# Tasks

## 0. JSON files

In the NetworkAssembler, on line 14, you'll find the following code: **MockApiClient(fileName: MockFile.packs.rawValue)**. This code determines which JSON data should be provided. By default, it uses the one that comes with the recruitment task, but you also have the option to change it to other data sources.

## 1. Add grouping to the list of Packs

Based on the provided Figma Design, two groups have been implemented:

- **inTransit:** Packages that are or should be delivered to the client.
- **deliveryCompleted:** Packages that are no longer in transit to the client.

In the Figma design, the first group was called "Gotowe do odbioru" ("Ready for pickup" in English), which might be a little confusing - consider renaming it. The second group is named "Pozostałe" ("Remaining" in English).

**Statuses in the "inTransit" group:**

- CREATED: Przesyłka utworzona
- NOT_READY: W trakcie przygotowania
- CONFIRMED: Odebrana od Nadawcy
- ADOPTED_AT_SOURCE_BRANCH: Przyjęta w Oddziale
- SENT_FROM_SOURCE_BRANCH: Wysłana z Oddziału
- ADOPTED_AT_SORTING_CENTER: Przyjęta w sortowni
- SENT_FROM_SORTING_CENTER: Wysłana z sortowni
- OTHER: Inne
- AVIZO: Pozostawiono Awizo
- OUT_FOR_DELIVERY: Wydana do doręczenia
- READY_TO_PICKUP: Gotowa do odbioru
- PICKUP_TIME_EXPIRED: Upłynął termin odbioru
- UNSUPPORTED: Inne

**Statuses in the "deliveryCompleted" group:**

- DELIVERED: Odebrana
- RETURNED_TO_SENDER: Zwrot do Nadawcy
- PICKUP_TIME_EXPIRED: Upłynął termin odbioru

**Assumption:** Assumptions made by the Developer should be discussed with the Project Manager, Product Owner , or Backend Developer who can explain real-life examples.

- Based on real-life experience and the official InPost site inPost - Aktualności - Poznaj drogę przesyłki do Paczkomatu InPost, the "NOT_READY" status, which was not mentioned in the files but was in the JSON example, is positioned between "CREATED" and "CONFIRMED."
- From a code perspective, the "OTHER" status appears to be more suitable for the "inTransit" group, but this may be subject to verification.
- The "UNSUPPORTED" status is translated as "Inne" because in future versions of the app, there may be cases where the backend sends new statuses that the app does not yet support, so a generic "Inne" is used to display such statuses in the app.

## 2. Style as requested from the Design department

The task is all set - nothing more to include.

## 3. Sort list items in groups by

- Convert "Status" to an integer value where each status has a value 10 greater than the previous one:
  - CREATED: 10
  - NOT_READY: 20
  - CONFIRMED: 30
  - …
  - PICKUP_TIME_EXPIRED: 140
  - UNSUPPORTED: -1

  This numeric representation allows for easy addition of new statuses in the future without major changes in the code. Sort in descending order, so that items with higher values are at the top, as users prefer to see packages ready for pickup first and newly created ones at the bottom. Note: "NOT_READY" falls between "CREATED" and "CONFIRMED" based on the numeric representation.

  Additionally, there's an "UNSUPPORTED" status with a converted value of -1. This is included to handle unsupported or erroneous statuses and should be placed at the very bottom of the group.
- Sort is done by algorithm as follows:
  - "sortOrderNumber" in descending order
  - "pickupDate" in descending order, so that items with the most recent pickup date are first.
  - "expireDate" in ascending order, prioritizing items that need to be picked up sooner.
  - "storeDate" is sorted similarly to "expireDate.
  - "id" in descending order.

## 4. Add pull to refresh and handle refresh progress

- This task is completed. It's worth mentioning that in **MockApiClient.swift**, on line 13, there is code that includes **try await Task.sleep(nanoseconds: 2,000,000,000)**, which creates a 2-second delay for a mock response. This allows us to test the loader and how pull-to-refresh works.
- One thing I didn't do is error handling. In **PackListViewModel.swift**, within the **func loadData()**, there's a catch block, which should be filled with code to change the state to "error" or "tryAgain". Additionally, we should add UI elements to our view to reflect this state."

## 5. Add storing shipments locally (using Realm)

- I've decided not to do this task because it's quite similar to Task 6 (but this is more time-consuming). There are several additional considerations, such as determining how and where to have and create database data models. I believe it should be placed in the DataStorage Package, but we might also need to consider creating another package.

