Write a GitLab API GraphQL query to get the last twelve hours' worth of pipeline status of three specific repositories.

1 / 1

To get the last twelve hours' worth of pipeline status for three specific repositories using the GitLab GraphQL API, you'll first need to determine the repository IDs for the projects you want to query. You can do this by searching for projects by their name or namespace and project name. Once you have the repository IDs, you can write a query to fetch the pipeline statuses.

Here's an example GraphQL query that retrieves the pipeline status for three specific repositories:

graphql

Copy code

```graphql
query {
  lastTwelveHours: subtractDuration(from: "now", value: "12", unit: HOUR)
  project1: project(fullPath: "namespace/repository1") {
    id
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
  project2: project(fullPath: "namespace/repository2") {
    id
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
  project3: project(fullPath: "namespace/repository3") {
    id
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
}
```

Replace `namespace/repository1`, `namespace/repository2`, and `namespace/repository3` with the appropriate paths for your desired repositories. This query will return the pipeline status for each repository's `master` branch created within the last twelve hours. Adjust the `ref` value if you want to query a different branch.

Keep in mind that you'll need to set up GitLab API authentication (by using a personal access token) before sending the query. You can find more information on GitLab API authentication in the GitLab GraphQL API documentation.

1 / 1

Is it possible to use the `projects()` query to select using the three known project IDs?

1 / 1

Yes, you can use the `projects()` query to fetch data for multiple projects using their IDs. You can pass a list of project IDs as an argument to the `projects()` query. Here's an example of how to do that:

graphql

Copy code

```graphql
query {
  lastTwelveHours: subtractDuration(from: "now", value: "12", unit: HOUR)
  projects(ids: ["gid://gitlab/Project/1", "gid://gitlab/Project/2", "gid://gitlab/Project/3"]) {
    nodes {
      id
      fullPath
      pipelines(ref: "master", createdAfter: $lastTwelveHours) {
        edges {
          node {
            id
            status
            createdAt
          }
        }
      }
    }
  }
}
```

Replace the project IDs `gid://gitlab/Project/1`, `gid://gitlab/Project/2`, and `gid://gitlab/Project/3` with the appropriate IDs for your desired repositories. This query will return the pipeline status for each repository's `master` branch created within the last twelve hours. Adjust the `ref` value if you want to query a different branch.

As mentioned in the previous answer, make sure to set up GitLab API authentication (by using a personal access token) before sending the query. You can find more information on GitLab API authentication in the GitLab GraphQL API documentation.

1 / 1

If I know the fullPath values with which to query my three projects, but I don't know the IDs just yet, can I still get all three projects using the projects() query in a single GraphQL request?

1 / 1

Unfortunately, the `projects()` query in GitLab GraphQL API doesn't support querying projects directly by their `fullPath` values. However, you can still achieve this using an alias for each project and using the `project()` query instead. The `project()` query allows you to fetch data for a single project by its `fullPath`. You can use aliases to make separate requests for each project in a single GraphQL request.

Here's an example of how to do that:

graphql

Copy code

```graphql
query {
  lastTwelveHours: subtractDuration(from: "now", value: "12", unit: HOUR)
  project1: project(fullPath: "namespace/repository1") {
    id
    fullPath
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
  project2: project(fullPath: "namespace/repository2") {
    id
    fullPath
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
  project3: project(fullPath: "namespace/repository3") {
    id
    fullPath
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
}
```

Replace `namespace/repository1`, `namespace/repository2`, and `namespace/repository3` with the appropriate `fullPath` values for your desired repositories. This query will return the pipeline status for each repository's `master` branch created within the last twelve hours. Adjust the `ref` value if you want to query a different branch.

Don't forget to set up GitLab API authentication (by using a personal access token) before sending the query. More information on GitLab API authentication can be found in the [GitLab GraphQL API documentation].

1 / 1

If I query each project by fullPath to identify their IDs, can I then use projects() given those IDs, and do all that in one GraphQL request?

1 / 1

Unfortunately, you cannot directly use the results of one query as input to another query within the same GraphQL request. GraphQL does not support query chaining, meaning you can't use the output of one query as an input variable for another query within the same request.

However, you can still achieve this in two steps:

1. Query the projects by their `fullPath` to get the project IDs.

2. Use the obtained project IDs in a second request to query the projects using the `projects()` query.

Here's the first query to get the project IDs using the `fullPath` values:

graphql

Copy code

```graphql
query {
  project1: project(fullPath: "namespace/repository1") {
    id
  }
  project2: project(fullPath: "namespace/repository2") {
    id
  }
  project3: project(fullPath: "namespace/repository3") {
    id
  }
}
```

Replace `namespace/repository1`, `namespace/repository2`, and `namespace/repository3` with the appropriate `fullPath` values for your desired repositories.

After running the first query, extract the project IDs from the response. Then, use the obtained project IDs to run the second query using the `projects()` query:

graphql

Copy code

```graphql
query {
  lastTwelveHours: subtractDuration(from: "now", value: "12", unit: HOUR)
  projects(ids: ["gid://gitlab/Project/1", "gid://gitlab/Project/2", "gid://gitlab/Project/3"]) {
    nodes {
      id
      fullPath
      pipelines(ref: "master", createdAfter: $lastTwelveHours) {
        edges {
          node {
            id
            status
            createdAt
          }
        }
      }
    }
  }
}
```

Replace the project IDs `gid://gitlab/Project/1`, `gid://gitlab/Project/2`, and `gid://`

`gitlab/Project/3` with the appropriate IDs you obtained from the first query's response.

Remember to set up GitLab API authentication (by using a personal access token) before sending the queries. More information on GitLab API authentication can be found in the GitLab GraphQL API documentation.

1 / 1

Cannot query field "subtractDuration" on type "Query"

1 / 1

I apologize for the confusion in my previous responses. It appears I made an error in providing the `subtractDuration` query, which is not a valid field in the GitLab GraphQL API. Instead, you should calculate the time for the last twelve hours in your client-side code or application and pass it as a variable to your GraphQL query.

For example, in JavaScript, you can calculate the time for the last twelve hours like this:

javascript

Copy code

```javascript
const twelveHoursAgo = new Date(Date.now() - 12 * 60 * 60 * 1000).toISOString();
```

Now, you can use this value in your GraphQL query:

graphql

Copy code

```graphql
query($lastTwelveHours: Time!) {
  project1: project(fullPath: "namespace/repository1") {
    id
    fullPath
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
  project2: project(fullPath: "namespace/repository2") {
    id
    fullPath
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
  project3: project(fullPath: "namespace/repository3") {
    id
    fullPath
    pipelines(ref: "master", createdAfter: $lastTwelveHours) {
      edges {
        node {
          id
          status
          createdAt
        }
      }
    }
  }
}
```

Replace `namespace/repository1`, `namespace/repository2`, and `namespace/repository3` with the appropriate `fullPath` values for your desired repositories. This query will return the pipeline status for each repository's `master` branch created within the last twelve hours. Adjust the `ref` value if you want to query a different branch.

Pass the `lastTwelveHours` variable when executing the query, like this (assuming you're using JavaScript and a GraphQL client like Apollo):

javascript

Copy code

```
const result = await client.query({
  query: YOUR_GRAPHQL_QUERY,
  variables: { lastTwelveHours: twelveHoursAgo },
});
```

Don't forget to set up GitLab API authentication (by using a personal access token) before sending the query. More information on GitLab API authentication can be found in the GitLab GraphQL API documentation.

1 / 1

Now I have this:

```
query {
  projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://gitlab/Project/10582521"]) {
    nodes {
      id
      fullPath
      pipelines(first: 10) {
        edges {
          node {
            id
            status
            createdAt
          }
        }
      }
    }
  }
}
```

Next let's include open merge requests.

1 / 1

To include open merge requests for each project in your GraphQL query, you can add the `mergeRequests` field to the `nodes` field in your query. The `mergeRequests` field accepts arguments to filter the results, such as `state` to filter by the state of the merge request.

Here's the modified query including open merge requests for each project:

graphql

Copy code

```graphql
query {
  projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://gitlab/Project/10582521"]) {
    nodes {
      id
      fullPath
      pipelines(first: 10) {
        edges {
          node {
            id
            status
            createdAt
          }
        }
      }
      mergeRequests(state: opened, first: 10) {
        edges {
          node {
            id
            title
            createdAt
            author {
              username
            }
          }
        }
      }
    }
  }
}
```

This query will return the first 10 open merge requests for each project, along with their ID, title, creation date, and the author's username. You can adjust the `first` argument if you want to fetch a different number of merge requests.

Remember to set up GitLab API authentication (by using a personal access token) before sending the query. More information on GitLab API authentication can be found in the GitLab GraphQL API documentation.

1 / 1

I've iterated a bit and now I have:

```
query {
  projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://gitlab/Project/10582521"]) {
    nodes {
      id
      fullPath
      webUrl
      pipelines(first: 5, ref: "master") {
        pageInfo {
          endCursor
        }
        nodes {
          id
          path
          createdAt
          queuedDuration
          duration
          status
          stages {
            nodes {
              name
              status
            }
          }
          testReportSummary {
            total {
              failed
              count
              time
            }
          }
        }
      }
      mergeRequests(state: opened, first: 10) {
        pageInfo {
          endCursor
        }
        nodes {
          id
          webUrl
          draft
          title
          createdAt
          author {
            username
          }
          headPipeline {
            id
            path
            createdAt
            queuedDuration
            duration
            status
            stages {
              nodes {
                name
                status
              }
            }
            testReportSummary {
              total {
                failed
                count
                time
              }
            }
          }
          approvalsLeft
          approvedBy {
            nodes {
              username
              avatarUrl
            }
          }
          shouldBeRebased
        }
      }
    }
  }
}
```

I think this includes all the fields I'm interested in.

Now let's make a macOS SwiftUI app to display this information from a Menu Bar Item. Where should I start?

1 / 1

To create a macOS SwiftUI app that displays this information from a Menu Bar Item, you can follow these steps:

1. Create a new macOS SwiftUI app.
2. Set up the GitLab API client and fetch data using the GraphQL query.
3. Create a SwiftUI view for displaying the fetched data.
4. Implement the Menu Bar Item and its popover content using the fetched data.

Here's a step-by-step guide to help you get started:

Step 1: Create a new macOS SwiftUI app

Open Xcode and create a new macOS app using the "App" template under the "macOS" section. Name your project (e.g., "GitLabStatusBarApp"), make sure to select "SwiftUI" as the Interface, and choose "Swift" as the language.

Step 2: Set up the GitLab API client and fetch data using the GraphQL query

First, you'll need to install a GraphQL client library like Apollo to make GraphQL requests. You can use Swift Package Manager to add Apollo as a dependency to your project. Follow the official Apollo installation guide for details.

Next, configure the Apollo client by creating a new Swift file called `Network.swift` and add the following code:

swift

Copy code

```
import Foundation
import Apollo

class Network {
    static let shared = Network()

    private(set) lazy var apollo: ApolloClient = {
        let url = URL(string: "https://gitlab.com/api/graphql")!
        var request = URLRequest(url: url)
        request.addValue("Bearer YOUR_PERSONAL_ACCESS_TOKEN", forHTTPHeaderField: "Authorization")
        let configuration = URLSessionConfiguration.default
        configuration.httpAdditionalHeaders = request.allHTTPHeaderFields
        let client = URLSessionClient(sessionConfiguration: configuration, callbackQueue: nil)
        let store = ApolloStore(cache: InMemoryNormalizedCache())
        return ApolloClient(networkTransport: HTTPNetworkTransport(url: url, client: client), store: store)
    }()
}
```

Replace `YOUR_PERSONAL_ACCESS_TOKEN` with your GitLab personal access token.

Now, create a new Swift file called `GitLabDataFetcher.swift` to fetch data using the provided GraphQL query. You can create a function like `fetchData(completion: @escaping (Result<YourDataType, Error>) -> Void)` that makes the GraphQL request and returns the data as a Swift model. Parse the received data and pass it to the completion handler.

Step 3: Create a SwiftUI view for displaying the fetched data

Create a new SwiftUI view called `GitLabDataView.swift` and design the view according to your requirements, using the fetched data from the previous step. You can create a List view or a custom layout to display the projects, pipelines, and merge requests.

Step 4: Implement the Menu Bar Item and its popover content

First, create a new SwiftUI view called `StatusBarMenu.swift`. This view will be shown when the user clicks the Menu Bar Item. You can add your `GitLabDataView` as a subview in this view.

Next, create a new Swift file called `StatusBarItem.swift` to manage the Menu Bar Item. In this file, create a class that conforms to `ObservableObject` and handles the Menu Bar Item creation, popover display, and fetching data.

Here's an example of how you can create the `StatusBarItem` class:

swift

Copy code

```swift
import SwiftUI
import Combine

class StatusBarItem: NSObject, ObservableObject {
    @Published var gitLabData: YourDataType?
    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem
    private var popover: NSPopover

    init()
```

1 / 1

Continue

1 / 1

Here's the continuation of the `StatusBarItem` class:

swift

Copy code

```swift
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength:
NSStatusItem.variableLength)
        popover = NSPopover()

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(named: "MenuBarIcon") // Replace
"MenuBarIcon" with your custom icon name
            button.action = #selector(togglePopover(_:))
            button.target = self
        }

        // Set up the popover content
        popover.behavior = .transient
        popover.animates = true
        popover.contentViewController = NSHostingController(rootView:
StatusBarMenu())
```

```
    }

    @objc func togglePopover(_ sender: AnyObject) {
        if popover.isShown {
            popover.performClose(sender)
        } else {
            if let button = statusBarItem.button {
                fetchData { [weak self] result in
                    switch result {
                    case .success(let data):
                        DispatchQueue.main.async {
                            self?.gitLabData = data
                        }
                    case .failure(let error):
                        print("Error fetching GitLab data: \(error)")
                    }
                }
                popover.show(relativeTo: button.bounds, of: button, preferredEdge: .minY)
            }
        }
    }

    func fetchData(completion: @escaping (Result<YourDataType, Error>) -> Void) {
        // Call the fetchData method from your GitLabDataFetcher.swift here and pass the completion handler
    }
}
```

Replace `YourDataType` with the appropriate data type you used for your fetched data model. In your `StatusBarMenu.swift`, make sure to use an `@EnvironmentObject` property wrapper to access the `gitLabData` and update the view accordingly:

swift

Copy code

```
import SwiftUI

struct StatusBarMenu: View {
    @EnvironmentObject var statusBarItem: StatusBarItem

    var body: some View {
        VStack {
            // Use GitLabDataView with the fetched data here
            GitLabDataView(data: statusBarItem.gitLabData)
        }
        .frame(width: 400, height: 600)
    }
}
```

Finally, update your `App.swift` file to use the `StatusBarItem` class:

swift

Copy code

```swift
import SwiftUI

@main
struct GitLabStatusBarApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}

class AppDelegate: NSObject, NSApplicationDelegate {
    var statusBarItem: StatusBarItem?

    func applicationDidFinishLaunching(_ notification: Notification) {
        statusBarItem = StatusBarItem()
    }
}
```

Now your macOS SwiftUI app should display the fetched GitLab data in a popover when the Menu Bar Item is clicked.

Please note that the provided code snippets are only a starting point and may need to be adjusted to fit your specific requirements and data model.

1 / 1

Cannot find 'HTTPNetworkTransport' in scope

1 / 1

I apologize for the confusion. It appears that the `HTTPNetworkTransport` class has been replaced with the `RequestChainNetworkTransport` class in the recent versions of Apollo iOS SDK.

Please update the `Network.swift` file to use the `RequestChainNetworkTransport` class as follows:

swift

Copy code

```swift
import Foundation
import Apollo

class Network {
    static let shared = Network()

    private(set) lazy var apollo: ApolloClient = {
        let url = URL(string: "https://gitlab.com/api/graphql")!

        let store = ApolloStore(cache: InMemoryNormalizedCache())

        let interceptorProvider = LegacyInterceptorProvider(store: store)
        let requestChainTransport =
RequestChainNetworkTransport(interceptorProvider: interceptorProvider,

endpointURL: url,

additionalHeaders: ["Authorization": "Bearer YOUR_PERSONAL_ACCESS_TOKEN"])

        return ApolloClient(networkTransport: requestChainTransport, store:
store)
    }()
}
```

Replace `YOUR_PERSONAL_ACCESS_TOKEN` with your GitLab personal access token.

With this update, you should no longer encounter the "Cannot find 'HTTPNetworkTransport' in scope" error.

1 / 1

Cannot find 'LegacyInterceptorProvider' in scope

1 / 1

I apologize for the confusion. It seems I made a mistake in my previous response. Please use the `DefaultInterceptorProvider` instead of the `LegacyInterceptorProvider`. Update the `Network.swift` file with the following code:

swift

Copy code

```
import Foundation
import Apollo

class Network {
    static let shared = Network()

    private(set) lazy var apollo: ApolloClient = {
        let url = URL(string: "https://gitlab.com/api/graphql")!

        let store = ApolloStore(cache: InMemoryNormalizedCache())

        let interceptorProvider = DefaultInterceptorProvider(store: store)
        let requestChainTransport =
RequestChainNetworkTransport(interceptorProvider: interceptorProvider,

endpointURL: url,

additionalHeaders: ["Authorization": "Bearer YOUR_PERSONAL_ACCESS_TOKEN"])

        return ApolloClient(networkTransport: requestChainTransport, store:
store)
    }()
}
```

Replace `YOUR_PERSONAL_ACCESS_TOKEN` with your GitLab personal access token.

This should resolve the "Cannot find 'LegacyInterceptorProvider' in scope" error, and your code should work with the recent versions of the Apollo iOS SDK.

1 / 1

I've updated that for my style preferences:

```
import Foundation
import Apollo

class Network {

    static let shared = Network()

    private(set) lazy var apollo: ApolloClient = {
        let url = URL(string: "https://gitlab.com/api/graphql")!

        let store = ApolloStore(cache: InMemoryNormalizedCache())

        let interceptorProvider = DefaultInterceptorProvider(store: store)
        let requestChainTransport = RequestChainNetworkTransport(
            interceptorProvider: interceptorProvider,
            endpointURL: url,
            additionalHeaders: ["Authorization": "Bearer YOUR_PERSONAL_ACCESS_TOKEN"]
        )

        return ApolloClient(networkTransport: requestChainTransport, store: store)
    }()

}
```

Now then, please write GitLabDataFetcher as you described, and stub GitLabDataView.

1 / 1

I will provide you with the `GitLabDataFetcher` and a stub for `GitLabDataView`. First, let's define a simple data model to represent the response structure. Create a new Swift file called `GitLabModels.swift` and add the following code:

swift

Copy code

```swift
import Foundation

struct GitLabProject {
    let id: String
    let fullPath: String
    let webUrl: URL
    let pipelines: [Pipeline]
    let mergeRequests: [MergeRequest]
}

struct Pipeline {
    let id: String
    let path: String
    let createdAt: Date
    let queuedDuration: TimeInterval?
    let duration: TimeInterval?
    let status: String
    let stages: [Stage]
    let testReportSummary: TestReportSummary?
}

struct Stage {
    let name: String
    let status: String
}

struct TestReportSummary {
    let failed: Int
    let count: Int
    let time: TimeInterval
}

struct MergeRequest {
    let id: String
    let webUrl: URL
    let draft: Bool
    let title: String
    let createdAt: Date
    let authorUsername: String
    let headPipeline: Pipeline?
    let approvalsLeft: Int
    let approvedBy: [Approver]
    let shouldBeRebased: Bool
}
```

```swift
struct Approver {
    let username: String
    let avatarUrl: URL?
}
```

Now, create a new Swift file called `GitLabDataFetcher.swift` and add the following code:

swift

Copy code

```swift
import Foundation
import Apollo

class GitLabDataFetcher {

    func fetchData(completion: @escaping (Result<[GitLabProject], Error>) -> Void) {
        let query = YOUR_GRAPHQL_QUERY // Replace with the name of your GraphQL query

        Network.shared.apollo.fetch(query: query) { result in
            switch result {
            case .success(let graphQLResult):
                if let errors = graphQLResult.errors {
                    completion(.failure(NSError(domain: "", code: -1, userInfo: [NSLocalizedDescriptionKey: "GraphQL errors: \(errors)"])))
                    return
                }

                guard let data = graphQLResult.data else {
                    completion(.failure(NSError(domain: "", code: -1, userInfo: [NSLocalizedDescriptionKey: "No data received"])))
                    return
                }

                // Parse the data into your GitLabProject model and related models
                let projects: [GitLabProject] = [] // Replace with parsed data

                completion(.success(projects))

            case .failure(let error):
                completion(.failure(error))
            }
        }
    }
}
```

Replace `YOUR_GRAPHQL_QUERY` with the name of your GraphQL query class.

