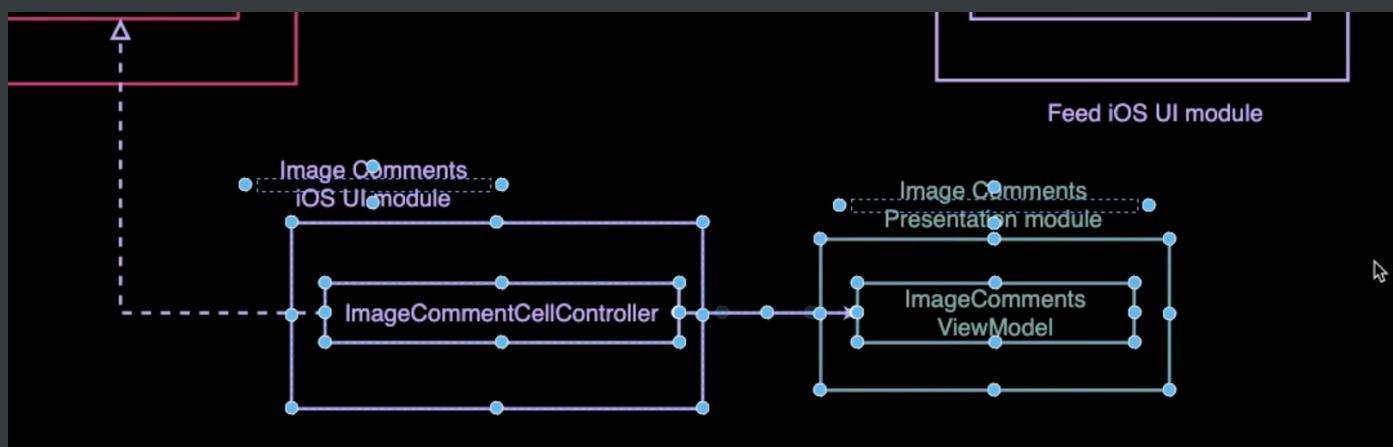
At this point we are already done with the first part:



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>` :

```

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

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

//UITableViewControllerDelegate
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 `UITableViewDataSource` 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 establish 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) {
    let controller = removeLoadingController(forRowAt: indexPath)
    controller?.tableView?(tableView, didEndDisplaying: cell, forRowAt:
indexPath)
}

//After
public override func tableView(_ tableView: UITableView,
didEndDisplaying cell: UITableViewCell, forRowAt 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(forRowAtIndexPath indexPath: IndexPath) ->
CellController {
    let controller = tableViewModel[indexPath.row]
    loadingControllers[indexPath] = controller
    return controller