## 6. Add local archiving of the shipment:

- When a user taps on a cell, the cell expands and displays an 'Archive' button. After the user taps 'Archive,' the cell's ID will be sent and stored in UserDefaults. Based on this, we can hide packs with IDs stored in UserDefaults. Furthermore, because we use UserDefaults, even if the user kills the app, we will still have the data of hidden packs.

## 7. Fix one Pack appearing with unsupported status

- ‘NOT_READY’ is a status that falls between ‘CREATED’ and ‘CONFIRMED’.
- ‘BROKEN_STATUS’ has a ‘Bad server response’ value in the ‘sender’ key, but at the same time, it has correct values for ‘shipmentType’ and ‘id’. Therefore, my assumption is that there’s some kind of error on the backend side, such as a failure in data retrieval from microservices. This should be considered a backend issue and should be addressed there. On the iOS app side, we should display it with the status ‘UNSUPPORTED’.
- Additionally, in the app, there is an enum called ‘Supportable,’ which is well-suited for handling such cases for two reasons:
  - If anything goes wrong on the backend side, we can still display data and avoid crashing our app, even if a new status comes from the backend that wasn’t present before.
  - In the future, there will likely be more statuses. If a user has an older version of the app and the backend introduces new statuses, the user’s app will not crash because ‘Supportable’ can gracefully handle new statuses."

## 8. Fix ViewController title disappearing in Dark Mode

- Since I’m using SwiftUI and completely rebuilt the view, this issue no longer occurs. The app now functions in both light and dark modes, but it does not support a dark theme.

# 9. Create unit tests

In this section, I will be using the word “screen”, which should be understood as the entire composition of View, ViewModel, Interactor, Presentables, DTOs, and Services that make up the real screen that the user interacts with.

- Unit tests were created following the principles of Test-Driven Development (TDD). The architecture I’ve designed comes with certain assumptions about how it should be tested and why.
- **View:**
  - We don’t typically test the View with unit tests. The View should not contain any logic, except for minimal cases such as checking if **viewModel.showFoo()** or **if let x = presentable.x**. Both of these examples only decide to whether or not a certain part of the view should be displayed, but the underlying logic should be handled elsewhere in the application.
- **ViewModel:**
  - The ViewModel acts as the “brain” of our Screen. This is the most critical and vulnerable part of the screen, and it’s essential to test it carefully, preferably using TDD. It’s where most of the main logic resides, and in the event of a bug, developers won’t need to search through multiple files to find the issue. However, for complex ViewModels or screens, it may be beneficial to split them into smaller files or even move advanced logic into separate classes or structs.
- **Interactor:**
  - Interactors don’t require testing by default because they serve as a bridge between the App and Domain modules. There might be cases where an Interactor calls two different Domain modules or services within a single function, and in such situations, it’s worth considering adding unit tests.
- **Presentable:**
  - Presentables are objects that provide data for our views. We map DTO objects to Presentables. In most cases, Presentables closely resemble DTOs, but they may not be identical. For instance, a **PackDTO** contain a variable **Status** that’s an enum, but in **PackPresentable**, we only need to know the status as a string for display purposes, so we map the Status Enum to a Status String. Similarly, **PackDTO** contains an enum **shipmentType** with values **COURIER** or **PARCEL_LOCKER**, **PackPresentable** might only need information about which icon to display based on the *shipmentType enum. In our solution, we handle this mapping within the initialization of PackPresentable. If multiple Presentables require the same mapping, it’s a good idea to consider creating a Mapper struct and moving the tests there to avoid duplicating code and tests.
- **DTOs:**
  - There’s no need to test DTOs, but it’s beneficial to have a well-documented contract with the backend to ensure compatibility.

- **Services:**
  - Services are primarily responsible for creating requests and using APIClient or DataStorage. They should be tested. However, when dealing with large DTOs, there may be some challenges. In such cases, we may need to omit certain tests that TDD theory might suggest.
- **Other Parts of the App:**
  - **Networking and DataStorage:**
    - Networking and DataStorage packages must be tested since they represent high-risk areas. Failures in these components can result in significant issues.
  - **Assemblers:**
    - The decision to test Assemblers, which are part of every package and module should be made by the team. In the current state of the app, there might not be a need to add tests.
  - **Snapshot Tests:**
    - I believe we should implement snapshot tests. They can be highly beneficial. Usually, I would add them, but I didn't want to invest more time in the project, considering its already significant size.
- **Why we shouldn't test all parts:**
  - While TDD is an effective process for maintaining well-structured code and covering numerous edge cases, striving for 100% code coverage can be time-consuming. This approach allows developers to skip some of the more tedious, boilerplate, and low-value tests, such as those for the Interactor, and allocate their time more efficiently to other aspects of the development process.