Next, create a stub for `GitLabDataView` by creating a new SwiftUI file called `GitLabDataView.swift`:

swift

Copy code

```swift
import SwiftUI

struct GitLabDataView: View {
    let projects: [GitLabProject]

    var body: some View {
        VStack {
            // Design your view layout to display the GitLabProject data
            Text("Display your GitLabProject data here.")
        }
    }
}

struct GitLabDataView_Previews: PreviewProvider {
    static var previews: some View {
        GitLabDataView(projects: [])
    }
}
```

Now you can use the `GitLabDataFetcher` to fetch the data and display it in the `GitLabDataView`. Please note that you still need to implement the data parsing logic inside the `fetchData` method in `GitLabDataFetcher.swift`.

1 / 1

I have:

import Foundation
import Apollo

```
class GitLabDataFetcher {

    func fetchData(completion: @escaping (Result<[GitLabProject], Error>) -> Void) {
        let query = """
        query {
          projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://
gitlab/Project/10582521"]) {
            nodes {
              id
              fullPath
              webUrl
              pipelines(first: 5, ref: "master") {
                pageInfo {
                  endCursor
                }
                nodes {
                  id
                  path
                  createdAt
                  queuedDuration
                  duration
                  status
                  stages {
                    nodes {
                      name
                      status
                    }
                  }
                  testReportSummary {
                    total {
                      failed
                      count
                      time
                    }
                  }
                }
              }
              mergeRequests(state: opened, first: 10) {
                pageInfo {
                  endCursor
                }
                nodes {
                  id
                  webUrl
                  draft
                  title
                  createdAt
                  author {
                    username
                  }
                  headPipeline {
                    id
                    path
                    createdAt
                    queuedDuration
                    duration
                    status
                    stages {
                      nodes {
                        name
                        status
                      }
                    }
                    testReportSummary {
                      total {
                        failed
                        count
                        time
                      }
                    }
                  }
                  approvalsLeft
                  approvedBy {
                    nodes {
                      username
                      avatarUrl
                    }
                  }
                  shouldBeRebased
                }
              }
            }
          }
        }
        """

        Network.shared.apollo.fetch(query: query) { result in
            switch result {
            case .success(let graphQLResult):
                if let errors = graphQLResult.errors {
                    completion(.failure(NSError(domain: "", code: -1, userInfo:
[NSLocalizedDescriptionKey: "GraphQL errors: \(errors)"])))
```

```
                    return
                }

                guard let data = graphQLResult.data else {
                    completion(.failure(NSError(domain: "", code: -1, userInfo: [NSLocalizedDescriptionKey: "No data received"])))
                    return
                }

                // Parse the data into your GitLabProject model and related models
                let projects: [GitLabProject] = [] // Replace with parsed data
                completion(.success(projects))

            case .failure(let error):
                completion(.failure(error))
            }
        }
    }
}
```

But on the Network.shared.apollo.fetch() line, I get the error, Instance method 'fetch(query:cachePolicy:contextIdentifier:queue:resultHandler:)' requires that 'String' conform to 'GraphQLQuery'

1 / 1

Apologies for the confusion. You should define a GraphQL query as a Swift type conforming to the `GraphQLQuery` protocol, not as a string. To do that, you can create a new Swift file called `GitLabProjectsQuery.swift` and add the following code:

swift

Copy code

```swift
import Foundation
import Apollo

struct GitLabProjectsQuery: GraphQLQuery {
    static let operationDefinition =
    """
    query GitLabProjectsQuery {
      projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://gitlab/Project/10582521"]) {
        nodes {
          id
          fullPath
          webUrl
          pipelines(first: 5, ref: "master") {
            pageInfo {
              endCursor
            }
            nodes {
              id
              path
              createdAt
              queuedDuration
              duration
              status
              stages {
                nodes {
                  name
                  status
                }
              }
              testReportSummary {
                total {
                  failed
                  count
                  time
                }
              }
            }
          }
          mergeRequests(state: opened, first: 10) {
            pageInfo {
              endCursor
            }
            nodes {
              id
              webUrl
              draft
              title
              createdAt
              author {
                username
              }
              headPipeline {
                id
                path
                createdAt
                queuedDuration
                duration
                status
```

```
                stages {
                  nodes {
                    name
                    status
                  }
                }
                testReportSummary {
                  total {
                    failed
                    count
                    time
                  }
                }
              }
              approvalsLeft
              approvedBy {
                nodes {
                  username
                  avatarUrl
                }
              }
              shouldBeRebased
            }
          }
        }
      }
    }
    """

    struct Data: GraphQLSelectionSet {
        // Define the GraphQL types for the response here
    }
}
```

Then, update the `fetchData` method in `GitLabDataFetcher.swift` to use the `GitLabProjectsQuery` type:

swift

Copy code

```
func fetchData(completion: @escaping (Result<[GitLabProject], Error>) -> Void) {
    let query = GitLabProjectsQuery()

    Network.shared.apollo.fetch(query: query) { result in
        // ...
    }
}
```

Please note that you will need to define the GraphQL types for the response data in the `Data` struct inside `GitLabProjectsQuery`. This will depend on the generated API.swift file created by Apollo. Once you have defined the response types, you can parse the data into your `GitLabProject` model and related models within the `fetchData` method.

1 / 1

Cannot find type 'GraphQLSelectionSet' in scope

1 / 1

I apologize for the mistake in my previous response. The `GraphQLSelectionSet` protocol has been removed in Apollo iOS 0.34.0 and above. Instead, you should use the `GraphQLFragment` protocol. Update the `GitLabProjectsQuery` definition as follows:

swift

Copy code

```swift
import Foundation
import Apollo

struct GitLabProjectsQuery: GraphQLQuery {
    static let operationDefinition =
    """
    query GitLabProjectsQuery {
      projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/
250833", "gid://gitlab/Project/10582521"]) {
        nodes {
          id
          fullPath
          webUrl
          pipelines(first: 5, ref: "master") {
            pageInfo {
              endCursor
            }
            nodes {
              id
              path
              createdAt
              queuedDuration
              duration
              status
              stages {
                nodes {
                  name
                  status
                }
              }
              testReportSummary {
                total {
                  failed
                  count
                  time
                }
              }
            }
          }
          mergeRequests(state: opened, first: 10) {
            pageInfo {
              endCursor
            }
            nodes {
              id
              webUrl
              draft
              title
              createdAt
              author {
                username
              }
              headPipeline {
                id
                path
                createdAt
                queuedDuration
                duration
                status
                stages {
                  nodes {
                    name
                    status
                  }
                }
                testReportSummary {
                  total {
                    failed
                    count
                    time
                  }
                }
              }
              approvalsLeft
              approvedBy {
                nodes {
                  username
                  avatarUrl
                }
              }
              shouldBeRebased
            }
          }
        }
      }
    }
    """

    struct Data: Codable, GraphQLFragment {
        // Define the GraphQL types for the response here
    }
}
```

Now you can use the `GitLabProjectsQuery` in the `fetchData` method as described in my previous response. Remember to define the GraphQL types for the response data in the `Data` struct inside `GitLabProjectsQuery`. Once you have defined the response types, you can parse the data into your `GitLabProject` model and related models within the `fetchData` method.

1 / 1

Now I get the error: Cannot find type 'GraphQLFragment' in scope.

I am using version 1.1.1 of the library, but please just assume the latest version you know about.

1 / 1

I apologize for the confusion caused earlier. Since you are using version 1.1.1, we need to define the `Data` struct in a different way. Update the `GitLabProjectsQuery` definition as follows:

swift

Copy code

```swift
import Foundation
import Apollo

struct GitLabProjectsQuery: GraphQLQuery {
    static let operationDefinition =
    """
    query GitLabProjectsQuery {
      projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://gitlab/Project/10582521"]) {
        nodes {
          id
          fullPath
          webUrl
          pipelines(first: 5, ref: "master") {
            pageInfo {
              endCursor
            }
            nodes {
              id
              path
              createdAt
              queuedDuration
              duration
              status
              stages {
                nodes {
                  name
                  status
                }
              }
              testReportSummary {
                total {
                  failed
                  count
                  time
                }
              }
            }
          }
          mergeRequests(state: opened, first: 10) {
            pageInfo {
              endCursor
            }
            nodes {
              id
              webUrl
              draft
              title
              createdAt
              author {
                username
              }
              headPipeline {
                id
                path
                createdAt
                queuedDuration
                duration
                status
                stages {
                  nodes {
                    name
                    status
                  }
                }
                testReportSummary {
                  total {
                    failed
                    count
                    time
                  }
                }
              }
              approvalsLeft
              approvedBy {
                nodes {
                  username
                  avatarUrl
                }
              }
              shouldBeRebased
            }
          }
        }
```

```
      }
    }
    """

    public private(set) var resultMap: ResultMap

    public init(resultMap: ResultMap) {
        self.resultMap = resultMap
    }

    public var data: Data? {
        get {
            (resultMap["data"] as? ResultMap).flatMap { Data(resultMap: $0) }
        }
        set {
            resultMap.updateValue(newValue?.resultMap, forKey: "data")
        }
    }

    public struct Data: GraphQLSelectionSet {
        public static let possibleTypes: [String] = ["Query"]

        public private(set) var resultMap: ResultMap

        public init(resultMap: ResultMap) {
            self.resultMap = resultMap
        }

        // Define the GraphQL types for the response here
    }
}
```

Now you can use the `GitLabProjectsQuery` in the `fetchData` method as described in my previous responses. Remember to define the GraphQL types for the response data in the `Data` struct inside `GitLabProjectsQuery`. Once you have defined the response types, you can parse the data into your `GitLabProject` model and related models within the `fetchData` method.

1 / 1

Cannot find type 'ResultMap' in scope

Cannot find type 'GraphQLSelectionSet' in scope

1 / 1

I apologize for the confusion earlier. Let's update the `GitLabProjectsQuery` for Apollo iOS version 1.1.1 by creating the query using the Apollo codegen tool, which will generate the necessary types automatically.

First, save the GraphQL query in a separate file named `gitlab_projects_query.graphql`:

graphql

Copy code

```graphql
query GitLabProjectsQuery {
  projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://gitlab/Project/10582521"]) {
    nodes {
      id
      fullPath
      webUrl
      pipelines(first: 5, ref: "master") {
        pageInfo {
          endCursor
        }
        nodes {
          id
          path
          createdAt
          queuedDuration
          duration
          status
          stages {
            nodes {
              name
              status
            }
          }
          testReportSummary {
            total {
              failed
              count
              time
            }
          }
        }
      }
      mergeRequests(state: opened, first: 10) {
        pageInfo {
          endCursor
        }
        nodes {
          id
          webUrl
          draft
          title
          createdAt
          author {
            username
          }
          headPipeline {
            id
            path
            createdAt
            queuedDuration
            duration
            status
            stages {
              nodes {
                name
                status
              }
            }
            testReportSummary {
              total {
                failed
                count
                time
              }
            }
          }
          approvalsLeft
          approvedBy {
            nodes {
              username
              avatarUrl
            }
          }
          shouldBeRebased
        }
      }
    }
  }
}
```

Next, run the Apollo CLI to generate Swift types for the query:

bash

Copy code

```
apollo client:codegen --target=swift --includes=./path/to/your/*.graphql --
localSchemaFile=./path/to/your/schema.json API.swift
```

Make sure you have the `schema.json` file downloaded and specify the correct paths for your `.graphql` files and the `schema.json` file.

The generated `API.swift` file will contain the `GitLabProjectsQuery` type and all necessary types for the query response. Import the generated `API.swift` file into your project.

Now, you can use the `GitLabProjectsQuery` in the `fetchData` method as described in my previous responses. You can parse the data into your `GitLabProject` model and related models within the `fetchData` method.

1 / 1

I don't have the apollo CLI tool. Can I get it through SPM as well?

1 / 1

Unfortunately, the Apollo CLI tool is not available through Swift Package Manager (SPM). However, you can install it using one of the following methods:

1. Install via **Homebrew** (macOS):

bash

Copy code

```
brew install apollo-cli
```

2. Install via **npm** (cross-platform):

bash

Copy code

```
npm install -g apollo
```

You will need to have Node.js installed on your system to use `npm`.

Once the Apollo CLI is installed, you can then generate the Swift types using the command I mentioned in the previous response.

If you cannot install the Apollo CLI tool, you can try writing the `GitLabProjectsQuery` struct manually, but you'll need to define all the nested response types. This can be a tedious process, and I would recommend using the Apollo CLI tool to generate types automatically if possible.

1 / 1

Let's back up and try this again without Apollo.

1 / 1

Certainly! In this case, we will make a standard HTTP request using URLSession to fetch data from the GitLab API and then parse the JSON response manually. Here's an example of how you can do that:

1. Create a `GitLabDataFetcher` class:

swift

Copy code

```
import Foundation

class GitLabDataFetcher {
    private let gitLabApiURL = "https://gitlab.com/api/graphql"
    private let personalAccessToken = "YOUR_PERSONAL_ACCESS_TOKEN"

    func fetchData(completion: @escaping (Result<[GitLabProject], Error>) ->
Void) {
        var request = URLRequest(url: URL(string: gitLabApiURL)!)
        request.httpMethod = "POST"
        request.addValue("Bearer \(personalAccessToken)", forHTTPHeaderField:
"Authorization")
        request.addValue("application/json", forHTTPHeaderField: "Content-
Type")

        let query = """
        {
          projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/
Project/250833", "gid://gitlab/Project/10582521"]) {
            nodes {
              id
              fullPath
              webUrl
              pipelines(first: 5, ref: "master") {
                pageInfo {
                  endCursor
                }
                nodes {
                  id
                  path
                  createdAt
                  queuedDuration
                  duration
                  status
                  stages {
                    nodes {
                      name
                      status
                    }
                  }
                  testReportSummary {
                    total {
                      failed
                      count
                      time
                    }
                  }
                }
              }
              mergeRequests(state: opened, first: 10) {
                pageInfo {
                  endCursor
                }
                nodes {
                  id
                  webUrl
                  draft
                  title
                  createdAt
                  author {
                    username
                  }
                  headPipeline {
                    id
                    path
                    createdAt
                    queuedDuration
                    duration
                    status
                    stages {
                      nodes {
                        name
                        status
                      }
                    }
                    testReportSummary {
                      total {
                        failed
                        count
                        time
                      }
                    }
                  }
                  approvalsLeft
                  approvedBy {
                    nodes {
                      username
                      avatarUrl
                    }
                  }
                  shouldBeRebased
                }
              }
            }
          }
        }
        """

        let requestBody = [
            "query": query
        ]
```

```
        do {
            request.httpBody = try JSONSerialization.data(withJSONObject: requestBody, options: [])
        } catch {
            completion(.failure(error))
            return
        }

        let task = URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                completion(.failure(error))
                return
            }

            guard let data = data else {
                completion(.failure(NSError(domain: "", code: -1, userInfo: [NSLocalizedDescriptionKey: "No data received"])))
                return
            }

            do {
                let json = try JSONSerialization.jsonObject(with: data, options: []) as? [String: Any]
                // Parse the JSON response into your GitLabProject model and related models
                let projects: [GitLabProject] = [] // Replace with parsed data
                completion(.success(projects))
            } catch {
                completion(.failure(error))
            }
        }

        task.resume()
    }
}
```

2. Create your `GitLabProject` model and other related models to represent the data you want to store from the API response.
3. Call the `fetchData` method on an instance of `GitLabDataFetcher` to fetch the data:

swift

Copy code

```
let fetcher = GitLabDataFetcher()
fetcher.fetchData { result in
    switch result {
    case
```

1 / 1

Go on

1 / 1

Certainly! Continuing from where we left off, we will handle the result of the `fetchData` method:

swift

Copy code

```swift
let fetcher = GitLabDataFetcher()
fetcher.fetchData { result in
    switch result {
    case .success(let projects):
        print("Fetched projects: \(projects)")
        // Use the fetched projects data to update your UI or perform other actions
    case .failure(let error):
        print("Error fetching projects: \(error.localizedDescription)")
        // Handle the error and inform the user or perform other error handling actions
    }
}
```

With this approach, you can fetch data from the GitLab API without using the Apollo library. Just make sure to replace "YOUR_PERSONAL_ACCESS_TOKEN" with your actual GitLab Personal Access Token in the `GitLabDataFetcher` class.

You will need to parse the JSON response and map it to your `GitLabProject` model and related models within the `fetchData` method. You can use the built-in Swift `Codable` protocol to help with the parsing process, or you can do it manually depending on your preferences and requirements.

1 / 1

Please connect the fetcher to StatusBarItem so its result ends up in the @Published property

1 / 1

To connect the `GitLabDataFetcher` to a `StatusBarItem`, you need to create an `ObservableObject` that will manage the data fetching and store the fetched projects. The `@Published` property will update the SwiftUI view whenever the fetched data changes. Here's an example:

1. Create a `GitLabDataViewModel` class conforming to the `ObservableObject` protocol:

swift

Copy code

```swift
import SwiftUI
import Combine

class GitLabDataViewModel: ObservableObject {
    @Published var projects: [GitLabProject] = []
```

1 / 1

StatusBarItem is already an ObservableObject. Let's locate the results there:

import SwiftUI
import Combine

```
class StatusBarItem: NSObject, ObservableObject {

    @Published var gitLabData: [GitLabProject]?

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem
    private var popover: NSPopover

    init() {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength: NSStatusItem.variableLength)
        popover = NSPopover()

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(named: "MenuBarIcon") // Replace "MenuBarIcon"
with your custom icon name
            button.action = #selector(togglePopover(_:))
            button.target = self
        }

        // Set up the popover content
        popover.behavior = .transient
        popover.animates = true
        popover.contentViewController = NSHostingController(rootView: StatusBarMenu())
    }

    @objc func togglePopover(_ sender: AnyObject) {
        if popover.isShown {
            popover.performClose(sender)
        } else {
            if let button = statusBarItem.button {
                fetchData { [weak self] result in
                    switch result {
                    case .success(let data):
                        DispatchQueue.main.async {
                            self?.gitLabData = data
                        }
                    case .failure(let error):
                        print("Error fetching GitLab data: \(error)")
                    }
                }
                popover.show(relativeTo: button.bounds, of: button, preferredEdge: .minY)
            }
        }
    }

    func fetchData(completion: @escaping (Result<[GitLabProject], Error>) -> Void) {
        // Call the fetchData method from your GitLabDataFetcher.swift here and pass the completion handler
    }

}
```

1 / 1

Great! Now that you have added the `gitLabData` property to your `StatusBarItem` class, let's update the `fetchData` method in the `StatusBarItem` class to use `GitLabDataFetcher`.

swift

Copy code

```swift
func fetchData(completion: @escaping (Result<[GitLabProject], Error>) ->
Void) {
    let fetcher = GitLabDataFetcher()
    fetcher.fetchData { result in
        completion(result)
    }
}
```

Now, every time the popover is shown, the `fetchData` method will be called to update the `gitLabData` property. To display this data in the `StatusBarMenu`, you can pass the `gitLabData` property to your `StatusBarMenu` view:

swift

Copy code

```swift
popover.contentViewController = NSHostingController(rootView:
StatusBarMenu(gitLabData: $gitLabData))
```

Make sure to update the `StatusBarMenu` view to accept a `Binding` to an optional `[GitLabProject]` array and display the data accordingly:

swift

Copy code

```swift
struct StatusBarMenu: View {
    @Binding var gitLabData: [GitLabProject]?

    var body: some View {
        VStack {
            // Display your gitLabData here
        }
        .frame(width: 200, height: 200) // Adjust the frame size to fit your content
    }
}
```

With this setup, the `gitLabData` property will be updated with the fetched projects whenever the popover is shown, and the `StatusBarMenu` view will be updated accordingly. Make sure to replace "MenuBarIcon" with your custom icon name for the status bar item, and update the `StatusBarMenu` view to display the fetched GitLab data as desired.