}

private func removeLoadingController(forRowAtIndexPath 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 necessary **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 tableViewModel = [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 `cellForRow(at: IndexPath)` from the traditional tableview datasource. So when adding diffable datasources, we will delete the datasource method `cellForRow(at: 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 = [:]
    tableViewModel = 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 won't 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 doesn't 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 `ResourceViewAdapter` 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 to 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)
})
}

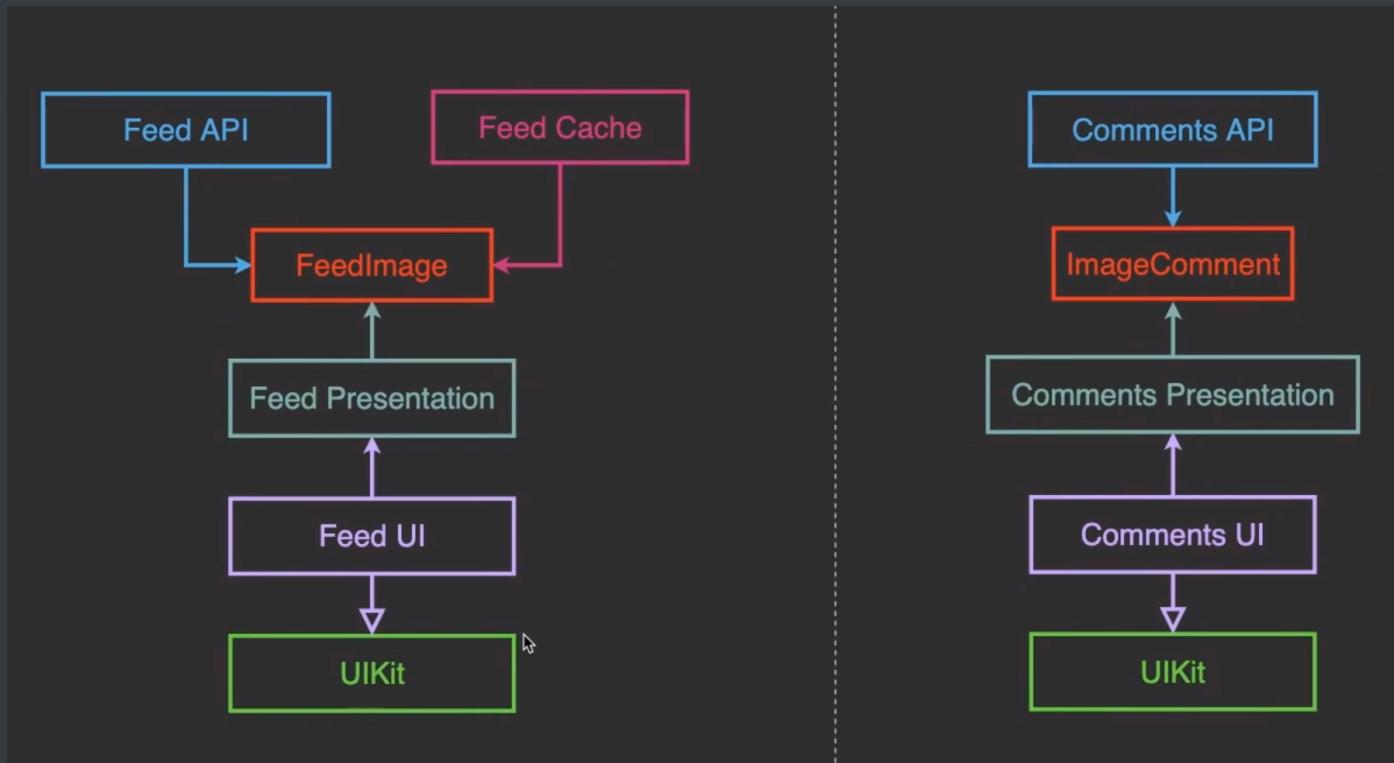
```

Composition and Navigation

Goal:

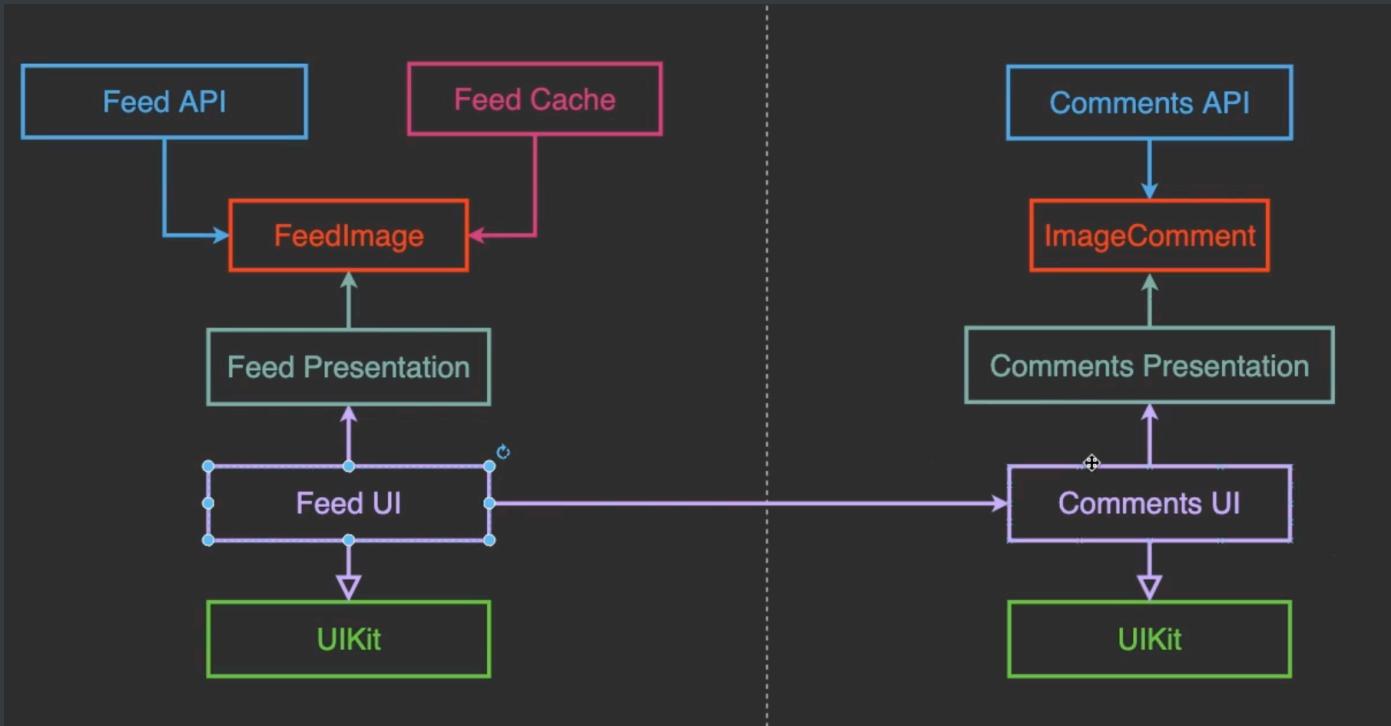
- Display a List of comments when the user taps on an image in the feed.
