Offline-first support

An offline-first application is an app capable of offering most or all of its functionality while being disconnected from the internet. Offline-first applications usually rely on stored data to offer users temporary access to data that would otherwise only be available online.

Some offline-first applications combine local and remote data seamlessly, while other applications inform the user when the application is using cached data. In the same way, some applications synchronize data in the background while others require the user to explicitly synchronize it. It all depends on the application requirements and the functionality it offers, and it's up to the developer to decide which implementation fits their needs.

In this guide, you will learn how to implement different approaches to offline-first applications in Flutter, following the Flutter Architecture guidelines.

Offline-first architecture

This example uses the `UserProfileRepository`, a repository that allows you to obtain and store `UserProfile` objects with offline-first support.

The `UserProfileRepository` uses two different data services: one works with remote data, and the other works with a local database.

The API client,`ApiClientService`, connects to a remote service using HTTP REST calls.

```dart
class ApiClientService {
  /// performs GET network request to obtain a UserProfile
  Future<UserProfile> getUserProfile() async {
    // ···
  }

  /// performs PUT network request to update a UserProfile
  Future<void> putUserProfile(UserProfile userProfile) async {
    // ···
  }
}
```

The database service, `DatabaseService`, stores data using SQL, similar to the one found in the Persistent Storage Architecture: SQL recipe.

```dart
class DatabaseService {
  /// Fetches the UserProfile from the database.
  /// Returns null if the user profile is not found.
  Future<UserProfile?> fetchUserProfile() async {
    // ···
  }

  /// Update UserProfile in the database.
  Future<void> updateUserProfile(UserProfile userProfile) async {
    // ···
  }
}
```

This example also uses the `UserProfile` data class that has been created using the `freezed` package.

```dart
@freezed
class UserProfile with _$UserProfile {
  const factory UserProfile({
    required String name,
    required String photoUrl,
  }) = _UserProfile;
}
```

In apps that have complex data, such as when the remote data contains more fields than the needed by the UI, you might want to have one data class for the API and database services, and another for the UI. For example, `UserProfileLocal` for the database entity, `UserProfileRemote` for the API response object, and then `UserProfile` for the UI data model class. The `UserProfileRepository` would take care of converting from one to the other when necessary.

This example also includes the `UserProfileViewModel`, a view model that uses the `UserProfileRepository` to display the `UserProfile` on a widget.

```dart
class UserProfileViewModel extends ChangeNotifier {
  // ···
  final UserProfileRepository _userProfileRepository;

  UserProfile? get userProfile => _userProfile;
  // ···

  /// Load the user profile from the database or the network
  Future<void> load() async {
    // ···
  }

  /// Save the user profile with the new name
  Future<void> save(String newName) async {
    // ···
  }
}
```

Reading data

Reading data is a fundamental part of any application that relies on remote API services.

In offline-first applications, you want to ensure that the access to this data is as fast as possible, and that it doesn't depend on the device being online to provide data to the user. This is similar to the Optimistic State design pattern.

In this section, you will learn two different approaches, one that uses the database as a fallback, and one that combines local and remote data using a `Stream`.

Using local data as a fallback

As a first approach, you can implement offline support by having a fallback mechanism for when the user is offline or a network call fails.

In this case, the `UserProfileRepository` attempts to obtain the `UserProfile` from the remote API server using the `ApiClientService`. If this request fails, then returns the locally stored `UserProfile` from the `DatabaseService`.

```dart
Future<UserProfile> getUserProfile() async {
  try {
    // Fetch the user profile from the API
    final apiUserProfile = await _apiClientService.getUserProfile(
    //Update the database with the API result
    await _databaseService.updateUserProfile(apiUserProfile);

    return apiUserProfile;
  } catch (e) {
    // If the network call failed,
    // fetch the user profile from the database
    final databaseUserProfile = await _databaseService.fetchUserPr

    // If the user profile was never fetched from the API
    // it will be null, so throw an  error
    if (databaseUserProfile != null) {
      return databaseUserProfile;
    } else {
      // Handle the error
      throw Exception('User profile not found');
    }
  }
}
```

Using a Stream

A better alternative presents the data using a `Stream`. In the best case scenario, the `Stream` emits two values, the locally stored data, and the data from the server.

First, the stream emits the locally stored data using the `DatabaseService`. This call is generally faster and less error prone than a network call, and by doing it first the view model can already display data to the user.

If the database does not contain any cached data, then the `Stream` relies completely on the network call, emitting only one value.