1 / 1

I have:

```
import SwiftUI

struct StatusBarMenu: View {

    var gitLabData: [GitLabProject]?

    var body: some View {
        VStack {
            // Display your gitLabData here
            Text("todo")
        }
        .frame(width: 200, height: 200) // Adjust the frame size to fit your content
    }

}

---

import SwiftUI
import Combine
```

```
class StatusBarItem: NSObject, ObservableObject {

    @Published var gitLabData: [GitLabProject]?

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem
    private var popover: NSPopover

    override init() {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength: NSStatusItem.variableLength)
        popover = NSPopover()

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(named: "MenuBarIcon") // Replace "MenuBarIcon"
with your custom icon name
            button.action = #selector(togglePopover(_:))
            button.target = self
        }

        // Set up the popover content
        popover.behavior = .transient
        popover.animates = true
        popover.contentViewController = NSHostingController(rootView:
StatusBarMenu(gitLabData: gitLabData))
    }

    @objc func togglePopover(_ sender: AnyObject) {
        if popover.isShown {
            popover.performClose(sender)
        } else {
            if let button = statusBarItem.button {
                fetchData { [weak self] result in
                    switch result {
                    case .success(let data):
                        DispatchQueue.main.async {
                            self?.gitLabData = data
                        }
                    case .failure(let error):
                        print("Error fetching GitLab data: \(error)")
                    }
                }
                popover.show(relativeTo: button.bounds, of: button, preferredEdge: .minY)
            }
        }
    }

    func fetchData(completion: @escaping (Result<[GitLabProject], Error>) -> Void) {
        let fetcher = GitLabDataFetcher()
        fetcher.fetchData { result in
            completion(result)
        }
    }

}
```

```
import SwiftUI

@main
struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}

class AppDelegate: NSObject, NSApplicationDelegate {
    var statusBarItem: StatusBarItem?

    func applicationDidFinishLaunching(_ notification: Notification) {
        statusBarItem = StatusBarItem()
    }
}
```

---

n

```
struct GitLabProject {
    let id: String
    let fullPath: String
    let webUrl: URL
    let pipelines: [Pipeline]
    let mergeRequests: [MergeRequest]
}

struct Pipeline {
    let id: String
    let path: String
    let createdAt: Date
    let queuedDuration: TimeInterval?
    let duration: TimeInterval?
    let status: String
    let stages: [Stage]
    let testReportSummary: TestReportSummary?
}

struct Stage {
    let name: String
    let status: String
}

struct TestReportSummary {
    let failed: Int
    let count: Int
    let time: TimeInterval
}

struct MergeRequest {
    let id: String
    let webUrl: URL
    let draft: Bool
    let title: String
    let createdAt: Date
    let authorUsername: String
    let headPipeline: Pipeline?
    let approvalsLeft: Int
    let approvedBy: [Approver]
    let shouldBeRebased: Bool
}

struct Approver {
    let username: String
    let avatarUrl: URL?
}
```

Your code looks good so far. You just need to make a small change in the

`StatusBarItem` class to pass a binding to the `gitLabData` property when creating the `StatusBarMenu` view. Update the `popover.contentViewController` line in `StatusBarItem` as follows:

swift

Copy code

```
popover.contentViewController = NSHostingController(rootView: 
StatusBarMenu(gitLabData: $gitLabData))
```

This will ensure that the `gitLabData` property updates are reflected in the `StatusBarMenu` view.

Next, update the `StatusBarMenu` view to accept a `Binding` to an optional `[GitLabProject]` array, as shown below:

swift

Copy code

```
struct StatusBarMenu: View {
    @Binding var gitLabData: [GitLabProject]?

    var body: some View {
        VStack {
            // Display your gitLabData here
            if let projects = gitLabData {
                ForEach(projects) { project in
                    Text(project.fullPath)
                }
            } else {
                Text("No data available")
            }
        }
        .frame(width: 200, height: 200) // Adjust the frame size to fit your 
content
    }
}
```

Make sure to update the `GitLabProject` struct to conform to the `Identifiable` protocol:

swift

Copy code

```
struct GitLabProject: Identifiable {
    let id: String
    let fullPath: String
    let webUrl: URL
    let pipelines: [Pipeline]
    let mergeRequests: [MergeRequest]
}
```

This code will display the full paths of the fetched projects in the popover view. Update the `StatusBarMenu` view to display other properties of the `gitLabData` as desired.

1 / 1

One change to make: Since StatusBarItem is an ObservableObject, let's supply it in the environment.

1 / 1

To supply the `StatusBarItem` instance in the environment, you need to make a couple of changes to your code. First, create an `EnvironmentKey` for the `StatusBarItem`, like this:

swift

Copy code

```swift
struct StatusBarItemKey: EnvironmentKey {
    static let defaultValue: StatusBarItem?
```

1 / 1

No, we can supply it as an environment object.

1 / 1

I apologize for the confusion. If you want to supply `StatusBarItem` as an environment object, you can do so by updating the `popover.contentViewController` line in the `StatusBarItem` class as follows:

swift

Copy code

```swift
popover.contentViewController = NSHostingController(rootView: StatusBarMenu().environmentObject(self))
```

Next, update the `StatusBarMenu` view to use the `@EnvironmentObject` property wrapper for the `statusBarItem`:

swift

Copy code

```swift
struct StatusBarMenu: View {
    @EnvironmentObject var statusBarItem: StatusBarItem

    var body: some View {
        VStack {
            // Display your gitLabData here
            if let projects = statusBarItem.gitLabData {
                ForEach(projects) { project in
                    Text(project.fullPath)
                }
            } else {
                Text("No data available")
            }
        }
        .frame(width: 200, height: 200) // Adjust the frame size to fit your content
    }
}
```

This way, the `StatusBarMenu` view will have access to the `StatusBarItem` instance as an environment object and can access its `gitLabData` property.

1 / 1

Great. Now let's add a refresh button to the view that causes the status bar item to reload its data again, updating the view through the @Published property.

1 / 1

To add a refresh button to the `StatusBarMenu` view, you can create a Button with an action that calls the `fetchData` method of the `StatusBarItem`. Here's the updated `StatusBarMenu` view:

swift

Copy code

```swift
struct StatusBarMenu: View {
    @EnvironmentObject var statusBarItem: StatusBarItem

    var body: some View {
        VStack {
            // Display your gitLabData here
            if let projects = statusBarItem.gitLabData {
                ForEach(projects, id: \.id) { project in
                    Text(project.fullPath)
                }
            } else {
                Text("No data available")
            }

            // Add a refresh button
            Button("Refresh") {
                statusBarItem.fetchData { result in
                    switch result {
                    case .success(let data):
                        DispatchQueue.main.async {
                            statusBarItem.gitLabData = data
                        }
                    case .failure(let error):
                        print("Error fetching GitLab data: \(error)")
                    }
                }
            }
            .padding(.top, 10)
        }
        .frame(width: 200, height: 200) // Adjust the frame size to fit your content
    }
}
```

This code adds a "Refresh" button below the list of projects. When clicked, it triggers the `fetchData` method of the `StatusBarItem`, which updates the `@Published` `gitLabData` property, causing the view to refresh with the new data.

1 / 1

Move the work done in the action of the Refresh button to a method of StatusBarItem.

1 / 1

To move the work done in the action of the Refresh button to a method of `StatusBarItem`, you can create a new method called `refreshData` inside the `StatusBarItem` class. Here's the updated `StatusBarItem`:

swift

Copy code

```swift
class StatusBarItem: NSObject, ObservableObject {

    @Published var gitLabData: [GitLabProject]?

    // ... other properties and initializers

    @objc func togglePopover(_ sender: AnyObject) {
        // ... togglePopover implementation
    }

    func fetchData(completion: @escaping (Result<[GitLabProject], Error>) ->
Void) {
        // ... fetchData implementation
    }

    func refreshData() {
        fetchData { [weak self] result in
            switch result {
            case .success(let data):
                DispatchQueue.main.async {
                    self?.gitLabData = data
                }
            case .failure(let error):
                print("Error fetching GitLab data: \(error)")
            }
        }
    }
}
```

Now, in your `StatusBarMenu` view, call the `refreshData` method of the `statusBarItem` instead of directly calling `fetchData`:

swift

Copy code

```swift
struct StatusBarMenu: View {
    @EnvironmentObject var statusBarItem: StatusBarItem

    var body: some View {
        VStack {
            // Display your gitLabData here
            if let projects = statusBarItem.gitLabData {
                ForEach(projects, id: \.id) { project in
                    Text(project.fullPath)
                }
            } else {
                Text("No data available")
            }

            // Add a refresh button
            Button("Refresh") {
                statusBarItem.refreshData()
            }
            .padding(.top, 10)
        }
        .frame(width: 200, height: 200) // Adjust the frame size to fit your content
    }
}
```

Now, the Refresh button calls the `refreshData` method of the `StatusBarItem`, which in turn calls the `fetchData` method and updates the `gitLabData` property.

1 / 1

Here's my updated data model:

```
struct GitLabProjectsResponse: Codable {
    let data: Data

    struct Data: Codable {
        let projects: Projects
    }

    struct Projects: Codable {
        let nodes: [GitLabProject]
    }
}

struct GitLabProject: Identifiable, Codable {
    let id: String
    let fullPath: String
    let webUrl: URL
    let pipelines: Pipelines
    let mergeRequests: MergeRequests

    struct Pipelines: Codable {
        let pageInfo: PageInfo
        let nodes: [Pipeline]
    }

    struct MergeRequests: Codable {
        let pageInfo: PageInfo
        let nodes: [MergeRequest]
    }

    struct PageInfo: Codable {
        let endCursor: String
    }
}

struct Pipeline: Codable {
    let id: String
    let path: String
    let createdAt: String
    let queuedDuration: TimeInterval?
    let duration: TimeInterval?
    let status: String
    let stages: Stages
    let testReportSummary: TestReportSummary?

    struct Stages: Codable {
        let nodes: [Stage]
    }
}

struct Stage: Codable {
    let name: String
    let status: String
}

struct TestReportSummary: Codable {
    let total: Total

    struct Total: Codable {
        let failed: Int
        let count: Int
        let time: TimeInterval
    }
}

struct MergeRequest: Codable {
    let id: String
    let webUrl: URL
    let draft: Bool
    let title: String
    let createdAt: String
    let author: Author
    let headPipeline: Pipeline?
    let approvalsLeft: Int
    let approvedBy: ApprovedBy
    let shouldBeRebased: Bool

    struct Author: Codable {
        let username: String
    }

    struct ApprovedBy: Codable {
        let nodes: [Approver]
    }
}
```

```
struct Approver: Codable {
    let username: String
    let avatarUrl: URL?
}
```

I have the data loading from GitLab now. Thank you for your help so far.

Please update the view to be taller, a little wider, and tabbed. In the first tab, show a list of merge requests with their statuses and other details, sectioned by project. In the second tab, show a list of pipelines with their statuses by stage, also sectioned by project. Keep the refresh button outside the tab pages.

1 / 1

I'm glad you have the data loading from GitLab now. To create a taller, wider, and tabbed view, we can use `TabView` with `List` to display the merge requests and pipelines. Here's the updated `StatusBarMenu` view:

swift

Copy code

```
struct StatusBarMenu: View {
    @EnvironmentObject var statusBarItem: StatusBarItem

    @State private var selectedTab = 0

    var body: some View {
        VStack {
            TabView(selection: $selectedTab) {
                // Merge requests tab
                List {
                    if let projects = statusBarItem.gitLabData {
                        ForEach(projects) { project in
                            Section(header: Text(project.fullPath)) {
                                ForEach(project.mergeRequests.nodes)
{ mergeRequest in
                                    VStack(alignment: .leading, spacing: 4) {
                                        Text("Title: \(mergeRequest.title)")
                                        Text("Author: \
(mergeRequest.author.username)")
                                        Text("Status: \(mergeRequest.draft ?
"Draft" : "Open")")
                                        Text("Approvals left: \
(mergeRequest.approvalsLeft)")
                                        Text("Created at: \
(mergeRequest.createdAt)")
```

```
                                    }
                                }
                            }
                        }
                    }
                }
                .tabItem {
                    Text("Merge Requests")
                    Image(systemName: "list.bullet")
                }
                .tag(0)

                // Pipelines tab
                List {
                    if let projects = statusBarItem.gitLabData {
                        ForEach(projects) { project in
                            Section(header: Text(project.fullPath)) {
                                ForEach(project.pipelines.nodes) { pipeline
in
                                    VStack(alignment: .leading, spacing: 4) {
                                        Text("Created at: \
(pipeline.createdAt)")
                                        Text("Duration: \
(pipeline.duration ?? 0, specifier: "%.2f") seconds")
                                        Text("Status: \(pipeline.status)")

                                        VStack(alignment: .leading, spacing:
2) {
                                            Text("Stages:")

                                            ForEach(pipeline.stages.nodes)
{ stage in
                                                Text("\(stage.name) - \
(stage.status)")
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
                .tabItem {
                    Text("Pipelines")
                    Image(systemName: "square.and.pencil")
                }
                .tag(1)
```

```
            }

            // Refresh button
            Button("Refresh") {
                statusBarItem.refreshData()
            }
            .padding(.top, 10)
        }
        .frame(width: 250, height: 400) // Adjust the frame size to fit your content
    }
}
```

This view contains a `TabView` with two tabs: one for merge requests and the other for pipelines. Each tab displays a list of items sectioned by the project. The Refresh button is placed outside the `TabView`, so it remains visible regardless of the selected tab. The view's width and height have been increased to accommodate the additional content.

1 / 1

I want to consume the data from the main view, too, but I can't create the StatusBarItem immediately, so let's factor the data source into its own ObservableObject separate from the StatusBarItem class.

1 / 1

To create a separate `ObservableObject` for the data source, you can create a new class called `GitLabDataStore`. This class will contain the `@Published` property for `gitLabData`, and the methods for fetching and refreshing data. The `StatusBarItem` and your main view will both be able to access the same instance of this `GitLabDataStore`:

swift

Copy code

```swift
import Combine

class GitLabDataStore: ObservableObject {
    @Published var gitLabData: [GitLabProject]?
    let dataFetcher: GitLabDataFetcher

    init(fetcher: GitLabDataFetcher = GitLabDataFetcher()) {
        self.dataFetcher = fetcher
    }

    func fetchData(completion: @escaping (Result<[GitLabProject], Error>) -> Void) {
        dataFetcher.fetchData { result in
            completion(result)
        }
    }

    func refreshData() {
        fetchData { [weak self] result in
            switch result {
            case .success(let data):
                DispatchQueue.main.async {
                    self?.gitLabData = data
                }
            case .failure(let error):
                print("Error fetching GitLab data: \(error)")
            }
        }
    }
}
```

Update the `StatusBarItem` to use `GitLabDataStore`:

swift

Copy code

```swift
class StatusBarItem: NSObject, ObservableObject {
    private(set) var dataStore: GitLabDataStore

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem
    private var popover: NSPopover

    init(dataStore: GitLabDataStore) {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength: NSStatusItem.variableLength)
        popover = NSPopover()
        self.dataStore = dataStore

        super.init()
        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(named: "MenuBarIcon") // Replace
"MenuBarIcon" with your custom icon name
            button.action = #selector(togglePopover(_:))
            button.target = self
        }

        // Set up the popover content
        popover.behavior = .transient
        popover.animates = true
        popover.contentViewController = NSHostingController(rootView:
StatusBarMenu().environmentObject(dataStore))
    }

    @objc func togglePopover(_ sender: AnyObject) {
        if popover.isShown {
            popover.performClose(sender)
        } else {
            if let button = statusBarItem.button {
                dataStore.refreshData()
                popover.show(relativeTo: button.bounds, of: button,
preferredEdge: .minY)
            }
        }
    }
}
```

Now, you can create an instance of `GitLabDataStore` and pass it to both the `StatusBarItem` and your main view. To do this, update your `AppDelegate` class:

swift

Copy code

```
class AppDelegate: NSObject, NSApplicationDelegate {
    var statusBarItem: StatusBarItem?
    var gitLabDataStore: GitLabDataStore?

    func applicationDidFinishLaunching(_ notification: Notification) {
        gitLabDataStore = GitLabDataStore()
        statusBarItem = StatusBarItem(dataStore: gitLabDataStore!)
        gitLabDataStore?.refreshData()
    }
}
```

You can now access the `gitLabDataStore` in your main view by adding it as an `EnvironmentObject`.

1 / 1

That's great. Now let's also add the GitLabDataStore as an environment object for the ContentView in the main window.

1 / 1

To add the `GitLabDataStore` as an environment object for the `ContentView` in the main window, you'll need to modify the `GitLabStatusMenuBarItemApp` struct. First, add a property for the `GitLabDataStore` and initialize it. Then, pass the `gitLabDataStore` instance to the `ContentView` as an environment object:

swift

Copy code

```swift
import SwiftUI

@main
struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate
    @StateObject private var gitLabDataStore = GitLabDataStore()

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(gitLabDataStore)
        }
    }
}
```

Now, your `ContentView` will have access to the `gitLabDataStore`. You can use the `@EnvironmentObject` property wrapper in your main view to access it:

swift

Copy code

```swift
import SwiftUI

struct ContentView: View {
    @EnvironmentObject var gitLabDataStore: GitLabDataStore

    var body: some View {
        // Your view content here, utilizing the gitLabDataStore
    }
}
```

Finally, make sure to update your `AppDelegate` class to use the `gitLabDataStore` instance from the `GitLabStatusMenuBarItemApp`:

swift

Copy code

```
class AppDelegate: NSObject, NSApplicationDelegate {
    var statusBarItem: StatusBarItem?
    @Environment(\.gitLabDataStore) private var gitLabDataStore

    func applicationDidFinishLaunching(_ notification: Notification) {
        statusBarItem = StatusBarItem(dataStore: gitLabDataStore)
        gitLabDataStore?.refreshData()
    }
}
```

Now, your `ContentView` and the `StatusBarItem` will both have access to the same `GitLabDataStore` instance, and they will be able to consume the data and update it as needed.

1 / 1

The AppDelegate can't use @Environment, so let's make the data store a shared instance.

1 / 1

You can create a shared instance of the `GitLabDataStore` by implementing the singleton pattern. Update your `GitLabDataStore` class to have a static `shared` property and a private initializer to ensure only one instance is created:

swift

Copy code

```swift
import SwiftUI
import Combine

class GitLabDataStore: ObservableObject {
    static let shared = GitLabDataStore()

    @Published var gitLabData: [GitLabProject]?

    private init() { }

    // ... (rest of the GitLabDataStore class)
}
```

Now, you can use the shared instance of `GitLabDataStore` in your `GitLabStatusMenuBarItemApp` struct:

swift

Copy code

```swift
import SwiftUI

@main
struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(GitLabDataStore.shared)
        }
    }
}
```

Also, update your `AppDelegate` class to use the shared instance of `GitLabDataStore`:

swift

Copy code

```swift
class AppDelegate: NSObject, NSApplicationDelegate {
    var statusBarItem: StatusBarItem?

    func applicationDidFinishLaunching(_ notification: Notification) {
        statusBarItem = StatusBarItem(dataStore: GitLabDataStore.shared)
        GitLabDataStore.shared.refreshData()
    }
}
```

Now, your `ContentView` and the `StatusBarItem` will both have access to the shared instance of `GitLabDataStore`, allowing them to consume the data and update it as needed.