- At all times , the user should have a back button to return to the feed screen.
- Cancel any running comments API requests when the user navigates back.
- Integration tests
- Acceptance tests

Taking a look at the diagram:

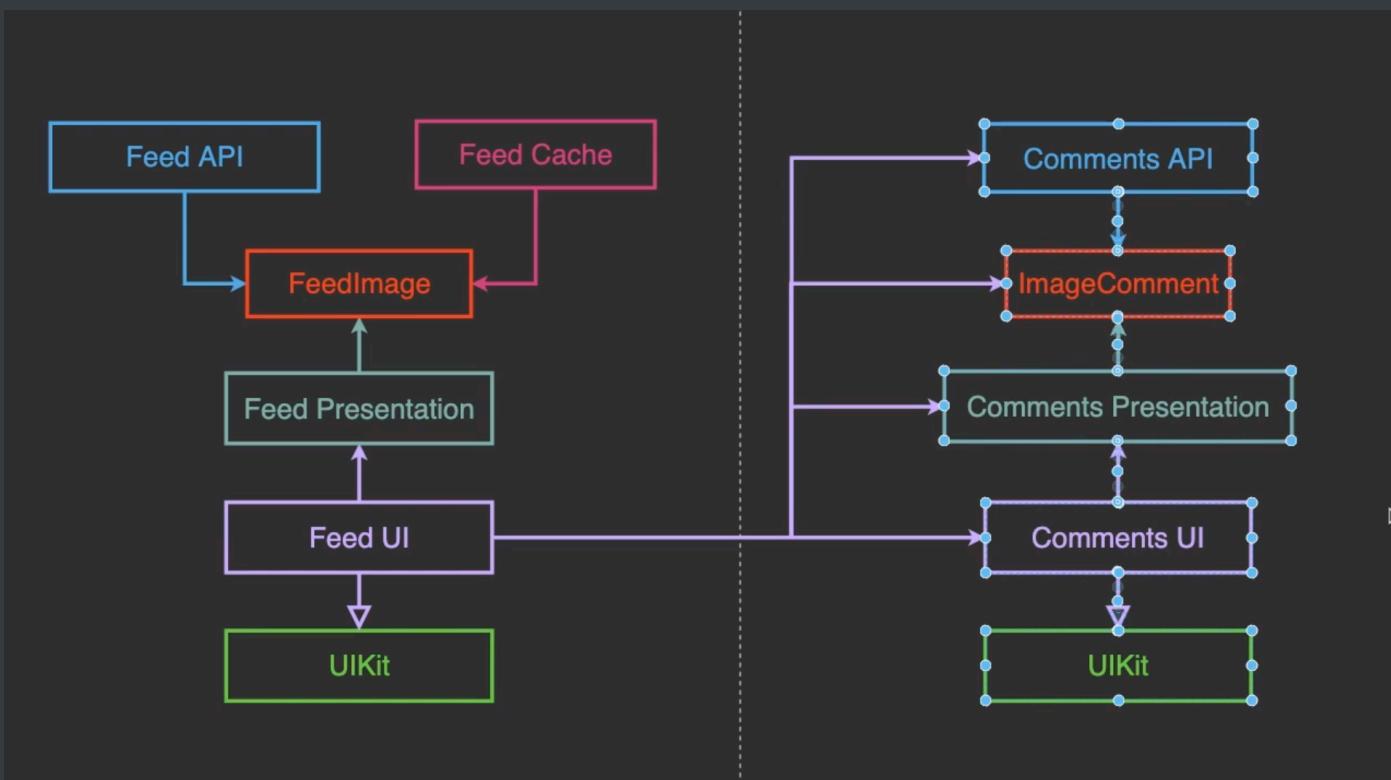


We've finished implementing all the layers, API, Presentation, UI and the data model, so now we need to plug the feed UI with the comments UI.