1 / 1

I chose to create the shared instance property in a private extension in the same file as my app and app delegate, to avoid it being used anywhere else.

I replaced the @EnvironmentObject property in my status bar menu view and called it "store".

1 / 1

That's a good way to limit the access to the shared instance of `GitLabDataStore`. With this change, you'll need to update the `StatusBarMenu` view to use the new `store` property instead of the `@EnvironmentObject`.

swift

Copy code

```swift
import SwiftUI

struct StatusBarMenu: View {
    @ObservedObject var store: GitLabDataStore

    var body: some View {
        VStack {
            // Use store.gitLabData to display your data here
            Text("todo")
            Button("Refresh") {
                store.refreshData()
            }
        }
        .frame(width: 300, height: 400) // Adjust the frame size to fit your
content
    }
}
```

Now, the `StatusBarMenu` view will use the `store` property to access the data and call the `refreshData()` method. The shared instance of `GitLabDataStore` is limited to the app and app delegate, as you intended.

1 / 1

Here are my views as they stand now:

```
struct ContentView: View {
    var body: some View {
        StatusBarMenu()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

struct StatusBarMenu: View {

    @EnvironmentObject var store: GitLabDataStore

    @State private var selectedTab = 0

    var body: some View {
        VStack {
            TabView(selection: $selectedTab) {
                // Merge requests tab
                List {
                    if let projects = store.gitLabData {
                        ForEach(projects) { project in
                            Section(header: Text(project.fullPath)) {
                                ForEach(project.mergeRequests.nodes) { mergeRequest in
                                    VStack(alignment: .leading, spacing: 4) {
                                        Text("Title: \(mergeRequest.title)")
                                        Text("Author: \(mergeRequest.author.username)")
                                        Text("Status: \(mergeRequest.draft ? "Draft" : "Open")")
                                        Text("Approvals left: \(mergeRequest.approvalsLeft)")
                                        Text("Created at: \(mergeRequest.createdAt)")
                                    }
                                }
                            }
                        }
                    }
                }
                .tabItem {
                    Text("Merge Requests")
                    Image(systemName: "list.bullet")
                }
                .tag(0)

                // Pipelines tab
                List {
                    if let projects = store.gitLabData {
                        ForEach(projects) { project in
                            Section(header: Text(project.fullPath)) {
                                ForEach(project.pipelines.nodes) { pipeline in
                                    VStack(alignment: .leading, spacing: 4) {
                                        Text("Created at: \(pipeline.createdAt)")
                                        Text("Duration: \(pipeline.duration ?? 0, specifier: "%.2f")
seconds")
                                        Text("Status: \(pipeline.status)")

                                        VStack(alignment: .leading, spacing: 2) {
                                            Text("Stages:")

                                            ForEach(pipeline.stages.nodes) { stage in
                                                Text("\(stage.name) - \(stage.status)")
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
                .tabItem {
                    Text("Pipelines")
                    Image(systemName: "square.and.pencil")
                }
                .tag(1)
            }
            // Refresh button
            Button("Refresh") {
                store.refreshData()
            }
            .padding(.top, 10)
        }
        .frame(width: 250, height: 400) // Adjust the frame size to fit your content
    }

}
```

Please factor StatusBarMenu into subviews for each of the two lists and each kind of list item.

1 / 1

You can refactor your `StatusBarMenu` view into separate subviews for better organization and readability. Here's an example of how to do this:

swift

Copy code

```swift
import SwiftUI

struct StatusBarMenu: View {

    @EnvironmentObject var store: GitLabDataStore

    @State private var selectedTab = 0

    var body: some View {
        VStack {
            TabView(selection: $selectedTab) {
                MergeRequestsListView()
                    .tabItem {
                        Text("Merge Requests")
                        Image(systemName: "list.bullet")
                    }
                    .tag(0)

                PipelinesListView()
                    .tabItem {
                        Text("Pipelines")
                        Image(systemName: "square.and.pencil")
                    }
                    .tag(1)
            }

            // Refresh button
            Button("Refresh") {
                store.refreshData()
            }
            .padding(.top, 10)
        }
        .frame(width: 250, height: 400) // Adjust the frame size to fit your
content
    }
}

struct MergeRequestsListView: View {
    @EnvironmentObject var store: GitLabDataStore

    var body: some View {
        List {
            if let projects = store.gitLabData {
                ForEach(projects) { project in
                    Section(header: Text(project.fullPath)) {
                        ForEach(project.mergeRequests.nodes) { mergeRequest
in
                            MergeRequestRow(mergeRequest: mergeRequest)
                        }
                    }
                }
            }
        }
    }
}

struct MergeRequestRow: View {
    var mergeRequest: MergeRequest

    var body: some View {
        VStack(alignment: .leading, spacing: 4) {
            Text("Title: \(mergeRequest.title)")
            Text("Author: \(mergeRequest.author.username)")
            Text("Status: \(mergeRequest.draft ? "Draft" : "Open")")
            Text("Approvals left: \(mergeRequest.approvalsLeft)")
            Text("Created at: \(mergeRequest.createdAt)")
        }
    }
}

struct PipelinesListView: View {
    @EnvironmentObject var store: GitLabDataStore

    var body: some View {
        List {
            if let projects = store.gitLabData {
                ForEach(projects) { project in
                    Section(header: Text(project.fullPath)) {
                        ForEach(project.pipelines.nodes) { pipeline in
                            PipelineRow(pipeline: pipeline)
                        }
                    }
                }
            }
        }
    }
}

struct PipelineRow: View {
    var pipeline: Pipeline

    var body: some View {
        VStack(alignment: .leading, spacing: 4) {
            Text("Created at: \(pipeline.createdAt)")
            Text("Duration: \(pipeline.duration ?? 0, specifier: "%.2f") seconds")
            Text("Status: \(pipeline.status)")

            VStack(alignment: .leading, spacing: 2) {
                Text("Stages:")

                ForEach(pipeline.stages.nodes) { stage in
                    Text("\(stage.name) - \(stage.status)")
                }
            }
        }
    }
}
```

In this refactored version, the `StatusBarMenu` view is now composed of `MergeRequestsListView` and `PipelinesListView` subviews, each with its own list of items. Additionally, the `MergeRequestRow` and `PipelineRow` views are used to display individual list items in each list.

Please create preview providers for each of these four new views with appropriate data.

1 / 1

Here are the preview providers for each of the four new views, along with example data for the previews:

swift

Copy code

```
// Preview providers

struct MergeRequestsListView_Previews: PreviewProvider {
    static var previews: some View {
        MergeRequestsListView()
            .environmentObject(GitLabDataStore.exampleStore)
    }
}

struct MergeRequestRow_Previews: PreviewProvider {
    static var previews: some View {
        MergeRequestRow(mergeRequest:
GitLabDataStore.exampleStore.gitLabData![0].mergeRequests.nodes[0])
    }
}

struct PipelinesListView_Previews: PreviewProvider {
    static var previews: some View {
        PipelinesListView()
            .environmentObject(GitLabDataStore.exampleStore)
    }
}

struct PipelineRow_Previews: PreviewProvider {
    static var previews: some View {
        PipelineRow(pipeline: GitLabDataStore.exampleStore.gitLabData!
[0].pipelines.nodes[0])
    }
}

// Example data in GitLabDataStore

extension GitLabDataStore {
    static var exampleStore: GitLabDataStore {
        let store = GitLabDataStore()
        store.gitLabData = [
            GitLabProject(id: "1", fullPath: "Example Project", webUrl: URL(string: "https://example.com")!, pipelines: examplePipelines, mergeRequests: exampleMergeRequests)
        ]
        return store
    }

    static var examplePipelines: GitLabProject.Pipelines {
        let pageInfo = GitLabProject.PageInfo(endCursor: "cursor")
        let pipeline = Pipeline(id: "1", path: "/path", createdAt: "2023-01-01", queuedDuration: 5, duration: 10, status: "success", stages: exampleStages, testReportSummary: exampleTestReportSummary)
        return GitLabProject.Pipelines(pageInfo: pageInfo, nodes: [pipeline])
    }

    static var exampleStages: Pipeline.Stages {
        let stage = Stage(name: "Build", status: "success")
        return Pipeline.Stages(nodes: [stage])
    }

    static var exampleTestReportSummary: TestReportSummary {
        let total = TestReportSummary.Total(failed: 0, count: 5, time: 12)
        return TestReportSummary(total: total)
    }

    static var exampleMergeRequests: GitLabProject.MergeRequests {
        let pageInfo = GitLabProject.PageInfo(endCursor: "cursor")
        let author = MergeRequest.Author(username: "user1")
        let approvedBy = MergeRequest.ApprovedBy(nodes: [Approver(username: "user2", avatarUrl: URL(string: "https://example.com/avatar")!)])
        let mergeRequest = MergeRequest(id: "1", webUrl: URL(string: "https://example.com/mr")!, draft: false, title: "Example Merge Request", createdAt: "2023-01-01", author: author, headPipeline: examplePipelines.nodes[0], approvalsLeft: 1, approvedBy: approvedBy, shouldBeRebased: false)
        return GitLabProject.MergeRequests(pageInfo: pageInfo, nodes: [mergeRequest])
    }
}
```

I have added example data to the `GitLabDataStore` using a `static var exampleStore` and created preview providers for `MergeRequestsListView`, `MergeRequestRow`, `PipelinesListView`, and `PipelineRow` views. These preview providers use the example data to display previews with appropriate data in Xcode's canvas.

1 / 1

Add a preview provider for StatusBarMenu too.

Now let's work on the appearance of PipelineRow. It should be about as thick as a finger. Across the row, show an icon for the overall status, then the branch/ref name (which we must add to the model too), then if the status is not started show the queued duration, otherwise show the actual duration and the statuses of each stage.

1 / 1

First, let's add a preview provider for `StatusBarMenu`:

swift

Copy code

```swift
struct StatusBarMenu_Previews: PreviewProvider {
    static var previews: some View {
        StatusBarMenu()
            .environmentObject(GitLabDataStore.exampleStore)
    }
}
```

Next, we need to update the data model to include the branch/ref name in the `Pipeline` structure. Add the following line inside the `Pipeline` structure:

swift

Copy code

```swift
let ref: String
```

Now, we can update the `PipelineRow` view to include the desired appearance:

swift

Copy code

```
struct PipelineRow: View {
    let pipeline: Pipeline

    var body: some View {
        HStack {
            Image(systemName: pipeline.status == "success" ? "checkmark.circle" : "xmark.circle")
                .resizable()
                .scaledToFit()
                .frame(height: 24)
                .foregroundColor(pipeline.status == "success" ? Color.green : Color.red)

            Text(pipeline.ref)
                .font(.headline)

            Spacer()

            if pipeline.status == "not_started" {
                Text("Queued: \(pipeline.queuedDuration ?? 0, specifier: "%.2f")s")
            } else {
                Text("Duration: \(pipeline.duration ?? 0, specifier: "%.2f")s")
                HStack(spacing: 4) {
                    ForEach(pipeline.stages.nodes) { stage in
                        Text(stage.status.prefix(1))
                            .padding(4)
                            .background(statusColor(for: stage.status))
                            .clipShape(Circle())
                            .foregroundColor(.white)
                            .font(.footnote)
                    }
                }
            }
        }
        .padding(.vertical, 8)
    }

    private func statusColor(for status: String) -> Color {
        switch status {
        case "success":
            return .green
        case "failed":
            return .red
        case "running":
            return .blue
        default:
            return .gray
        }
    }
}
```

This updated `PipelineRow` view includes an icon for the overall status, the branch/ref name, and the queued or actual duration. If the pipeline has started, it also shows the statuses of each stage as circular badges with the first letter of the status. The height of the row is approximately the height of a finger.

1 / 1

That's very good. Now let's do something similar with the merge request rows. Also, for both kinds of item, clicking them should open the pipeline or merge request in a web browser.

1 / 1

First, let's update the `MergeRequestRow` view to have a similar appearance to `PipelineRow`. In this case, we'll show an icon for the overall status (draft or open), the title of the merge request, and the author's username:

swift

Copy code

```
struct MergeRequestRow: View {
    let mergeRequest: MergeRequest

    var body: some View {
        Button(action: {
            NSWorkspace.shared.open(mergeRequest.webUrl)
        }) {
            HStack {
                Image(systemName: mergeRequest.draft ? "pencil.circle" :
"checkmark.circle")
                    .resizable()
                    .scaledToFit()
                    .frame(height: 24)
                    .foregroundColor(mergeRequest.draft ? Color.orange :
Color.green)

                VStack(alignment: .leading, spacing: 2) {
                    Text(mergeRequest.title)
                        .font(.headline)
                    Text("Author: \(mergeRequest.author.username)")
                        .font(.footnote)
                        .foregroundColor(.gray)
                }

                Spacer()
            }
            .padding(.vertical, 8)
        }
        .buttonStyle(PlainButtonStyle())
    }
}
```

Now let's make the `PipelineRow` clickable, opening the pipeline in a web browser. We'll wrap the existing `HStack` inside a `Button`:

swift

Copy code

```
struct PipelineRow: View {
    let pipeline: Pipeline

    var body: some View {
        Button(action: {
            if let webUrl = URL(string: pipeline.path) {
                NSWorkspace.shared.open(webUrl)
            }
        }) {
            HStack {
                // The existing HStack content
            }
            .padding(.vertical, 8)
        }
        .buttonStyle(PlainButtonStyle())
    }

    // The rest of the PipelineRow view
}
```

These updates will make the rows for both merge requests and pipelines clickable, opening the corresponding web page in a browser when clicked.

1 / 1

Let's revise what we show in MergeRequestRow. Let's show an indicator if it's a draft, then the title with the author and branch underneath, then show an indicator if no approvals are needed and no rebasing is needed, or show a different indicator if no approvals are needed and a rebase is needed, or else show the number of approvals needed, and finally show user avatars for those who have approved thus far.

1 / 1

Here's the updated `MergeRequestRow` view incorporating your suggestions:

swift

Copy code

```swift
struct MergeRequestRow: View {
    let mergeRequest: MergeRequest

    var body: some View {
        Button(action: {
            NSWorkspace.shared.open(mergeRequest.webUrl)
        }) {
            HStack {
                if mergeRequest.draft {
                    Image(systemName: "pencil.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.orange)
                }

                VStack(alignment: .leading, spacing: 2) {
                    Text(mergeRequest.title)
                        .font(.headline)
                    Text("Author: \(mergeRequest.author.username)")
                        .font(.footnote)
                        .foregroundColor(.gray)
                    Text("Branch: ...") // Replace with the actual branch
name when available
                        .font(.footnote)
                        .foregroundColor(.gray)
                }

                Spacer()

                if mergeRequest.approvalsLeft == 0 && !
mergeRequest.shouldBeRebased {
                    Image(systemName: "checkmark.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.green)
                } else if mergeRequest.approvalsLeft == 0 &&
mergeRequest.shouldBeRebased {
                    Image(systemName: "arrow.clockwise.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.blue)
                } else {
                    Text("\(mergeRequest.approvalsLeft)")
                        .font(.headline)
                        .foregroundColor(.red)
                }

                HStack {
                    ForEach(mergeRequest.approvedBy.nodes, id: \.username)
{ approver in
                        if let avatarUrl = approver.avatarUrl {
                            AsyncImage(url: avatarUrl) { phase in
                                if let image = phase.image {
                                    image
                                        .resizable()
                                        .scaledToFit()
                                        .frame(width: 24, height: 24)
                                        .clipShape(Circle())
                                } else {
                                    Circle()
                                        .fill(Color.gray)
                                        .frame(width: 24, height: 24)
                                }
                            }
                        } else {
                            Circle()
```

```
                                .fill(Color.gray)
                                .frame(width: 24, height: 24)
                        }
                    }
                }
            }
            .padding(.vertical, 8)
        }
        .buttonStyle(PlainButtonStyle())
    }
}
```

This view now shows a draft indicator if the merge request is a draft, the title with the author and branch underneath, an indicator if no approvals are needed and no rebasing is needed (green checkmark), an indicator if no approvals are needed and a rebase is needed (blue arrow), the number of approvals needed otherwise, and user avatars for those who have approved the merge request. Replace the "Branch: ..." with the actual branch name when it's available in your data model.

1 / 1

Let's add the working branch/ref name to the merge request model and fill in that ellipsis.

1 / 1

First, let's add the `sourceBranch` property to the `MergeRequest` struct in the GitLab data model:

swift

Copy code

```
struct MergeRequest: Codable {
    // ...
    let sourceBranch: String
    // ...
}
```

Now, assuming you have the source branch/ref name available in your data, you can modify the `MergeRequestRow` view to display it:

swift

Copy code

```
VStack(alignment: .leading, spacing: 2) {
    Text(mergeRequest.title)
        .font(.headline)
    Text("Author: \(mergeRequest.author.username)")
        .font(.footnote)
        .foregroundColor(.gray)
    Text("Branch: \(mergeRequest.sourceBranch)")
        .font(.footnote)
        .foregroundColor(.gray)
}
```

This change will display the working branch/ref name for each merge request in the `MergeRequestRow`. Make sure to update your data parsing to include the source branch/ref name when fetching merge request data from GitLab.

1 / 1

Here's what i have for the two row views:

```
struct MergeRequestRow: View {
    let mergeRequest: MergeRequest

    var body: some View {
        Button(action: {
            NSWorkspace.shared.open(mergeRequest.webUrl)
        }) {
            HStack {
                if mergeRequest.draft {
                    Image(systemName: "pencil.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.orange)
                }
                VStack(alignment: .leading, spacing: 2) {
                    Text(mergeRequest.title)
                        .font(.headline)
                    HStack {
                        Text("Author: \(mergeRequest.author.username)")
                            .font(.footnote)
                            .foregroundColor(.gray)
                        Text("Branch: \(mergeRequest.sourceBranch)")
                            .font(.footnote)
                            .foregroundColor(.gray)
                    }
                }

                Spacer()

                if mergeRequest.approvalsLeft == 0 && !mergeRequest.shouldBeRebased {
                    Image(systemName: "checkmark.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.green)
                } else if mergeRequest.approvalsLeft == 0 &&
mergeRequest.shouldBeRebased {
                    Image(systemName: "arrow.clockwise.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.blue)
                } else {
                    Text("\(mergeRequest.approvalsLeft)")
                        .font(.headline)
                        .foregroundColor(.red)
                }

                HStack {
                    ForEach(mergeRequest.approvedBy.nodes, id: \.username) { approver in
                        if let avatarUrl = approver.avatarUrl {
                            AsyncImage(url: avatarUrl) { phase in
                                if let image = phase.image {
                                    image
                                        .resizable()
                                        .scaledToFit()
                                        .frame(width: 24, height: 24)
                                        .clipShape(Circle())
                                } else {
                                    Circle()
                                        .fill(Color.gray)
                                        .frame(width: 24, height: 24)
                                }
                            }
                        } else {
                            Circle()
                                .fill(Color.gray)
                                .frame(width: 24, height: 24)
                        }
                    }
                }
            }
            .padding(.vertical, 8)
        }
        .buttonStyle(PlainButtonStyle())
    }
}

struct MergeRequestRow: View {
    let mergeRequest: MergeRequest

    var body: some View {
        Button(action: {
            NSWorkspace.shared.open(mergeRequest.webUrl)
        }) {
            HStack {
                if mergeRequest.draft {
                    Image(systemName: "pencil.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.orange)
                }

                VStack(alignment: .leading, spacing: 2) {
                    Text(mergeRequest.title)
                        .font(.headline)
                    HStack {
                        Text("Author: \(mergeRequest.author.username)")
                            .font(.footnote)
                            .foregroundColor(.gray)
                        Text("Branch: \(mergeRequest.sourceBranch)")
                            .font(.footnote)
                            .foregroundColor(.gray)
                    }
                }

                Spacer()

                if mergeRequest.approvalsLeft == 0 && !mergeRequest.shouldBeRebased {
                    Image(systemName: "checkmark.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.green)
                } else if mergeRequest.approvalsLeft == 0 &&
mergeRequest.shouldBeRebased {
                    Image(systemName: "arrow.clockwise.circle")
                        .resizable()
                        .scaledToFit()
                        .frame(height: 24)
                        .foregroundColor(Color.blue)
                } else {
                    Text("\(mergeRequest.approvalsLeft)")
                        .font(.headline)
                        .foregroundColor(.red)
                }

                HStack {
                    ForEach(mergeRequest.approvedBy.nodes, id: \.username) { approver in
                        if let avatarUrl = approver.avatarUrl {
                            AsyncImage(url: avatarUrl) { phase in
                                if let image = phase.image {
                                    image
                                        .resizable()
                                        .scaledToFit()
                                        .frame(width: 24, height: 24)
                                        .clipShape(Circle())
                                } else {
                                    Circle()
                                        .fill(Color.gray)
                                        .frame(width: 24, height: 24)
                                }
                            }
                        } else {
                            Circle()
                                .fill(Color.gray)
                                .frame(width: 24, height: 24)
                        }
                    }
                }
```

```
            }
            .padding(.vertical, 8)
        }
        .buttonStyle(PlainButtonStyle())
    }
}
```

I want to show the head pipeline status on merge request rows too, though I don't need to show each stage status in that case. Let's keep it DRY.