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



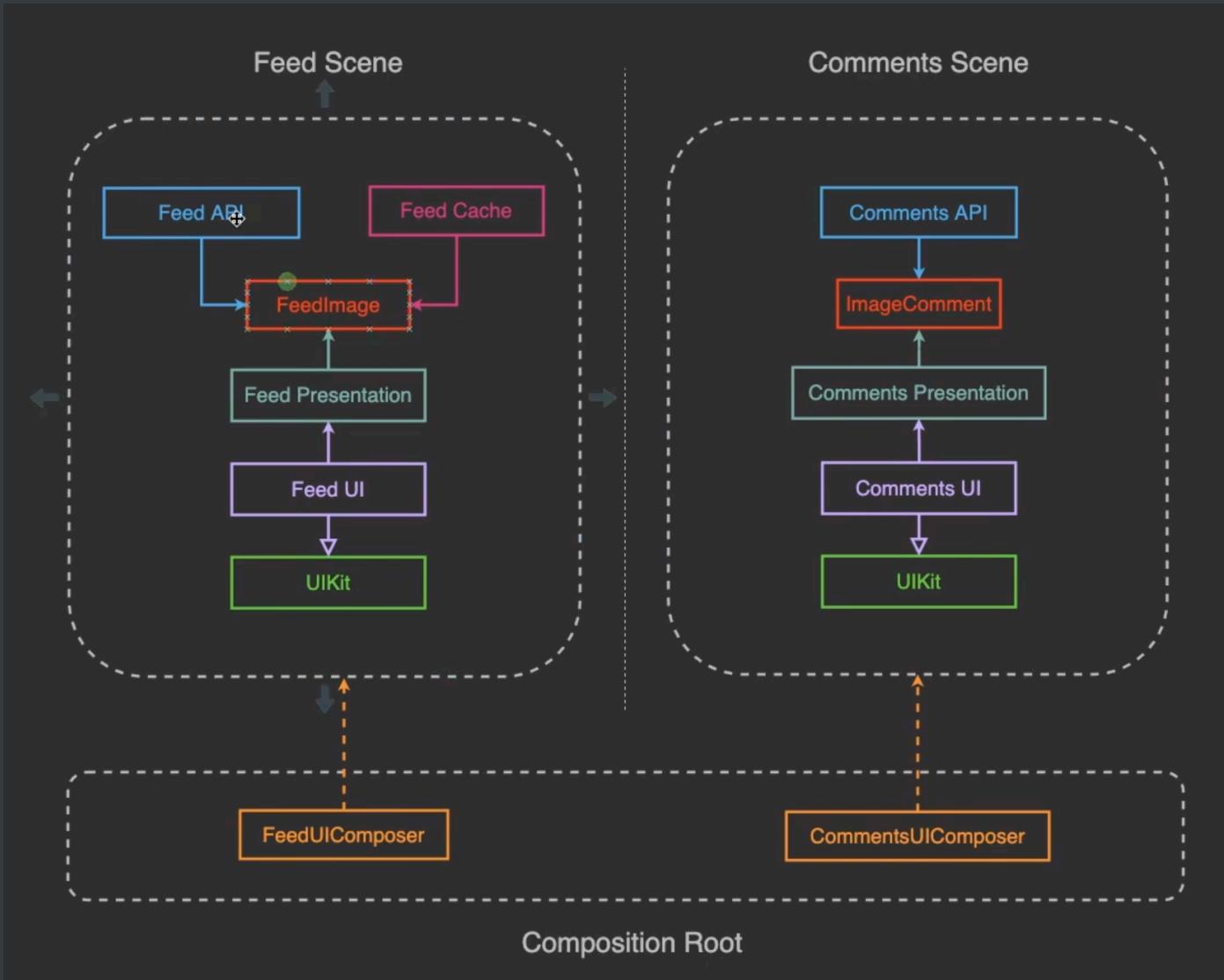
For simple UI transitions that don't require this kind of composition of dependencies, just pushing a viewcontroller within another works well, there is no problem with that, for example if our **FeedImageViewModel** already had access to all the comments (via a `comments: [ImageCommentViewModel]` property) and we were using a **FeedViewController** and we selected a cell, we could use `didSelectRowAt` 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 shouldn't 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 **CommentsUIComposer** 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 each other 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 don't 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 release pool as an autoreleased object that is kept until the next cycle because probably 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 it's 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 invoked after our test finishes, in the `tearDown` method, AFTER our `trackForMemoryLeaks` method has been executed, that is why if we don't 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 **FeedUITests**, 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 establishing 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:

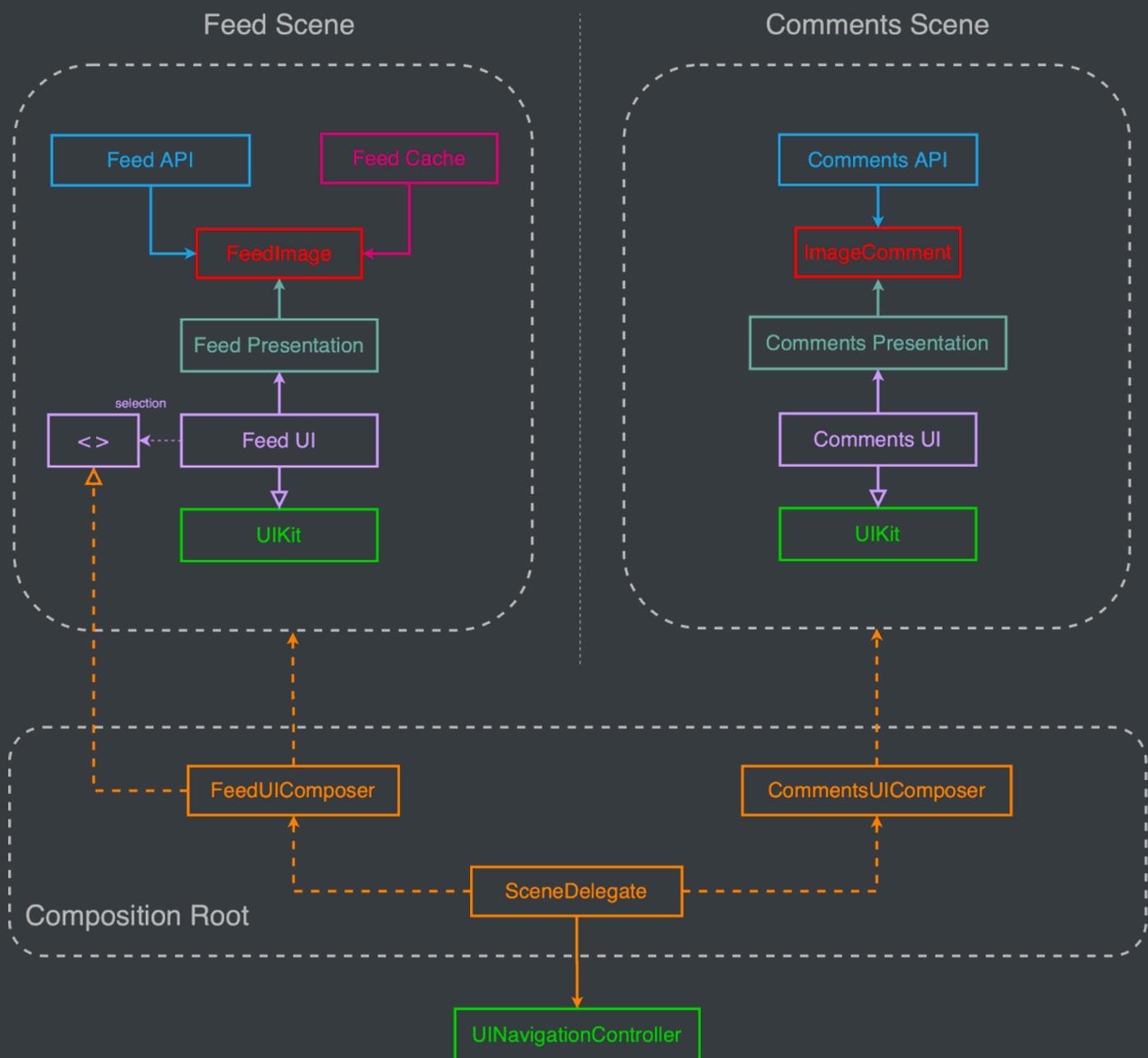
```
//FeedEndpoint
public enum FeedEndpoint {
    case get

    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 each other. 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:

- 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 don't 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 **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),  
            label.trailingAnchor.constraint(equalTo:  
                contentView.trailingAnchor, constant: 8),  
            label.topAnchor.constraint(equalTo:  
                contentView.topAnchor, constant: 8),  
            label.bottomAnchor.constraint(equalTo:  
                contentView.bottomAnchor, 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 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 isn't 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 doesn't 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 as well, 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 doesn't 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 can't 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 don't 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` property 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 multiple 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 hasn't completed and another loadMore is triggered it shouldn't 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 isn't 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 isn't nil, otherwise we display a single feed section in our `ListViewController`. If all is okay, we create 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 isn't 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, 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, 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 because we 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 FeedEndpoint 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 irrelevant, 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 doesn't need to know anything about pagination, since it's an API detail, this is why we didn't have to make any changes to the cache logic.