1 / 1

It appears that you provided the same view code for both `MergeRequestRow` and `PipelineRow`. I assume you meant to provide the `PipelineRow` code as well. Nevertheless, I will show you how to display the head pipeline status on the `MergeRequestRow` and keep the code DRY.

First, let's create a new view, `PipelineStatusView`, that displays the overall pipeline status. This view can be used in both `MergeRequestRow` and `PipelineRow`:

swift

Copy code

```swift
struct PipelineStatusView: View {
    let pipeline: Pipeline

    var body: some View {
        VStack {
            if let status = pipeline.status {
                Image(systemName: statusIcon(status))
                    .resizable()
                    .scaledToFit()
                    .frame(height: 24)
                    .foregroundColor(statusColor(status))
            }
        }
    }

    func statusIcon(_ status: String) -> String {
        switch status {
        case "success":
            return "checkmark.circle"
        case "failed":
            return "xmark.circle"
        case "running":
            return "arrow.triangle.2.circlepath.circle"
        default:
            return "questionmark.circle"
        }
    }

    func statusColor(_ status: String) -> Color {
        switch status {
        case "success":
            return .green
        case "failed":
            return .red
        case "running":
            return .blue
        default:
            return .gray
        }
    }
}
```

Now, update the `MergeRequestRow` to display the head pipeline status using the `PipelineStatusView`:

swift

Copy code

```
HStack {
    // ...
    if let headPipeline = mergeRequest.headPipeline {
        PipelineStatusView(pipeline: headPipeline)
    }
    // ...
}
```

Finally, update the `PipelineRow` to use the `PipelineStatusView`:

swift

Copy code

```
HStack {
    PipelineStatusView(pipeline: pipeline)
    // ...
}
```

This way, you have a reusable `PipelineStatusView` that displays the overall pipeline status, and you can use it in both `MergeRequestRow` and `PipelineRow`.

1 / 1

Here is my status bar item class again, along with the two top level views.

```
class StatusBarItem: NSObject {

    private(set) var dataStore: GitLabDataStore

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem
    private var popover: NSPopover

    init(dataStore: GitLabDataStore) {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength: NSStatusItem.variableLength)
        popover = NSPopover()
        self.dataStore = dataStore

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(systemSymbolName: "target",
accessibilityDescription: nil)
            button.action = #selector(togglePopover(_:))
            button.target = self
        }

        // Set up the popover content
        popover.behavior = .transient
        popover.animates = true
        popover.contentViewController = NSHostingController(rootView:
StatusBarMenu().environmentObject(dataStore))
    }

    @objc func togglePopover(_ sender: AnyObject) {
        if popover.isShown {
            popover.performClose(sender)
        } else {
            if let button = statusBarItem.button {
                dataStore.refreshData()
                popover.show(relativeTo: button.bounds, of: button, preferredEdge: .minY)
            }
        }
    }

}

struct StatusBarMenu: View {

    @EnvironmentObject var store: GitLabDataStore

    @State private var selectedTab = 0

    var body: some View {
        VStack {
            TabView(selection: $selectedTab) {
                MergeRequestsListView()
                    .tabItem {
                        Text("Merge Requests")
                        Image(systemName: "list.bullet")
                    }
                    .tag(0)

                PipelinesListView()
                    .tabItem {
                        Text("Pipelines")
                        Image(systemName: "square.and.pencil")
                    }
                    .tag(1)
            }
        }
        .padding()
    }
}

struct ContentView: View {
    @EnvironmentObject var store: GitLabDataStore

    var body: some View {
        StatusBarMenu()
            .toolbar {
                Button("Refresh") {
                    store.refreshData()
                }
            }
    }
```

}

I've removed the .frame modifier from the statsu bar menu view to allow ContentView to fill the main window and resize along with it. I'd also like to let the popover resize. How can I do that?

1 / 1

To allow the popover to resize, you can create a custom `NSPopover` subclass that handles the resizing by updating its `contentSize`. Then, use this subclass in the `StatusBarItem` class.

First, create a `ResizablePopover` class that subclasses `NSPopover`:

swift

Copy code

```swift
class ResizablePopover: NSPopover {
    override func show(relativeTo positioningRect: NSRect, of
positioningView: NSView, preferredEdge: NSRectEdge) {
        if let contentViewController = contentViewController {
            let fittingSize = contentViewController.view.fittingSize
            contentSize = NSSize(width: max(fittingSize.width, 250), height:
max(fittingSize.height, 400))
        }
        super.show(relativeTo: positioningRect, of: positioningView,
preferredEdge: preferredEdge)
    }
}
```

Here, we override the `show` method to update the `contentSize` of the popover based on the fitting size of its content view. The `max` function is used to set minimum width and height values.

Next, update the `StatusBarItem` class to use `ResizablePopover` instead of `NSPopover`:

swift

Copy code

```swift
class StatusBarItem: NSObject {

    // ...
    private var popover: ResizablePopover // Change this line

    init(dataStore: GitLabDataStore) {
        // ...
        popover = ResizablePopover() // Change this line
        // ...
    }

    // ...
}
```

Now, the popover should resize based on the content size of the `StatusBarMenu`. The ContentView's main window will also be resizable as you removed the .frame modifier from the `StatusBarMenu` view.

1 / 1

Can the user resize it by dragging the corner?

1 / 1

No, the current implementation doesn't allow the user to resize the popover by dragging the corner. NSPopover does not come with built-in support for user resizing.

If you want to provide a user-resizable popover, you may need to consider using a custom NSWindow instead. Here's an example of how you could achieve this:

1. Create a custom NSWindowController:

swift

Copy code

```swift
class CustomWindowController: NSWindowController {

    convenience init(rootView: AnyView) {
        let window = NSWindow(
            contentRect: NSRect(x: 0, y: 0, width: 250, height: 400),
            styleMask: [.titled, .closable, .resizable, .miniaturizable],
            backing: .buffered,
            defer: false
        )
        window.contentView = NSHostingView(rootView: rootView)
        window.title = "Status Menu"
        self.init(window: window)
    }
}
```

2. Update the `StatusBarItem` class to use `CustomWindowController` instead of `NSPopover`:

swift

Copy code

```
class StatusBarItem: NSObject {

    private(set) var dataStore: GitLabDataStore

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem
    private var windowController: CustomWindowController // Change this line

    init(dataStore: GitLabDataStore) {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength:
NSStatusItem.variableLength)
        self.dataStore = dataStore

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(systemSymbolName: "target",
accessibilityDescription: nil)
            button.action = #selector(togglePopover(_:))
            button.target = self
        }

        // Set up the custom window
        let rootView = StatusBarMenu().environmentObject(dataStore)
        windowController = CustomWindowController(rootView:
AnyView(rootView)) // Change this line
    }

    @objc func togglePopover(_ sender: AnyObject) {
        if let window = windowController.window {
            if window.isVisible {
                windowController.close()
            } else {
                dataStore.refreshData()
                windowController.showWindow(nil)
                window.makeKeyAndOrderFront(nil)
                NSApp.activate(ignoringOtherApps: true)
            }
        }
    }
}
```

Now, the content will be displayed in a resizable window instead of a popover. The user can resize the window by dragging its corners, just like any other window. The window's initial size and minimum size are set in the `CustomWindowController` initializer.

1 / 1

I made a couple changes, but I like this.

```
class StatusBarItem: NSObject {

    private(set) var dataStore: GitLabDataStore

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem
    private var windowController: CustomWindowController

    init(dataStore: GitLabDataStore) {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength: NSStatusItem.variableLength)
        self.dataStore = dataStore

        // Set up the custom window
        let rootView = StatusBarMenu()
            .frame(width: 320, height: 700)
            .environmentObject(dataStore)
        windowController = CustomWindowController(rootView: AnyView(rootView))

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(systemSymbolName: "target", accessibilityDescription: nil)
            button.action = #selector(togglePopover(_:))
            button.target = self
        }
    }

    @objc func togglePopover(_ sender: AnyObject) {
        if let window = windowController.window {
            if window.isVisible && NSApplication.shared.isActive {
                windowController.close()
            } else {
                dataStore.refreshData()
                windowController.showWindow(nil)
                window.makeKeyAndOrderFront(nil)
                NSApp.activate(ignoringOtherApps: true)
            }
        }
    }
}
```

Here's my app code.

```
@main
struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        Window("GitLab Status", id: "main") {
            ContentView()
                .environmentObject(GitLabDataStore.shared)
        }
    }
}

class AppDelegate: NSObject, NSApplicationDelegate {
    var statusBarItem: StatusBarItem?

    func applicationDidFinishLaunching(_ notification: Notification) {
        statusBarItem = StatusBarItem(dataStore: GitLabDataStore.shared)
        GitLabDataStore.shared.refreshData()
    }
}

private extension GitLabDataStore {
    static let shared = GitLabDataStore()
}
```

At this point I have a main window as well as a window presented from the status bar item, but they're basically the same. I'd like to remove that duplication.

1 / 1

You can remove the duplication by removing the main window from your `GitLabStatusMenuBarItemApp` and keeping only the status bar item window. Since the content of both windows is the same, having just the status bar item window should be sufficient. Here's how you can do that:

1. Remove the `Window` from the `GitLabStatusMenuBarItemApp` body:

swift

Copy code

```swift
@main
struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        // No window here, keep it empty
    }
}
```

2. Remove the `ContentView` since it's no longer needed:

swift

Copy code

```swift
// Remove the ContentView
```

Now, your app will have only one window: the one presented from the status bar item. The content will be the same as before, and there will be no duplication.

1 / 1

I can't leave the body empty. Property declares an opaque return type, but has no initializer expression from which to infer an underlying type

1 / 1

Apologies for the confusion. In this case, you can use an `EmptyView` for the body of the

`GitLabStatusMenuBarItemApp`.

Here's the updated code:

swift

Copy code

```swift
@main
struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            EmptyView()
        }
    }
}
```

This way, your app will not have a main window, and the only window will be the one presented from the status bar item.

1 / 1

That creates an empty view in a window, so I still have two windows. Let's go back and remove the status item window intsead of the main window, and use the status item to show and hide the main window.

1 / 1

I understand. In this case, you can modify the `StatusBarItem` class to control the main window's visibility. You can use `NSApplication.shared.mainWindow` to get a reference to the main window.

Here's the updated `StatusBarItem` class:

swift

Copy code

```
class StatusBarItem: NSObject {

    private(set) var dataStore: GitLabDataStore

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem

    init(dataStore: GitLabDataStore) {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength:
NSStatusItem.variableLength)
        self.dataStore = dataStore

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(systemSymbolName: "target",
accessibilityDescription: nil)
            button.action = #selector(toggleMainWindow(_:))
            button.target = self
        }
    }

    @objc func toggleMainWindow(_ sender: AnyObject) {
        if let mainWindow = NSApplication.shared.mainWindow {
            if mainWindow.isVisible && NSApplication.shared.isActive {
                mainWindow.close()
            } else {
                dataStore.refreshData()
                mainWindow.makeKeyAndOrderFront(nil)
                NSApp.activate(ignoringOtherApps: true)
            }
        }
    }
}
```

And here's the updated `GitLabStatusMenuBarItemApp`:

swift

Copy code

```
@main
struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(GitLabDataStore.shared)
        }
    }
}
```

With these changes, your app will have a single main window, and the status item will control its visibility.

1 / 1

Here's what I did:

```
class StatusBarItem: NSObject {

    private(set) var dataStore: GitLabDataStore

    private var statusBar: NSStatusBar
    private var statusBarItem: NSStatusItem

    init(dataStore: GitLabDataStore) {
        statusBar = NSStatusBar.system
        statusBarItem = statusBar.statusItem(withLength: NSStatusItem.variableLength)
        self.dataStore = dataStore

        super.init()

        // Set up the Menu Bar Item
        if let button = statusBarItem.button {
            button.image = NSImage(systemSymbolName: "target",
accessibilityDescription: nil)
            button.action = #selector(hideOrActivate(_:))
            button.target = self
        }
    }

    @objc func hideOrActivate(_ sender: AnyObject) {
        if NSApplication.shared.isActive {
            NSApplication.shared.hide(self)
        } else {
            dataStore.refreshData()
            NSApp.activate(ignoringOtherApps: true)
        }
    }
}

struct GitLabStatusMenuBarItemApp: App {
    @NSApplicationDelegateAdaptor(AppDelegate.self) var appDelegate

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(GitLabDataStore.shared)
        }
    }
}
```

Now that the window is defined entirely in SwiftUI, can we do the same with the menu bar item?

1 / 1

Yes, you can create the status bar item using SwiftUI by using the `NSStatusBar` in a SwiftUI `App`. This is possible by defining a `Settings` scene in the `App` and managing the status bar item there.

Here's an example of how you can achieve this:

1. Create a new SwiftUI View called `StatusBarItemView`:

swift

Copy code

```swift
import SwiftUI

struct StatusBarItemView: View {
    @EnvironmentObject var dataStore: GitLabDataStore

    var body: some View {
        Button(action: {
            if NSApplication.shared.isActive {
                NSApplication.shared.hide(self)
            } else {
                dataStore.refreshData()
                NSApp.activate(ignoringOtherApps: true)
            }
        }) {
            Image(systemName: "target")
        }
    }
}
```

2. Update your `GitLabStatusMenuBarItemApp` to include a `Settings` scene:

swift

Copy code

```
@main
struct GitLabStatusMenuBarItemApp: App {
    @StateObject private var dataStore = GitLabDataStore()

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(dataStore)
        }
        .settings {
            StatusBarItemView()
                .frame(width: 24, height: 24)
                .environmentObject(dataStore)
        }
    }
}
```

Please note that the `Settings` scene is only available on macOS 12 Monterey and later. If you need to support older versions of macOS, you should stick to the previous approach using `NSStatusBar` and `NSStatusItem` in the `AppDelegate`.

1 / 1

There's no .settings modifier. What else can I try?

1 / 1

Apologies for the confusion. It seems there is no built-in SwiftUI support for creating status bar items in macOS apps at the moment.

However, you can wrap the status bar item initialization inside a SwiftUI view that will be displayed in the main window. You can create a SwiftUI `View` that holds the `StatusBarItem`, and use it in your `ContentView`. This way, you won't need the `AppDelegate`, and the status bar item will be created alongside your main window.

Here's an example:

1. Create a new SwiftUI `View` called `StatusBarItemHandler`:

swift

Copy code

```swift
import SwiftUI

struct StatusBarItemHandler: NSViewRepresentable {
    @EnvironmentObject var dataStore: GitLabDataStore

    class Coordinator {
        var statusBarItem: StatusBarItem?
    }

    func makeCoordinator() -> Coordinator {
        Coordinator()
    }

    func makeNSView(context: Context) -> NSView {
        let view = NSView()

        DispatchQueue.main.async {
            context.coordinator.statusBarItem = StatusBarItem(dataStore: dataStore)
        }

        return view
    }

    func updateNSView(_ nsView: NSView, context: Context) {
        // No updates needed.
    }
}
```

2. Add the `StatusBarItemHandler` to your `ContentView`:

swift

Copy code

```swift
struct ContentView: View {
    @EnvironmentObject var store: GitLabDataStore
    var body: some View {
        VStack {
            StatusBarMenu()
                .toolbar {
                    Button("Refresh") {
                        store.refreshData()
                    }
                }
            StatusBarItemHandler()
                .frame(width: 0, height: 0)
                .hidden()
        }
    }
}
```

3. Remove the `AppDelegate` and related code from your `GitLabStatusMenuBarItemApp`:

swift

Copy code

```
@main
struct GitLabStatusMenuBarItemApp: App {
    @StateObject private var dataStore = GitLabDataStore()

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(dataStore)
        }
    }
}
```

This way, you're initializing the `StatusBarItem` within the SwiftUI view hierarchy. The `StatusBarItemHandler` creates a zero-sized, hidden `NSView` that initializes the `StatusBarItem` when it's displayed in the main window.

1 / 1

I figured it out.

What's a one-liner to quit the application gracefully? I would run this from a menu item action in SwiftUI.

1 / 1

You can use the following one-liner to quit the application gracefully from a SwiftUI menu item action:

swift

Copy code

```swift
NSApp.terminate(nil)
```

Here's an example of how to use it in a SwiftUI `Button` action:

swift

Copy code

```swift
Button("Quit") {
    NSApp.terminate(nil)
}
```

1 / 1

Now my code looks like this:

```
@main
struct GitLabStatusMenuBarItemApp: App {

    let store = GitLabDataStore()

    var body: some Scene {
        MenuBarExtra {
            StatusBarMenu()
                .environmentObject(store)
        } label: {
            Image(systemName: "target")
        }
        .menuBarExtraStyle(.window)
    }

}

struct StatusBarMenu: View {

    enum Tab: Hashable {
        case mergeRequests, pipelines
    }

    @EnvironmentObject var store: GitLabDataStore

    @State private var selectedTab: Tab = .mergeRequests

    var body: some View {
        VStack {
            HStack {
                Picker(selection: $selectedTab) {
                    Text("Merge Requests")
                        .tag(Tab.mergeRequests)

                    Text("Pipelines")
                        .tag(Tab.pipelines)
                } label: {}
                .pickerStyle(.segmented)

                Menu {
                    Button("Quit") {
                        NSApp.terminate(nil)
                    }
                } label: {
                    Image(systemName: "arrow.clockwise")
                } primaryAction: {
                    store.refreshData()
                }
                .fixedSize()
            }
            .padding([.top, .horizontal])

            Group {
                switch selectedTab {
                case .mergeRequests:
                    MergeRequestsListView()
                case .pipelines:
                    PipelinesListView()
                }
            }
            .listStyle(.inset(alternatesRowBackgrounds: true))
        }
        .frame(width: 320, height: 700)
        .background(.background)
        .onAppear {
            store.refreshData()
        }
    }
}
```

In the label of the menu whose primary action is to refresh, I'd like to show the arrow icon only when no refresh is in progress. When a refresh is happening, I want to show a progress indicator in that place.

1 / 1

You can achieve this by using a conditional view based on the `isLoading` property of your data store. You'll need to create an `isLoading` property in your `GitLabDataStore` and update it accordingly during the refresh process. Then, you can conditionally display the arrow icon or a progress indicator in the menu label.

Here's an example of how you can modify the `StatusBarMenu` view:

swift

Copy code

```swift
struct StatusBarMenu: View {
    // ...

    @EnvironmentObject var store: GitLabDataStore

    // ...

    var body: some View {
        VStack {
            HStack {
                // ...

                Menu {
                    Button("Quit") {
                        NSApp.terminate(nil)
                    }
                } label: {
                    if store.isLoading {
                        ProgressView()
                            .frame(width: 16, height: 16)
                            .progressViewStyle(CircularProgressViewStyle(tint: .blue))
                    } else {
                        Image(systemName: "arrow.clockwise")
                    }
                } primaryAction: {
                    store.refreshData()
                }
                .fixedSize()
            }
            .padding([.top, .horizontal])

            // ...
        }
        .frame(width: 320, height: 700)
        .background(.background)
        .onAppear {
            store.refreshData()
        }
    }
}
```

This will display a progress indicator when `store.isLoading` is true, and the arrow icon otherwise. Make sure to update the `isLoading` property in your data store during the refresh process to reflect the loading state correctly.

1 / 1

Here are two view types that are almost the same.

```
struct MergeRequestsListView: View {
    var projects: [GitLabProject]?

    var body: some View {
        List {
            if let projects {
                ForEach(projects) { project in
```

```
                    Section(header: Text(project.fullPath)) {
                        ForEach(project.mergeRequests.nodes) { mergeRequest in
                            MergeRequestRow(mergeRequest: mergeRequest)
                        }
                    }
                }
            } else {
                Text("Not loaded")
            }
        }
    }
}

struct PipelinesListView: View {
    var projects: [GitLabProject]?

    var body: some View {
        List {
            if let projects {
                ForEach(projects) { project in
                    Section(header: Text(project.fullPath)) {
                        ForEach(project.pipelines.nodes) { pipeline in
                            PipelineRow(projectURL: project.webUrl, pipeline: pipeline)
                        }
                    }
                }
            } else {
                Text("Not loaded")
            }
        }
    }
}
```

Please DRY these into one view. Use generics and key paths.

1 / 1

You can create a generic `ListView` that accepts a list of items and a key path for the nested items. Here's an example:

swift

Copy code

```swift
struct ListView<Item: Identifiable, NestedItem: Identifiable, Content: View>: View {
    var projects: [Item]?
    var nestedItemsKeyPath: KeyPath<Item, [NestedItem]>
    var content: (NestedItem) -> Content

    init(projects: [Item]?, nestedItemsKeyPath: KeyPath<Item, [NestedItem]>,
@ViewBuilder content: @escaping (NestedItem) -> Content) {
        self.projects = projects
        self.nestedItemsKeyPath = nestedItemsKeyPath
        self.content = content
    }

    var body: some View {
        List {
            if let projects = projects {
                ForEach(projects) { project in
                    Section(header: Text(project.fullPath)) {
                        ForEach(project[keyPath: nestedItemsKeyPath])
{ nestedItem in
                            content(nestedItem)
                        }
                    }
                }
            } else {
                Text("Not loaded")
            }
        }
    }
}
```

Now you can use this `ListView` to create both `MergeRequestsListView` and `PipelinesListView`:

swift

Copy code

```
struct MergeRequestsListView: View {
    var projects: [GitLabProject]?

    var body: some View {
        ListView(projects: projects, nestedItemsKeyPath:
\.mergeRequests.nodes) { mergeRequest in
            MergeRequestRow(mergeRequest: mergeRequest)
        }
    }
}

struct PipelinesListView: View {
    var projects: [GitLabProject]?
    var body: some View {
        ListView(projects: projects, nestedItemsKeyPath: \.pipelines.nodes) { pipeline in
            PipelineRow(projectURL: project.webUrl, pipeline: pipeline)
        }
    }
}
```

This solution utilizes generics and key paths to create a reusable `ListView` that can handle different types of nested items. The `content` closure allows for custom row content depending on the type of the nested item.

1 / 1

I went with this:

```
struct ProjectItemList<NestedItem: Identifiable, Content: View>: View {
    var projects: [GitLabProject]?
    var nestedItemsKeyPath: KeyPath<GitLabProject, [NestedItem]>
    var content: (GitLabProject, NestedItem) -> Content

    init(projects: [GitLabProject]?, nestedItemsKeyPath: KeyPath<GitLabProject, [NestedItem]>, @ViewBuilder content: @escaping (GitLabProject, NestedItem) -> Content) {
        self.projects = projects
        self.nestedItemsKeyPath = nestedItemsKeyPath
        self.content = content
    }

    var body: some View {
        List {
            if let projects {
                ForEach(projects) { project in
                    Section(header: Text(project.fullPath)) {
                        ForEach(project[keyPath: nestedItemsKeyPath]) { nestedItem in
                            content(project, nestedItem)
                        }
                    }
                }
            } else {
                Text("Not loaded")
            }
        }
    }
}

struct ProjectsView: View {

    enum Tab: Hashable {
        case mergeRequests, pipelines
    }

    @EnvironmentObject var store: ProjectStore

    @State private var selectedTab: Tab = .mergeRequests

    var body: some View {
        VStack(alignment: .leading) {
            HStack {
                Picker(selection: $selectedTab) {
                    Text("Merge Requests")
                        .tag(Tab.mergeRequests)

                    Text("Pipelines")
                        .tag(Tab.pipelines)
                } label: {}
                .pickerStyle(.segmented)
                .fixedSize()

                Spacer()

                Menu {
                    Button("Quit") {
                        NSApp.terminate(nil)
                    }
                } label: {
                    if store.isLoading {
                        Image(systemName: "ellipsis")
                    } else {
                        Image(systemName: "arrow.clockwise")
                    }
                } primaryAction: {
                    store.refreshData()
                }
                .fixedSize()
            }
            .padding([.top, .horizontal])

            if let message = store.errorMessage {
                Text(message)
                    .padding(.horizontal)
            }

            Group {
                switch selectedTab {
                case .mergeRequests:
                    ProjectItemList(
                        projects: store.projects,
                        nestedItemsKeyPath: \.mergeRequests.nodes
                    ) { _, mergeRequest in
                        MergeRequestRow(mergeRequest: mergeRequest)
                    }
                case .pipelines:
                    ProjectItemList(
                        projects: store.projects,
                        nestedItemsKeyPath: \.pipelines.nodes
                    ) { project, pipeline in
                        PipelineRow(projectURL: project.webUrl, pipeline: pipeline)
                    }
                }
            }
            .listStyle(.inset(alternatesRowBackgrounds: true))
        }
        .frame(width: 320, height: 700)
        .background(.background)
        .onAppear {
            store.refreshData()
        }
    }
}
```

Please factor ProjectView's top HStack into a new view called ProjectsToolbar. Include a preview provider.

Then, when the user selects a new item from the toolbar's menu, present a modal dialog for collecting the user's gitlab personal access token. It should use a secure text field and, if the user confirms, save it to user defaults. The store should observe the user default and reload when it changes.

First, let's create the `ProjectsToolbar` view:

swift

Copy code

```
struct ProjectsToolbar: View {
    @Binding var selectedTab: ProjectsView.Tab
    @EnvironmentObject var store: ProjectStore
    @State private var showingTokenSheet = false

    var body: some View {
        HStack {
            Picker(selection: $selectedTab) {
                Text("Merge Requests")
                    .tag(ProjectsView.Tab.mergeRequests)

                Text("Pipelines")
                    .tag(ProjectsView.Tab.pipelines)
            } label: {}
            .pickerStyle(.segmented)
            .fixedSize()

            Spacer()

            Menu {
                Button("Set GitLab Personal Access Token") {
                    showingTokenSheet = true
                }
                Button("Quit") {
                    NSApp.terminate(nil)
                }
            } label: {
                if store.isLoading {
                    Image(systemName: "ellipsis")
                } else {
                    Image(systemName: "arrow.clockwise")
                }
            } primaryAction: {
                store.refreshData()
            }
            .fixedSize()
        }
        .sheet(isPresented: $showingTokenSheet) {
            TokenSheet()
        }
    }
}

struct ProjectsToolbar_Previews: PreviewProvider {
    static var previews: some View {
        ProjectsToolbar(selectedTab: .constant(.mergeRequests))
            .environmentObject(ProjectStore())
    }
}
```

Now, let's create the `TokenSheet` view for collecting the user's GitLab personal access token:

swift

Copy code

```
struct TokenSheet: View {
    @Environment(\.dismiss) var dismiss
    @AppStorage("gitlabToken") private var gitlabToken: String = ""
    @State private var tokenInput: String = ""

    var body: some View {
        VStack {
            Text("Enter your GitLab Personal Access Token")
                .font(.headline)
            SecureField("Personal Access Token", text: $tokenInput)
            HStack {
                Button("Cancel") {
                    dismiss()
                }
                Spacer()
                Button("Save") {
                    gitlabToken = tokenInput
                    dismiss()
                }
            }
        }
        .padding()
    }
}
```

Now, update the `ProjectsView` to use the new `ProjectsToolbar`:

swift

Copy code

```
struct ProjectsView: View {
    // ...
    var body: some View {
        VStack(alignment: .leading) {
            ProjectsToolbar(selectedTab: $selectedTab)
                .environmentObject(store)
                .padding([.top, .horizontal])

            // ...
        }
        // ...
    }
}
```

Finally, let's add an `objectWillChange` publisher to the `ProjectStore` to observe changes to the user default and reload when it changes:

swift

Copy code

```
import Combine

class ProjectStore: ObservableObject {
    @Published var projects: [GitLabProject]?
    @Published var isLoading: Bool = false
    @Published var errorMessage: String?

    private var cancellables: Set<AnyCancellable> = []
    private let gitlabAPI: GitLabAPI

    init() {
        gitlabAPI = GitLabAPI()

        // Listen to gitlabToken changes
        UserDefaults.standard.publisher(for: \.gitlabToken)
            .sink { [weak self] _ in
                self?.refreshData()
            }
            .store(in: &cancellables)

        // ...
    }

    // ...
}
```

Now, when the user selects a new item from the toolbar's menu, it will present a modal dialog for collecting the user's GitLab personal access token. The token will be saved to user defaults, and the store will observe the user default and reload when it changes.

1 / 1

Close… but "Value of type 'UserDefaults' has no member 'gitlabToken'"

1 / 1

Apologies for the confusion. The issue is that the `UserDefaults` publisher is only available for key paths of the default `UserDefaults` instance. We can work around this by defining a custom key path for the `gitlabToken` and using it in the `publisher(for:)` method.

First, create an extension for `UserDefaults` to define the custom key path:

swift

Copy code

```swift
extension UserDefaults {
    @objc dynamic var gitlabToken: String {
        get { string(forKey: "gitlabToken") ?? "" }
        set { set(newValue, forKey: "gitlabToken") }
    }
}
```

Now, update the `ProjectStore` class to use this custom key path:

swift

Copy code

```swift
import Combine

class ProjectStore: ObservableObject {
    // ...

    init() {
        gitlabAPI = GitLabAPI()

        // Listen to gitlabToken changes
        UserDefaults.standard.publisher(for: \UserDefaults.gitlabToken)
            .sink { [weak self] _ in
```

```
                self?.refreshData()
            }
            .store(in: &cancellables)

        // ...
    }

    // ...
}
```

This should resolve the issue, and the store will now observe changes to the `gitlabToken` in user defaults and refresh data when it changes.

1 / 1

Add a preview provider for TokenSheet.

1 / 1

You can add a preview provider for `TokenSheet` like this:

swift

Copy code

```swift
struct TokenSheet_Previews: PreviewProvider {
    static var previews: some View {
        TokenSheet(isPresented: .constant(true), onConfirm: { _ in })
    }
}
```

This creates a preview of `TokenSheet` with the `isPresented` binding set to `true` and an empty closure for `onConfirm`. This allows you to see how the `TokenSheet` looks like in the preview canvas.

1 / 1

Here is the code I ended up with:

```
struct TokenSheet: View {
    @Environment(\.dismiss) var dismiss
    @AppStorage("gitLabToken") private var gitLabToken: String = ""
    @State private var tokenInput: String = ""

    var body: some View {
        VStack(alignment: .leading) {
            Text("Enter your GitLab Personal Access Token")
                .font(.headline)
            Text("First, [create a token](https://gitlab.com/-/profile/personal_access_tokens) with `read_api` rights.")
                .font(.subheadline)

            SecureField("", text: $tokenInput)

            HStack {
                Button("Cancel") {
                    dismiss()
                }
                Spacer()
                Button("Save") {
                    gitLabToken = tokenInput
                    dismiss()
                }
            }
        }
        .padding()
        .frame(width: 300)
    }
}

struct TokenSheet_Previews: PreviewProvider {
    static var previews: some View {
        TokenSheet()
    }
}

class ProjectStore: ObservableObject {
```

```
@Published var projects: [GitLabProject]?
@Published var isLoading: Bool = false
@Published var errorMessage: String?

private var cancellables: Set<AnyCancellable> = []

init() {
    UserDefaults.standard.publisher(for: \.gitLabToken)
        .sink { [weak self] _ in
            guard let self else { return }
            self.errorMessage = nil
            self.refreshData()
        }
        .store(in: &cancellables)
}

func refreshData() {
    guard !isLoading else {
        return
    }

    let token = UserDefaults.standard.gitLabToken
    guard !token.isEmpty else {
        errorMessage = "No GitLab token. Set it in the corner menu."
        return
    }

    isLoading = true

    ProjectQuery().fetchData(token: token) { [weak self] result in
        DispatchQueue.main.async {
            guard let self else { return }

            self.isLoading = false

            switch result {
            case .success(let data):
                self.projects = data
                self.errorMessage = nil
            case .failure(let error):
                self.errorMessage = "Error fetching GitLab data: \(error)"
            }
        }
    }
}
```

```
}

class ProjectQuery {

    private let gitLabApiURL = "https://gitlab.com/api/graphql"

    func fetchData(token: String, completion: @escaping (Result<[GitLabProject], Error>) -> Void) {
        var request = URLRequest(url: URL(string: gitLabApiURL)!)
        request.httpMethod = "POST"
        request.addValue("Bearer \(token)", forHTTPHeaderField: "Authorization")
        request.addValue("application/json", forHTTPHeaderField: "Content-Type")


        let pipelineFields = """
        id
        path
        createdAt
        ref
        queuedDuration
        duration
        status
        stages {
          nodes {
            id
            name
            status
          }
        }
        testReportSummary {
          total {
            failed
            count
            time
          }
        }
        """

        let query = """
        {
          projects(ids: ["gid://gitlab/Project/3472737", "gid://gitlab/Project/250833", "gid://gitlab/Project/10582521"]) {
            nodes {
              id
              fullPath
              webUrl
```

```
      pipelines(first: 5, ref: "master") {
        pageInfo {
          endCursor
        }
        nodes {
          \(pipelineFields)
        }
      }
      mergeRequests(state: opened, first: 10) {
        pageInfo {
          endCursor
        }
        nodes {
          id
          webUrl
          draft
          title
          createdAt
          author {
            username
          }
          sourceBranch
          headPipeline {
            \(pipelineFields)
          }
          approvalsLeft
          approvedBy {
            nodes {
              username
              avatarUrl
            }
          }
          shouldBeRebased
        }
      }
    }
  }
}
"""

let requestBody = [
    "query": query
]

do {
    request.httpBody = try JSONSerialization.data(withJSONObject: requestBody,
options: [])
        } catch {
            completion(.failure(error))
            return
        }

        let task = URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                completion(.failure(error))
                return
            }

            guard let data = data else {
                completion(.failure(NSError(domain: "", code: -1, userInfo:
[NSLocalizedDescriptionKey: "No data received"])))
                return
            }

            do {
                // let json = String(data: data, encoding: .utf8)
                // print(json!)

                let response = try JSONDecoder().decode(GitLabProjectsResponse.self,
from: data)
                completion(.success(response.data.projects.nodes))
            } catch {
                completion(.failure(error))
            }
        }

        task.resume()
    }

}
```

Now then, previously we discussed discovering a gitlab project's ID given its full path, like this:

```
query {
  subscription: project(fullPath: "inkscape/inkscape") {
    id
    fullPath
  }
}
```

Let's now add a new menu item, that presents a sheet in which we show a user default-backed list of project subscriptions. A project subscription will consist of a project full path and project ID. On this sheet, the user can delete subscriptions from the list, reorder the subscriptions, and add a new subscription. To add a new subscription, they type a project's full path into a search field and either click a search button or type return. The search operation looks up a project ID and full path given the full path, as just discussed. If it's found, that's a new subscription, so we add it to the list. All changes to the list save to user defaults immediately. User defaults changes reload data in the project store. Finally, the project store supplies the list of subscriptions' IDs from user defaults to the project query's fetchData method, which then incorporates them into the project query.

1 / 1

First, let's create a `Subscription` struct and a `SubscriptionList` class that will handle subscriptions and manage the user defaults.

swift

Copy code

```swift
struct Subscription: Identifiable, Codable, Hashable {
    let id: String
    let fullPath: String
}

class SubscriptionList: ObservableObject {
    @Published var subscriptions: [Subscription] {
        didSet {
            saveToUserDefaults()
        }
    }

    private let userDefaultsKey = "subscriptions"

    init() {
        subscriptions = Self.loadFromUserDefaults(forKey: userDefaultsKey) ?? []
    }

    private func saveToUserDefaults() {
        Self.saveToUserDefaults(subscriptions, forKey: userDefaultsKey)
    }

    private static func saveToUserDefaults(_ subscriptions: [Subscription], forKey key: String) {
        if let data = try? JSONEncoder().encode(subscriptions) {
            UserDefaults.standard.set(data, forKey: key)
        }
    }

    private static func loadFromUserDefaults(forKey key: String) ->
[Subscription]? {
        guard let data = UserDefaults.standard.data(forKey: key) else
{ return nil }
        return try? JSONDecoder().decode([Subscription].self, from: data)
    }
}
```

Then, update the `ProjectStore` class to observe changes in the subscriptions and adjust the `fetchData` method to use the new subscription list:

swift

Copy code

```
class ProjectStore: ObservableObject {
    // ...
    private var subscriptionsCancellable: AnyCancellable?

    init() {
        subscriptionsCancellable = SubscriptionList.shared.$subscriptions
            .sink { [weak self] _ in
                self?.refreshData()
            }

        // ...
    }

    func refreshData() {
        // ...
        let subscriptionIDs = SubscriptionList.shared.subscriptions.map(\.id)
        // ...
        ProjectQuery().fetchData(token: token, projectIDs: subscriptionIDs,
completion: /*...*/)
    }

    // ...
}

class ProjectQuery {
    // ...
    func fetchData(token: String, projectIDs: [String], completion: @escaping
(Result<[GitLabProject], Error>) -> Void) {
        // ...

        let query = """
        {
          projects(ids: \(projectIDs)) {
            nodes {
              // ...
            }
          }
        }
        """

        // ...
    }

    // ...
}
```

Next, create a `SubscriptionsView` to allow the user to manage subscriptions:

swift

Copy code

```
struct SubscriptionsView: View {
    @Environment(\.dismiss) var dismiss
    @EnvironmentObject var subscriptionList: SubscriptionList
    @State private var newSubscriptionPath: String = ""
    @State private var subscriptionSearchResult: Subscription?

    var body: some View {
        VStack {
            HStack {
                TextField("Enter a project's full path", text:
$newSubscriptionPath)
                    .textFieldStyle(RoundedBorderTextFieldStyle())

                Button("Search") {
                    searchProjectID(for: newSubscriptionPath)
                }
            }

            if let subscription = subscriptionSearchResult {
                Button("Add \(subscription.fullPath)") {
                    subscriptionList.subscriptions.append(subscription)
                    newSubscriptionPath = ""
                    subscriptionSearchResult = nil
                }
            }

            List {
                ForEach(subscriptionList.subscriptions) { subscription in
```

```
                    Text(subscription.fullPath)
                }
                .onDelete { indexSet in
                    subscriptionList.subscriptions.remove(atOffsets:
indexSet)
                }
            }
            .listStyle(InsetListStyle())

            HStack {
                Button("Done") {
                    dismiss()
                }
            }
        }
        .padding()
    }

    func searchProjectID(for fullPath: String) {
        // Add your logic to search the project ID given the
```

1 / 1

Continue

1 / 1

full path using the GraphQL API. This will update the `subscriptionSearchResult` state when a result is found.

swift

Copy code

```
    func searchProjectID(for fullPath: String) {
        let token = UserDefaults.standard.gitLabToken
        guard !token.isEmpty else { return }

        let query = """
        {
          project(fullPath: "\(fullPath)") {
            id
            fullPath
          }
        }
        """

        // Perform the query and handle the response
        // If a project is found, update the subscriptionSearchResult state
        performGraphQLQuery(token: token, query: query) { (result: Result<ProjectSearchResult, Error>) in
            switch result {
            case .success(let projectSearchResult):
                if let project = projectSearchResult.data.project {
                    DispatchQueue.main.async {
                        self.subscriptionSearchResult = Subscription(id: project.id, fullPath: project.fullPath)
                    }
                } else {
                    DispatchQueue.main.async {
                        self.subscriptionSearchResult = nil
                    }
                }
            case .failure(let error):
                print("Error searching for project: \(error)")
                DispatchQueue.main.async {
                    self.subscriptionSearchResult = nil
                }
            }
        }
    }
}
```

Now, add a new menu item in the `ProjectsToolbar` that presents the `SubscriptionsView` when clicked:

swift

Copy code

```
struct ProjectsToolbar: View {
    @EnvironmentObject var store: ProjectStore
    @State private var showTokenSheet = false
    @State private var showSubscriptionsView = false

    var body: some View {
        HStack {
            // ...
            Menu {
                Button("Subscriptions") {
                    showSubscriptionsView.toggle()
                }
                .sheet(isPresented: $showSubscriptionsView) {
                    SubscriptionsView()
                        .environmentObject(SubscriptionList.shared)
                }

                // ...
            }
            .fixedSize()
        }
    }
}
```

This will add the new menu item for managing subscriptions, allowing users to add, reorder, and delete subscriptions. The `ProjectStore` will observe changes in the subscriptions and refresh data accordingly.