Refactoring

Now that everything 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 needed)

This pagination with caching we developed are independant/ decoupled from the pagination method/algorith, so we can change it from keyset to offset to whatever 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 instantiate 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 common to see a lot of code full of guards and returns that don't handle the errors.

Safe code isn't 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 Crashlytics**, 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 doesn't 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 :

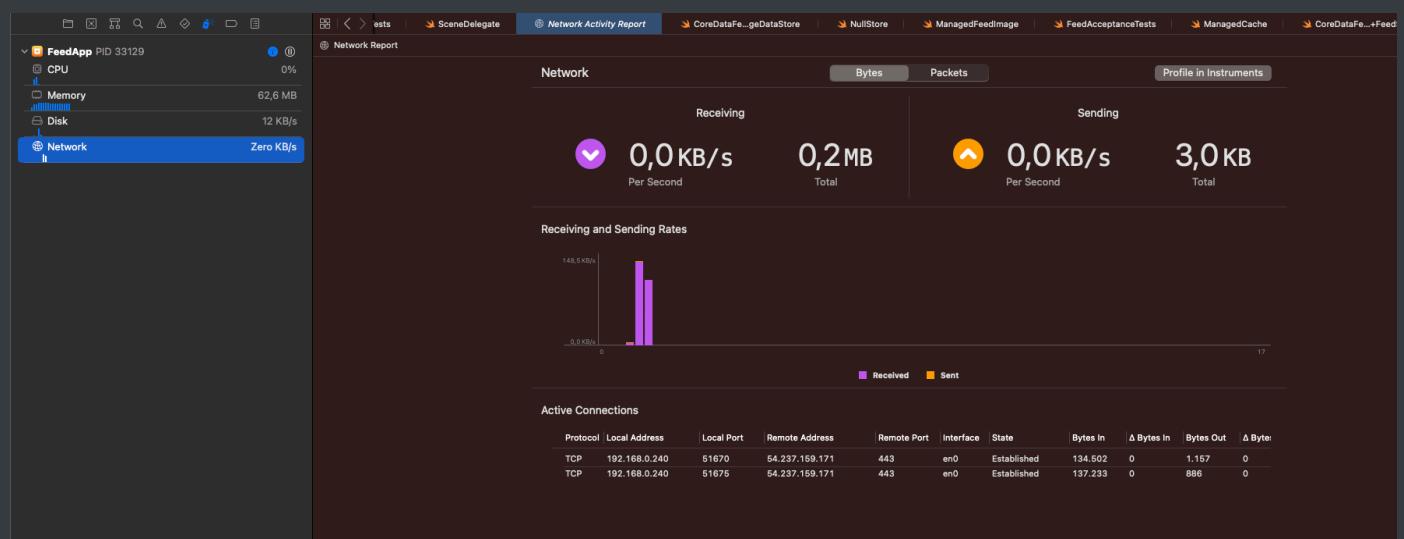
- **Don't ignore errors**
- **Don't let your app get into a weird state**
- **Provide a fallback strategy if you can** (and when you do, log the errors)
- **If the issue is really critical, it's better to crash the app than to get into a weird state where the App becomes unusable and must be reinstalled**

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.)