Please note that you might need to adjust the code according to your project structure and import statements if needed. Additionally, you may need to implement the `performGraphQLQuery` function to execute a GraphQL query with the given token and handle the JSON response.

1 / 1

We already have a definition of how to make a GraphQL query in ProjectQuery, so let's follow the same pattern to do the subscription query.

1 / 1

You can create a new method within the `ProjectQuery` class to fetch the project details for a given full path using the GraphQL API. Here's the updated `ProjectQuery` class with the new method:

swift

Copy code

```
class ProjectQuery {
    // ... Existing code ...

    func fetchProjectDetails(token: String, fullPath: String, completion: @escaping (Result<GitLabProject?, Error>) -> Void) {
        var request = URLRequest(url: URL(string: gitLabApiURL)!)
        request.httpMethod = "POST"
        request.addValue("Bearer \(token)", forHTTPHeaderField: "Authorization")
        request.addValue("application/json", forHTTPHeaderField: "Content-Type")

        let query = """
        {
          project(fullPath: "\(fullPath)") {
            id
            fullPath
          }
        }
        """

        let requestBody = [
            "query": query
        ]

        do {
            request.httpBody = try JSONSerialization.data(withJSONObject: requestBody, options: [])
        } catch {
            completion(.failure(error))
            return
        }

        let task = URLSession.shared.dataTask(with: request) { data, response, error in
            if let error = error {
                completion(.failure(error))
                return
            }

            guard let data = data else {
                completion(.failure(NSError(domain: "", code: -1, userInfo: [NSLocalizedDescriptionKey: "No data received"])))
                return
            }
            do {
                let response = try
JSONDecoder().decode(GitLabProjectResponse.self, from: data)
                completion(.success(response.data.project))
            } catch {
                completion(.failure(error))
            }
        }

        task.resume()
    }
}
```

Now, you can use this method in the `SubscriptionsView` to search for a project by its full path:

swift

Copy code

```
func searchProjectID(for fullPath: String) {
    let token = UserDefaults.standard.gitLabToken
    guard !token.isEmpty else { return }

    ProjectQuery().fetchProjectDetails(token: token, fullPath: fullPath)
{ [weak self] result in
        guard let self = self else { return }

        switch result {
        case .success(let project):
            DispatchQueue.main.async {
                self.subscriptionSearchResult = project.map
{ Subscription(id: $0.id, fullPath: $0.fullPath) }
            }
        case .failure(let error):
            print("Error searching for project: \(error)")
            DispatchQueue.main.async {
                self.subscriptionSearchResult = nil
            }
        }
    }
}
```

With this update, you can now use the existing `ProjectQuery` class to fetch project details and reuse the pattern for making GraphQL queries.

1 / 1

Move searchProjectID into ProjectStore, please.

1 / 1

Here's the updated `ProjectStore` class with the `searchProjectID` method:

swift

Copy code

```swift
class ProjectStore: ObservableObject {
    // ... Existing code ...

    func searchProjectID(for fullPath: String, completion: @escaping (Result<Subscription?, Error>) -> Void) {
        let token = UserDefaults.standard.gitLabToken
        guard !token.isEmpty else { return }

        ProjectQuery().fetchProjectDetails(token: token, fullPath: fullPath) { result in
            switch result {
            case .success(let project):
                DispatchQueue.main.async {
                    completion(.success(project.map { Subscription(id: $0.id, fullPath: $0.fullPath) }))
                }
            case .failure(let error):
                print("Error searching for project: \(error)")
                DispatchQueue.main.async {
                    completion(.failure(error))
                }
            }
        }
    }
}
```

Now you can call this method from the `SubscriptionsView`:

swift

Copy code

```swift
func searchProject() {
    store.searchProjectID(for: projectSearchText) { [weak self] result in
        guard let self = self else { return }

        switch result {
        case .success(let subscription):
            DispatchQueue.main.async {
                self.subscriptionSearchResult = subscription
            }
        case .failure(let error):
            print("Error searching for project: \(error)")
            DispatchQueue.main.async {
                self.subscriptionSearchResult = nil
            }
        }
    }
}
```

Don't forget to update the `SubscriptionsView` by adding an `@EnvironmentObject var store: ProjectStore` property at the beginning of the view.

With this change, the `searchProjectID` function is now part of the `ProjectStore`, making it easier to manage and maintain the code.

1 / 1

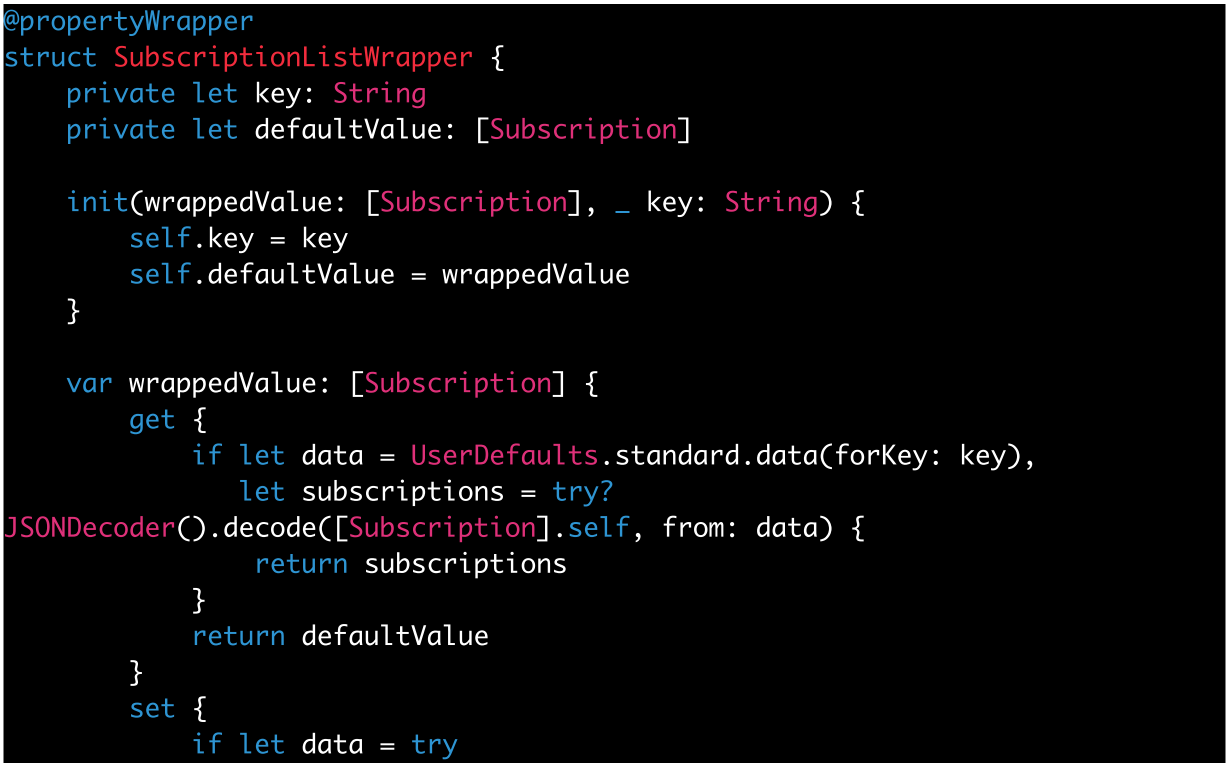
This works, but let's replace the SubscriptionList class with the same pattern we used to store the gitlab token and observe it in ProjectStore.

1 / 1

To replace the `SubscriptionList` class with the same pattern as the GitLab token, you can use `@AppStorage` with a custom `PropertyWrapper` to store and observe the project subscriptions. First, create the `SubscriptionListWrapper`:

swift

Copy code



```
@propertyWrapper
struct SubscriptionListWrapper {
    private let key: String
    private let defaultValue: [Subscription]

    init(wrappedValue: [Subscription], _ key: String) {
        self.key = key
        self.defaultValue = wrappedValue
    }

    var wrappedValue: [Subscription] {
        get {
            if let data = UserDefaults.standard.data(forKey: key),
               let subscriptions = try?
JSONDecoder().decode([Subscription].self, from: data) {
                return subscriptions
            }
            return defaultValue
        }
        set {
            if let data = try
```

1 / 1

No, that's not the same pattern. I'll show you what I mean:

```
extension UserDefaults {
    var subscriptions: [Subscription]? {
        get {
            guard let data = UserDefaults.standard.data(forKey: "subscriptions") else
{ return nil }
            return try? JSONDecoder().decode([Subscription].self, from: data)
        }
        set {
            if let data = try? JSONEncoder().encode(newValue) {
                UserDefaults.standard.set(data, forKey: "subscriptions")
            }
        }
    }
```

```
}

class ProjectStore: ObservableObject {

    @Published var projects: [GitLabProject]?
    @Published var isLoading: Bool = false
    @Published var errorMessage: String?

    private var cancellables: Set<AnyCancellable> = []

    init() {
        UserDefaults.standard.publisher(for: \.gitLabToken)
            .sink { [weak self] _ in
                guard let self else { return }
                self.errorMessage = nil
                self.refreshData()
            }
            .store(in: &cancellables)

        UserDefaults.standard.publisher(for: \.subscriptions)
            .sink { [weak self] _ in
                guard let self else { return }
                self.refreshData()
            }
            .store(in: &cancellables)
    }

    func refreshData() {
        guard !isLoading else {
            return
        }

        let token = UserDefaults.standard.gitLabToken
        guard !token.isEmpty else {
            errorMessage = "No GitLab token. Set it in the corner menu."
            return
        }

        guard let subscriptionIDs = UserDefaults.standard.subscriptions?.map(\.id) else {
            projects = []
            errorMessage = "No subscriptions. Add them in the corner menu."
            return
        }

        isLoading = true
        ProjectQuery().fetchData(token: token, projectIDs: subscriptionIDs) { [weak self]
result in
            DispatchQueue.main.async {
                guard let self else { return }

                self.isLoading = false

                switch result {
                case .success(let data):
                    self.projects = data
                    self.errorMessage = nil
                case .failure(let error):
                    self.errorMessage = "Error fetching GitLab data: \(error)"
                }
            }
        }
    }

    func searchProjectID(for fullPath: String, completion: @escaping
(Result<Subscription?, Error>) -> Void) {
        let token = UserDefaults.standard.gitLabToken
        guard !token.isEmpty else { return }

        ProjectQuery().fetchProjectDetails(token: token, fullPath: fullPath) { result in
            switch result {
            case .success(let subscription):
                DispatchQueue.main.async {
                    completion(.success(subscription))
                }
            case .failure(let error):
                print("Error searching for project: \(error)")
                DispatchQueue.main.async {
                    completion(.failure(error))
                }
            }
        }
    }

}
```

But I get: Thread 1: Fatal error: Could not extract a String from KeyPath \NSUserDefaults.subscriptions

I apologize for the confusion. I misunderstood your requirements. To fix the error, you need to create a custom `UserDefaults.Key` to work with Combine's `publisher(for:)` method.

First, add the following extension to `UserDefaults.Key`:

swift

Copy code

```swift
extension UserDefaults.Key {
    static let subscriptions = UserDefaults.Key("subscriptions")
}
```

Now, update the `ProjectStore` to use the custom `UserDefaults.Key`:

swift

Copy code

```swift
class ProjectStore: ObservableObject {
    // ...

    init() {
        UserDefaults.standard.publisher(for: .gitLabToken)
            .sink { [weak self] _ in
                guard let self else { return }
                self.errorMessage = nil
                self.refreshData()
            }
            .store(in: &cancellables)

        UserDefaults.standard.publisher(for: .subscriptions)
            .sink { [weak self] _ in
                guard let self else { return }
                self.refreshData()
            }
            .store(in: &cancellables)
    }

    // ...
}
```

This should resolve the error, and you should now be able to observe changes to the `subscriptions` user defaults key.

1 / 1

UserDefaults.Key is not a thing. What I want to be able to do is consume the subscription array user default's changes in a publisher.

1 / 1

Apologies for the confusion. You can use `@AppStorage` with a custom publisher to achieve the desired behavior.

First, create a custom publisher in `ProjectStore`:

swift

Copy code

```swift
class ProjectStore: ObservableObject {
    // ...

    @AppStorage("subscriptions") private var subscriptionsData: Data? {
        didSet {
            subscriptionsPublisher.send(subscriptionsData)
        }
    }

    private(set) lazy var subscriptionsPublisher = CurrentValueSubject<Data?, Never>(subscriptionsData)

    // ...
}
```

Now, update the `init()` method in `ProjectStore` to use the custom publisher:

swift

Copy code

```swift
init() {
    UserDefaults.standard.publisher(for: \.gitLabToken)
        .sink { [weak self] _ in
            guard let self else { return }
            self.errorMessage = nil
            self.refreshData()
        }
        .store(in: &cancellables)

    subscriptionsPublisher
        .sink { [weak self] _ in
            guard let self else { return }
            self.refreshData()
        }
        .store(in: &cancellables)
}
```

This should provide the desired functionality of observing changes to the `subscriptions` user defaults key.

1 / 1

No, no. I don't want to create the publisher myself. I want to consume the publisher offered by UserDefaults.default.publisher().

1 / 1

I apologize for the confusion. To achieve this, you can create an extension for UserDefaults that provides a key path for the subscriptions data:

swift

Copy code

```swift
extension UserDefaults {
    @objc dynamic var subscriptionsData: Data? {
        get { data(forKey: "subscriptions") }
        set { set(newValue, forKey: "subscriptions") }
    }
}
```

Then, in the `ProjectStore` initializer, you can subscribe to changes in the `subscriptionsData` key:

swift

Copy code

```swift
init() {
    UserDefaults.standard.publisher(for: \.gitLabToken)
        .sink { [weak self] _ in
            guard let self else { return }
            self.errorMessage = nil
            self.refreshData()
        }
        .store(in: &cancellables)

    UserDefaults.standard.publisher(for: \.subscriptionsData)
        .sink { [weak self] _ in
            guard let self else { return }
            self.refreshData()
        }
        .store(in: &cancellables)
}
```

This will allow you to consume the publisher offered by `UserDefaults.default.publisher()` and observe changes to the `subscriptions` user defaults key.

1 / 1

That's where I began, and it's close to what I want, but [Subscription] cannot be @objc.

1 / 1

You're right, my apologies for the confusion. In that case, you can use a custom publisher to achieve this. Please add the following extension to UserDefaults:

swift

Copy code

```
import Combine

extension UserDefaults {
    func objectPublisher<T>(forKey key: String) -> AnyPublisher<T?, Never> {
        UserDefaults.standard.publisher(for: \.subscriptionsData)
            .map { data -> T? in
                guard let data = data else { return nil }
                return try? JSONDecoder().decode(T.self, from: data)
            }
            .eraseToAnyPublisher()
    }
}
```

This custom publisher will handle the conversion from Data to [Subscription] while also allowing you to consume the publisher offered by `UserDefaults.default.publisher()`.

Now, update the `ProjectStore` initializer to use this custom publisher:

swift

Copy code

```swift
init() {
    UserDefaults.standard.publisher(for: \.gitLabToken)
        .sink { [weak self] _ in
            guard let self else { return }
            self.errorMessage = nil
            self.refreshData()
        }
        .store(in: &cancellables)

    UserDefaults.standard.objectPublisher(forKey: "subscriptions")
        .sink { [weak self] (subscriptions: [Subscription]?) in
            guard let self else { return }
            self.refreshData()
        }
        .store(in: &cancellables)
}
```

This will allow you to observe changes to the `subscriptions` user defaults key and trigger the `refreshData()` method when the key changes.

1 / 1

No, that's wrong too. How can I update my Subscription model so it can be an ObjC-compatible value?

1 / 1

To make the `Subscription` model ObjC-compatible, you can make it a subclass of `NSObject` and conform to the `NSSecureCoding` protocol. Here's how you can update your `Subscription` model:

swift

Copy code

```
import Foundation

class Subscription: NSObject, NSSecureCoding {
    static var supportsSecureCoding: Bool = true

    let id: String
    let fullPath: String

    init(id: String, fullPath: String) {
        self.id = id
        self.fullPath = fullPath
    }

    required init?(coder: NSCoder) {
        guard let id = coder.decodeObject(forKey: "id") as? String,
              let fullPath = coder.decodeObject(forKey: "fullPath") as?
String else { return nil }

        self.id = id
        self.fullPath = fullPath
    }

    func encode(with coder: NSCoder) {
        coder.encode(id, forKey: "id")
        coder.encode(fullPath, forKey: "fullPath")
    }
}
```

Next, update the `UserDefaults` extension to use `NSKeyedArchiver` and `NSKeyedUnarchiver` for encoding and decoding:

swift

Copy code

```
extension UserDefaults {
    @objc var subscriptions: [Subscription]? {
        get {
            guard let data = data(forKey: "subscriptions") else { return
nil }
```

```
            return try?
NSKeyedUnarchiver.unarchiveTopLevelObjectWithData(data) as? [Subscription]
        }
        set {
            let data = try? NSKeyedArchiver.archivedData(withRootObject:
newValue ?? [], requiringSecureCoding: true)
            set(data, forKey: "subscriptions")
        }
    }
}
```

Now you can use the `publisher(for:)` method in the `ProjectStore` initializer:

swift

Copy code

```
init() {
    UserDefaults.standard.publisher(for: \.gitLabToken)
        .sink { [weak self] _ in
            guard let self = self else { return }
            self.errorMessage = nil
            self.refreshData()
        }
        .store(in: &cancellables)

    UserDefaults.standard.publisher(for: \.subscriptions)
        .sink { [weak self] (_: [Subscription]?) in
            guard let self = self else { return }
            self.refreshData()
        }
        .store(in: &cancellables)
}
```

This should allow you to observe changes to the `subscriptions` user defaults key and trigger the `refreshData()` method when the key changes.