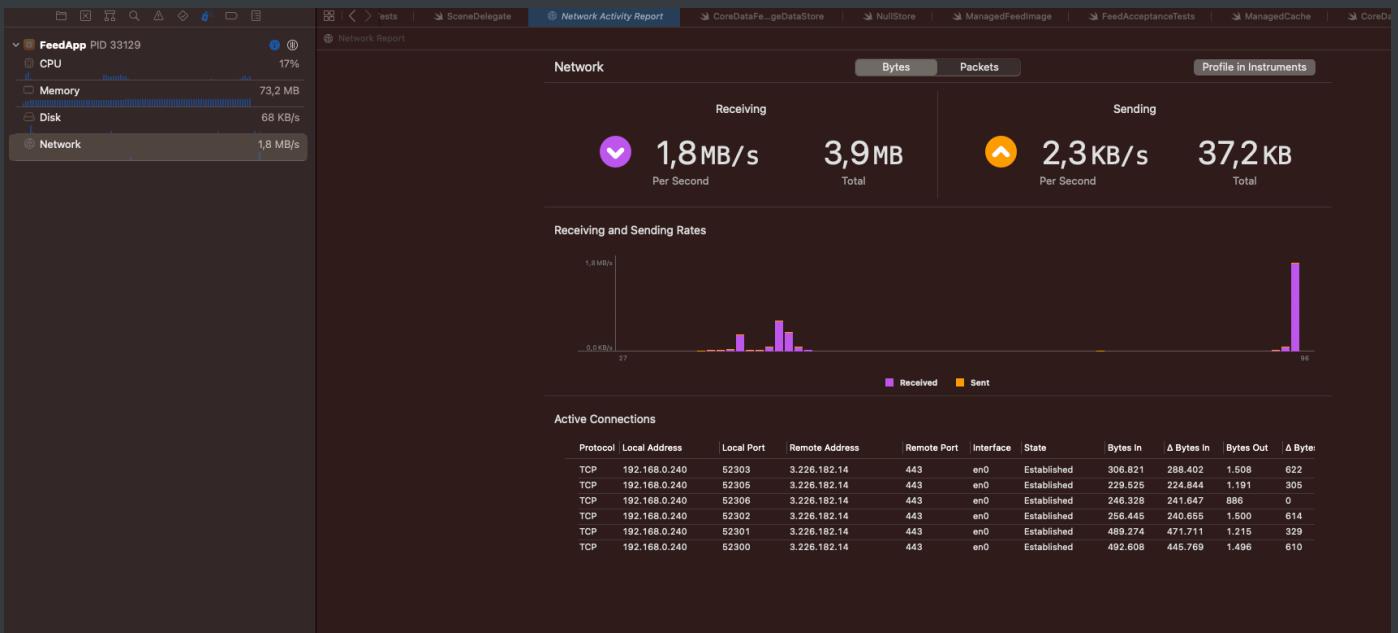
Logging is not free, it should be decided in the main module how to do it and when to do it and the more you log, the less you can find, so logs should be kept at a minimum for really critical errors and issues.

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(FeedImageMapper.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:

```

2023-07-19 23:03:51.421392-0300 FeedApp[35044:854929] [main] Started loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/11E123D5-1272-4F17-9B91-F3D0FFEC895A/blob
2023-07-19 23:03:51.449839-0300 FeedApp[35044:854929] [main] Started loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/31768993-1A2E-4B65-BD2A-D8AF06416730/blob
2023-07-19 23:03:52.261387-0300 FeedApp[35044:854931] [main] Finished loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/11E123D5-1272-4F17-9B91-F3D0FFEC895A/blob
2023-07-19 23:03:52.730269-0300 FeedApp[35044:854928] [main] Finished loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/31768993-1A2E-4B65-BD2A-D8AF06416730/blob
2023-07-19 23:04:08.215938-0300 FeedApp[35044:854928] [main] Started loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/54F35D06-9CC6-4294-A0D8-963D397E8B98/blob
2023-07-19 23:04:08.217087-0300 FeedApp[35044:854928] [main] Started loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/93F3F9B1-D3B0-46B6-9546-2808A65FCC20/blob
2023-07-19 23:04:08.217959-0300 FeedApp[35044:854928] [main] Started loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/CF68C4D7-D7B9-44BE-8D10-BDD8464915C2/blob
2023-07-19 23:04:08.223578-0300 FeedApp[35044:854928] [main] Started loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/8FD1F6C1-F8DE-4C76-B57C-4AE28F14D270/blob
2023-07-19 23:04:08.227988-0300 FeedApp[35044:854928] [main] Started loading url:
  https://ile-api.essentialdeveloper.com/essential-feed/v1/image/5ACC3B53-F55F-4C89-8DD4-4013420D6D15/blob
2023-07-19 23:04:08.230732-0300 FeedApp[35044:854928] [main] Started loading url:

```

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("Failed 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] Failed 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] Failed 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 :

Parameters

| | |
|----------------------------------|--|
| <code>receiveSubscription</code> | An optional closure that executes when the publisher receives the subscription from the upstream publisher. This value defaults to nil. |
| <code>receiveOutput</code> | An optional closure that executes when the publisher receives a value from the upstream publisher. This value defaults to nil. |
| <code>receiveCompletion</code> | An optional closure that executes when the upstream publisher finishes normally or terminates with an error. This value defaults to nil. |
| <code>receiveCancel</code> | An optional closure that executes when the downstream receiver cancels publishing. This value defaults to nil. |
| <code>receiveRequest</code> | An optional closure that executes when the publisher receives a request for more elements. This value defaults to nil. |

Returns

A publisher that performs the specified closures when publisher events occur.

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:

```

https://ile-api.essentialdeveloper.com/essential-feed/v1/image/8FD1F6C1-F8DE-4C76-B57C-4AE28F14D270/blob
2023-07-19 23:38:24.282763-0300 FeedApp[36043:883399] [boringssl] boringssl_context_handle_fatal_alert(1991)
[C5.1.2:2][0x7fc611729590] write alert, level: fatal, description: certificate unknown
2023-07-19 23:38:24.291930-0300 FeedApp[36043:883399] [boringssl] boringssl_context_error_print(1981)
[C6.1.2:2][0x7fc611729590] Error: 140488686564984:error:1000007d:SSL
routines:OPENSSL_internal:CERTIFICATE_VERIFY_FAILED:/Library/Caches/com.apple.xbs/Sources/boringssl_Sim/ssl/handshake
.cc:419:
2023-07-19 23:38:24.311127-0300 FeedApp[36043:883399] [boringssl] boringssl_session_handshake_incomplete(88)
[C5.1.2:2][0x7fc611729590] SSL library error
2023-07-19 23:38:24.311576-0300 FeedApp[36043:883399] [boringssl] boringssl_session_handshake_error_print(43)
[C6.1.2:2][0x7fc611729590] Error: 140488686564984:error:1000007d:SSL
routines:OPENSSL_internal:CERTIFICATE_VERIFY_FAILED:/Library/Caches/com.apple.xbs/Sources/boringssl_Sim/ssl/handshake
.cc:419:
2023-07-19 23:38:24.314241-0300 FeedApp[36043:883399] [boringssl] nw_protocol_boringssl_handshake_negotiate_proceed(771)
[C5.1.2:2][0x7fc611729590] handshake failed at state 12288: not completed
2023-07-19 23:38:24.486433-0300 FeedApp[36043:883403] [main] Started loading url:
https://ile-api.essentialdeveloper.com/essential-feed/v1/image/5ACC3B53-F55F-4C89-8DD4-4013420D6D15/blob
2023-07-19 23:38:24.500306-0300 FeedApp[36043:883399] [boringssl] boringssl_context_handle_fatal_alert(1991)
[C6.1.2:2][0x7fc61250a2e0] write alert, level: fatal, description: certificate unknown
2023-07-19 23:38:24.506404-0300 FeedApp[36043:883399] [boringssl] boringssl_context_error_print(1981)
[C6.1.2:2][0x7fc61250a2e0] Error: 140488686564984:error:1000007d:SSL
routines:OPENSSL_internal:CERTIFICATE_VERIFY_FAILED:/Library/Caches/com.apple.xbs/Sources/boringssl_Sim/ssl/handshake
.cc:419:
2023-07-19 23:38:24.512745-0300 FeedApp[36043:883399] [boringssl] boringssl_session_handshake_incomplete(88)
[C6.1.2:2][0x7fc61250a2e0] SSL library error
2023-07-19 23:38:24.514326-0300 FeedApp[36043:883399] [boringssl] boringssl_session_handshake_error_print(43)
[C6.1.2:2][0x7fc61250a2e0] Error: 140488686564984:error:1000007d:SSL
routines:OPENSSL_internal:CERTIFICATE_VERIFY_FAILED:/Library/Caches/com.apple.xbs/Sources/boringssl_Sim/ssl/handshake
.cc:419:
2023-07-19 23:38:24.515373-0300 FeedApp[36043:883399] [boringssl] nw_protocol_boringssl_handshake_negotiate_proceed(771)
[C6.1.2:2][0x7fc61250a2e0] handshake failed at state 12288: not completed

```

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.