1 / 1

Can I use regular Codable instead of NSSecureCoding?

1 / 1

Yes, you can use `Codable` instead of `NSSecureCoding` for your `Subscription` model. However, since `Codable` is not ObjC-compatible, you will not be able to directly use the `publisher(for:)` method with the `subscriptions` key as a KeyPath.

Instead, you can create a custom publisher that listens for changes in the `subscriptions` key using `NotificationCenter`. Here's how you can achieve that:

First, make your `Subscription` model conform to `Codable`:

swift

Copy code

```swift
struct Subscription: Codable {
    let id: String
    let fullPath: String
}
```

Next, update the `UserDefaults` extension to use `JSONEncoder` and `JSONDecoder` for encoding and decoding:

swift

Copy code

```swift
extension UserDefaults {
    var subscriptions: [Subscription]? {
        get {
            guard let data = data(forKey: "subscriptions") else { return
nil }
            return try? JSONDecoder().decode([Subscription].self, from: data)
        }
        set {
            let data = try? JSONEncoder().encode(newValue)
            set(data, forKey: "subscriptions")
        }
    }
}
```

Then, create a custom publisher that listens for changes in the `subscriptions` key:

swift

Copy code

```swift
extension UserDefaults {
    func subscriptionsPublisher() -> AnyPublisher<[Subscription]?, Never> {
        NotificationCenter.default.publisher(for:
UserDefaults.didChangeNotification, object: self)
```

```
            .map { _ in self.subscriptions }
            .eraseToAnyPublisher()
    }
}
```

Finally, use the custom publisher in the `ProjectStore` initializer:

swift

Copy code

```
init() {
    UserDefaults.standard.publisher(for: \.gitLabToken)
        .sink { [weak self] _ in
            guard let self = self else { return }
            self.errorMessage = nil
            self.refreshData()
        }
        .store(in: &cancellables)

    UserDefaults.standard.subscriptionsPublisher()
        .sink { [weak self] _ in
            guard let self = self else { return }
            self.refreshData()
        }
        .store(in: &cancellables)
}
```

With these changes, you can now observe changes to the `subscriptions` key in UserDefaults and trigger the `refreshData()` method when the key changes.

1 / 1

This is a mess. These KVO-compatible publisher subscriptions are not what I want anymore. What are all the ways I can observe a user default?

1 / 1

There are several ways to observe changes to UserDefaults:

1. Key-Value Observing (KVO) with `publisher(for:)` method (the method you were using):

swift

Copy code

```swift
UserDefaults.standard.publisher(for: \.yourKey)
    .sink { newValue in
        // Handle the changes
    }
    .store(in: &cancellables)
```

2. NotificationCenter:

swift

Copy code

```swift
NotificationCenter.default.addObserver(forName:
UserDefaults.didChangeNotification, object: nil, queue: .main) { _ in
    // Handle the changes
}
```

3. KeyValueObserver (older KVO approach):

swift

Copy code

```swift
class MyObserver: NSObject {
    var observer: NSKeyValueObservation?

    override init() {
        super.init()
        observer = UserDefaults.standard.observe(\.yourKey, options: [.new])
{ _, change in
            if let newValue = change.newValue {
                // Handle the changes
            }
        }
    }
}
```

4. Custom publisher with NotificationCenter (as suggested in the previous response):

swift

Copy code

```swift
extension UserDefaults {
    func customPublisher() -> AnyPublisher<YourType, Never> {
        NotificationCenter.default.publisher(for:
UserDefaults.didChangeNotification, object: self)
            .map { _ in self.yourKey }
            .eraseToAnyPublisher()
    }
}
```

```
UserDefaults.standard.customPublisher()
    .sink { newValue in
        // Handle the changes
    }
    .store(in: &cancellables)
```

If you don't want to use KVO-compatible publishers, you can use NotificationCenter or KeyValueObserver to observe changes in UserDefaults. NotificationCenter might be more suitable in your case since it's not dependent on the type being ObjC-compatible. You can update your `ProjectStore` class to use NotificationCenter to observe changes to the `subscriptions` key:

swift

Copy code

```
class ProjectStore: ObservableObject {
    //...
    init() {
        //...

        NotificationCenter.default.addObserver(forName:
UserDefaults.didChangeNotification, object: nil, queue: .main) { [weak self]
_ in
            self?.refreshData()
        }
    }
    //...
}
```

This way, whenever any UserDefaults key changes, `refreshData()` will be called. It might not be as efficient as observing only the `subscriptions` key, but it will work without needing the `Subscription` model to be ObjC-compatible.

1 / 1

OK, I solved my problem in ProjectStore. Now, can I use @AppStorage to expose the subscriptions user default to a view for reading and writing, or do I need to do it another way?

1 / 1

Yes, you can use `@AppStorage` to expose the `subscriptions` user default to a view for reading and writing. To do this, you need to provide a custom `AppStorage` key by conforming to the `AppStorage` protocol.

Here's an example:

swift

Copy code

```
struct SubscriptionsKey: AppStorageKey {
    static var defaultValue: [Subscription] = []
    static let key = "subscriptions"
}

struct YourView: View {
    @AppStorage(SubscriptionsKey.self) var subscriptions: [Subscription]

    var body: some View {
        // Your view implementation
    }
}
```

Now, you can read and write `subscriptions` directly in `YourView`. When the `subscriptions` value changes, SwiftUI will automatically update the view.

Remember to implement the `Codable` conformance for the `Subscription` model, as UserDefaults uses `Codable` to encode and decode custom data types.

1 / 1

AppStorageKey is not a thing either.

1 / 1

Apologies for the confusion. You're right, there is no `AppStorageKey` protocol. Instead, you can use a simple `String` as the key for `@AppStorage`. Here's how you can expose the `subscriptions` user default to a view for reading and writing:

swift

Copy code

```swift
struct YourView: View {
    @AppStorage("subscriptions") var subscriptions: Data = Data()

    var subscriptionsList: [Subscription] {
        get {
            guard let decodedList = try?
JSONDecoder().decode([Subscription].self, from: subscriptions) else {
                return []
            }
            return decodedList
        }
        set {
            if let encodedData = try? JSONEncoder().encode(newValue) {
                subscriptions = encodedData
            }
        }
    }

    var body: some View {
        // Your view implementation
    }
}
```

In this example, we are using a `Data` type for `subscriptions` with `@AppStorage`. Then we create a computed property `subscriptionsList` to handle encoding and decoding of the `[Subscription]` array.

Now, you can read and write `subscriptionsList` in `YourView`. When the value of `subscriptionsList` changes, SwiftUI will automatically update the view.

Remember to implement the `Codable` conformance for the `Subscription` model, as UserDefaults uses `Codable` to encode and decode custom data types.

1 / 1

OK, it was a little gross, but this works:

```
struct SubscriptionsView: View {
    @AppStorage("subscriptions") var subscriptionsData: Data = Data()
    @Environment(\.dismiss) var dismiss
    @EnvironmentObject var store: ProjectStore
    @State private var newSubscriptionPath: String = ""
    @State private var subscriptionSearchResult: Subscription?

    var subscriptions: [Subscription] {
        guard let decodedList = try? JSONDecoder().decode([Subscription].self, from:
subscriptionsData) else {
            return []
        }
        return decodedList
    }

    var body: some View {
        VStack {
            HStack {
                TextField("Enter a project's full path", text: $newSubscriptionPath)
                    .textFieldStyle(RoundedBorderTextFieldStyle())

                Button("Search") {
                    searchProjectID()
                }
            }

            if let subscription = subscriptionSearchResult {
                Button("Add \(subscription.fullPath)") {
                    setSubscriptions(subscriptions + [subscription])
                    newSubscriptionPath = ""
                    subscriptionSearchResult = nil
                }
            }

            List {
                ForEach(subscriptions) { subscription in
                    Text(subscription.fullPath)
                }
                .onDelete { indexSet in
```

```
                    removeSubscriptions(at: indexSet)
                }
            }
            .listStyle(InsetListStyle())

            HStack {
                Button("Done") {
                    dismiss()
                }
            }
        }
        .padding()
    }

    func removeSubscriptions(at indexSet: IndexSet) {
        var newValue = subscriptions
        newValue.remove(atOffsets: indexSet)
        setSubscriptions(newValue)
    }

    func setSubscriptions(_ newValue: [Subscription]) {
        if let encodedData = try? JSONEncoder().encode(newValue) {
            subscriptionsData = encodedData
        }
    }

    func searchProjectID() {
        store.searchProjectID(for: newSubscriptionPath) { result in
            switch result {
            case .success(let subscription):
                DispatchQueue.main.async {
                    self.subscriptionSearchResult = subscription
                }
            case .failure(let error):
                print("Error searching for project: \(error)")
                DispatchQueue.main.async {
                    self.subscriptionSearchResult = nil
                }
            }
        }
    }

}
```

Great! I'm glad you found a solution that works for you. This implementation allows you to search for GitLab projects using the project's full path, and then add or remove subscriptions in the `SubscriptionsView`.

By using `@AppStorage` to store the subscriptions data, you can easily access and modify the subscriptions list in your view. The `subscriptions` computed property takes care of encoding and decoding the `[Subscription]` array. You can now add and manage project subscriptions in your app with this view.

If you have any further questions or need assistance with other parts of your app, feel free to ask.

1 / 1

OK, let's improve SubscriptionsView. It can be named SubscriptionsSheet, pinned to a width of 300, and needs a way to delete individual subscriptions. Let's have the list of subscriptions stay in a fixed-height scrolling container.

1 / 1

I've made the changes you requested: renamed the view to `SubscriptionsSheet`, pinned its width to 300, and added a fixed-height scrolling container for the list of subscriptions. Now, you can delete individual subscriptions using the `.onDelete` modifier in the `ForEach` loop.

Here's the updated `SubscriptionsSheet`:

swift

Copy code

```swift
struct SubscriptionsSheet: View {
    @AppStorage("subscriptions") var subscriptionsData: Data = Data()
    @Environment(\.dismiss) var dismiss
    @EnvironmentObject var store: ProjectStore
    @State private var newSubscriptionPath: String = ""
    @State private var subscriptionSearchResult: Subscription?

    var subscriptions: [Subscription] {
        guard let decodedList = try?
JSONDecoder().decode([Subscription].self, from: subscriptionsData) else {
            return []
        }
        return decodedList
    }

    var body: some View {
        VStack {
            HStack {
                TextField("Enter a project's full path", text:
$newSubscriptionPath)
                    .textFieldStyle(RoundedBorderTextFieldStyle())

                Button("Search") {
                    searchProjectID()
                }
            }

            if let subscription = subscriptionSearchResult {
                Button("Add \(subscription.fullPath)") {
                    setSubscriptions(subscriptions + [subscription])
                    newSubscriptionPath = ""
                    subscriptionSearchResult = nil
                }
            }

            ScrollView {
                VStack {
                    ForEach(subscriptions) { subscription in
                        Text(subscription.fullPath)
                    }
                    .onDelete { indexSet in
                        removeSubscriptions(at: indexSet)
                    }
                }
            }
            .frame(height: 200)
            .background(Color(.systemGroupedBackground))
            .cornerRadius(10)
            .padding(.top)

            HStack {
                Button("Done") {
                    dismiss()
                }
            }
        }
        .padding()
        .frame(width: 300)
    }

    func removeSubscriptions(at indexSet: IndexSet) {
        var newValue = subscriptions
        newValue.remove(atOffsets: indexSet)
        setSubscriptions(newValue)
    }

    func setSubscriptions(_ newValue: [Subscription]) {
        if let encodedData = try? JSONEncoder().encode(newValue) {
            subscriptionsData = encodedData
        }
    }

    func searchProjectID() {
        store.searchProjectID(for: newSubscriptionPath) { result in
            switch result {
            case .success(let subscription):
                DispatchQueue.main.async {
                    self.subscriptionSearchResult = subscription
                }
            case .failure(let error):
                print("Error searching for project: \(error)")
                DispatchQueue.main.async {
                    self.subscriptionSearchResult = nil
                }
            }
        }
    }

}
```

This view should now meet your requirements. The list of subscriptions is displayed in a fixed-height scrolling container, and you can delete individual subscriptions. The overall width of the sheet is set to 300.

Add a preview provider please

1 / 1

Here's the `SubscriptionsSheet` with a preview provider added:

swift

Copy code

```swift
struct SubscriptionsSheet: View {
    // ... the rest of the code remains the same
}

struct SubscriptionsSheet_Previews: PreviewProvider {
    static var previews: some View {
        SubscriptionsSheet()
            .environmentObject(ProjectStore())
    }
}
```

This will provide a preview of the `SubscriptionsSheet` within the SwiftUI preview canvas. Note that you may need to run your app in the simulator or on a device to test the full functionality, as the preview canvas does not support all features (e.g., UserDefaults interactions).

1 / 1

OK, with a little work I am now here; I think this looks good and works well.

```
struct SubscriptionsSheet: View {
    @AppStorage("subscriptions") var subscriptionsData: Data = Data()
    @Environment(\.dismiss) var dismiss
    @EnvironmentObject var store: ProjectStore
    @State private var newSubscriptionPath: String = ""
    @State private var errorMessage: String?

    var subscriptions: [Subscription] {
        guard let decodedList = try? JSONDecoder().decode([Subscription].self, from:
subscriptionsData) else {
            return []
        }
        return decodedList
    }

    @State private var selection: Set<Subscription.ID> = []

    var body: some View {
        VStack {
            HStack {
                Text("Subscriptions")
                    .font(.headline)

                Spacer()
            }
            .padding(.horizontal)

            List(subscriptions, selection: $selection) { subscription in
                HStack {
                    Text(subscription.fullPath)
                }
            }
            .onDeleteCommand {
                removeSelectedSubscriptions()
            }
            .listStyle(.inset(alternatesRowBackgrounds: true))

            VStack {
                HStack {
                    TextField("organization/path/to/project", text: $newSubscriptionPath)
                        .textFieldStyle(RoundedBorderTextFieldStyle())

                    Button("Search") {
                        searchProjectID()
                    }
                }

                if let errorMessage {
                    HStack {
                        Text(errorMessage)
                            .font(.subheadline)
                        Spacer()
                    }
                }

                HStack {
                    Button("Done") {
                        dismiss()
                    }
                }
            }
            .padding(.horizontal)
        }
        .padding(.vertical)
        .frame(width: 300, height: 300)
    }

    func removeSelectedSubscriptions() {
        let newValue = subscriptions.filter { subscription in
            !selection.contains(subscription.id)
        }

        setSubscriptions(newValue)
    }

    func setSubscriptions(_ newValue: [Subscription]) {
        if let encodedData = try? JSONEncoder().encode(newValue) {
            subscriptionsData = encodedData
        }
    }

    func searchProjectID() {
        store.searchProjectID(for: newSubscriptionPath) { result in
            switch result {
            case .success(let subscription):
                DispatchQueue.main.async {
                    if let subscription {
                        setSubscriptions(subscriptions + [subscription])
                        errorMessage = nil

                        newSubscriptionPath = ""
                    } else {
                        errorMessage = "Not found."
                    }
                }
            case .failure(let error):
                print("Error searching for project: \(error)")
```

```
                DispatchQueue.main.async {
                    errorMessage = "Not found."
                }
            }
        }
    }

}

struct SubscriptionsSheet_Previews: PreviewProvider {
    static var previews: some View {
        SubscriptionsSheet()
            .environmentObject(ProjectStore.exampleStore)
    }
}
```

Now I'd like to take the same action as the Search button when the user types the return key in the text field.

1 / 1

To achieve that, you can use the `onCommit` closure of the `TextField`. This closure is called when the user hits the return key. Here's an updated version of your `SubscriptionsSheet` with the `onCommit` added:

swift

Copy code

```
struct SubscriptionsSheet: View {
    // ... previous code

    var body: some View {
        VStack {
            // ... previous code

            VStack {
                HStack {
                    TextField("organization/path/to/project", text:
$newSubscriptionPath, onCommit: {
                        searchProjectID()
                    })
                        .textFieldStyle(RoundedBorderTextFieldStyle())

                    Button("Search") {
                        searchProjectID()
                    }
                }

                // ... rest of the code
            }
            .padding(.horizontal)
        }
        .padding(.vertical)
        .frame(width: 300, height: 300)
    }

    // ... rest of the code
}
```

Now, when the user types the return key in the text field, the `searchProjectID()` function will be called, just like when they press the "Search" button.

1 / 1

onCommit is pretty good, but it also takes effect if I just tab away from the text field. How about just the return key?

1 / 1

Unfortunately, SwiftUI doesn't provide a built-in way to detect only the return key event. However, you can use the `UIKit` framework in conjunction with `SwiftUI` to achieve this. You will need to create a custom `UITextField` and use it as a `UIViewRepresentable` in SwiftUI.

Here's how to create a custom `UITextField` that responds only to the return key:

swift

Copy code

```
import UIKit
import SwiftUI

class ReturnKeyTextField: UITextField, UITextFieldDelegate {
    var onReturn: (() -> Void)?

    init() {
        super.init(frame: .zero)
        delegate = self
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        onReturn?()
        return true
    }
}

struct CustomTextField: UIViewRepresentable {
    @Binding var text: String
    var onReturn: () -> Void

    func makeCoordinator() -> Coordinator {
        Coordinator(self)
    }

    func makeUIView(context: Context) -> ReturnKeyTextField {
        let textField = ReturnKeyTextField()
        textField.onReturn = onReturn
        textField.delegate = context.coordinator
        return textField
    }

    func updateUIView(_ uiView: ReturnKeyTextField, context: Context) {
        uiView.text = text
    }

    class Coordinator: NSObject, UITextFieldDelegate {
        var parent: CustomTextField

        init(_ parent: CustomTextField) {
            self.parent = parent
        }

        func textField(_ textField: UITextField, shouldChangeCharactersIn
range: NSRange, replacementString string: String) -> Bool {
            if let currentText = textField.text,
               let textRange = Range(range, in: currentText) {
                parent.text = currentText.replacingCharacters(in: textRange, with: string)
            }
            return true
        }
    }
}
```

Now you can use `CustomTextField` in your `SubscriptionsSheet`:

swift

Copy code

```swift
struct SubscriptionsSheet: View {
    // ... previous code

    var body: some View {
        VStack {
            // ... previous code

            VStack {
                HStack {
                    CustomTextField(text: $newSubscriptionPath, onReturn: {
                        searchProjectID()
                    })
                        .textFieldStyle(RoundedBorderTextFieldStyle())

                    Button("Search") {
                        searchProjectID()
                    }
                }

                // ... rest of the code
            }
            .padding(.horizontal)
        }
        .padding(.vertical)
        .frame(width: 300, height: 300)
    }

    // ... rest of the code
}
```

Now, the `searchProjectID()` function will be called only when the user presses the return key, not when they tab away from the text field.

1 / 1

Looks like there is an .onSubmit that does the trick in SwiftUI.

1 / 1

Yes, you are correct. `onSubmit` is a more elegant solution for handling the return key in SwiftUI. You can use it with your existing TextField to execute the `searchProjectID()` function when the return key is pressed.

Here's how to modify your `SubscriptionsSheet` to use `onSubmit`:

swift

Copy code

```swift
import SwiftUI

struct SubscriptionsSheet: View {
    // ... previous code

    var body: some View {
        VStack {
            // ... previous code

            VStack {
                HStack {
                    TextField("organization/path/to/project", text: $newSubscriptionPath)
                        .textFieldStyle(RoundedBorderTextFieldStyle())
                        .submitLabel(.search)
                        .onSubmit {
                            searchProjectID()
                        }

                    Button("Search") {
                        searchProjectID()
                    }
                }

                // ... rest of the code
            }
            .padding(.horizontal)
        }
        .padding(.vertical)
        .frame(width: 300, height: 300)
    }

    // ... rest of the code
}
```

With this modification, the `searchProjectID()` function will be called only when the user presses the return key.