Then, the method performs the network call using the `ApiClientService` to obtain up-to-date data. If the request was successful, it updates the database with the newly obtained data, and then yields the value to the view model, so it can be displayed to the user.

```dart
Stream<UserProfile> getUserProfile() async* {
  // Fetch the user profile from the database
  final userProfile = await _databaseService.fetchUserProfile();
  // Returns the database result if it exists
  if (userProfile != null) {
    yield userProfile;
  }

  // Fetch the user profile from the API
  try {
    final apiUserProfile = await _apiClientService.getUserProfile(
    //Update the database with the API result
    await _databaseService.updateUserProfile(apiUserProfile);
    // Return the API result
    yield apiUserProfile;
  } catch (e) {
    // Handle the error
  }
}
```

The view model must subscribe to this `Stream` and wait until it has completed. For that, call `asFuture()` with the `Subscription` object and await the result.

For each obtained value, update the view model data and call `notifyListeners()` so the UI shows the latest data.

```dart
Future<void> load() async {
  await _userProfileRepository.getUserProfile().listen((userProfil
    _userProfile = userProfile;
    notifyListeners();
  }, onError: (error) {
    // handle error
  }).asFuture();
}
```

Using only local data

Another possible approach uses locally stored data for read operations. This approach requires that the data has been preloaded at some point into the database, and requires a synchronization mechanism that can keep the data up to date.

```dart
Future<UserProfile> getUserProfile() async {
  // Fetch the user profile from the database
  final userProfile = await _databaseService.fetchUserProfile();

  // Return the database result if it exists
  if (userProfile == null) {
    throw Exception('Data not found');
  }
```

```
    return userProfile;
  }

  Future<void> sync() async {
    try {
      // Fetch the user profile from the API
      final userProfile = await _apiClientService.getUserProfile();

      // Update the database with the API result
      await _databaseService.updateUserProfile(userProfile);
    } catch (e) {
      // Try again later
    }
  }
```

This approach can be useful for applications that don't require data to be in sync with the server at all times. For example, a weather application where the weather data is only updated once a day.

Synchronization could be done manually by the user, for example, a pull-to-refresh action that then calls the `sync()` method, or done periodically by a `Timer` or a background process. You can learn how to implement a synchronization task in the section about synchronizing state.

Writing data

Writing data in offline-first applications depends fundamentally on the application use case.

Some applications might require the user input data to be immediately available on the server side, while other applications might be more flexible and allow data to be out-of-sync temporarily.

This section explains two different approaches for implementing writing data in offline-first applications.

Online-only writing

One approach for writing data in offline-first applications is to enforce being online to write data. While this might sound counterintuitive, this ensures that the data the user has modified is fully synchronized with the server, and the application doesn't have a different state than the server.

In this case, you first attempt to send the data to the API service, and if the request succeeds, then store the data in the database.

```dart
Future<void> updateUserProfile(UserProfile userProfile) async {
  try {
    // Update the API with the user profile
    await _apiClientService.putUserProfile(userProfile);

    // Only if the API call was successful
    // update the database with the user profile
    await _databaseService.updateUserProfile(userProfile);
  } catch (e) {
    // Handle the error
  }
}
```

The disadvantage in this case is that the offline-first functionality is only available for read operations, but not for write operations, as those require the user being online.

Offline-first writing

The second approach works the other way around. Instead of performing the network call first, the application first stores the new data in the database, and then attempts to send it to the API service once it has been stored locally.

```dart
Future<void> updateUserProfile(UserProfile userProfile) async {
  // Update the database with the user profile
  await _databaseService.updateUserProfile(userProfile);

  try {
    // Update the API with the user profile
    await _apiClientService.putUserProfile(userProfile);
  } catch (e) {
    // Handle the error
  }
}
```

This approach allows users to store data locally even when the application is offline, however, if the network call fails, the local database and the API service are no longer in sync. In the next section, you will learn different approaches to handle synchronization between local and remote data.

Synchronizing state

Keeping the local and remote data in sync is an important part of offline-first applications, as the changes that have been done locally need to be copied to the remote service. The app must also ensure that, when the user goes back to the application, the locally stored data is the same as in the remote service.

Writing a synchronization task

There are different approaches for implementing synchronization in a background task.

A simple solution is to create a `Timer` in the `UserProfileRepository` that runs periodically, for example every five minutes.

```dart
Timer.periodic(
  const Duration(minutes: 5),
  (timer) => sync(),
);
```

The `sync()` method then fetches the `UserProfile` from the database, and if it requires synchronization, it is then sent to the API service.

```dart
Future<void> sync() async {
  try {
    // Fetch the user profile from the database
    final userProfile = await _databaseService.fetchUserProfile();

    // Check if the user profile requires synchronization
    if (userProfile == null || userProfile.synchronized) {
      return;
    }
```

```
      // Update the API with the user profile
      await _apiClientService.putUserProfile(userProfile);

      // Set the user profile as synchronized
      await _databaseService
          .updateUserProfile(userProfile.copyWith(synchronized: true
    } catch (e) {
      // Try again later
    }
  }
```

A more complex solution uses background processes like the `workmanager` plugin. This allows your application to run the synchronization process in the background even when the application is not running.

> **Note**
>
> Running background operations continuosly can drain the device battery dramatically, and some devices limit the background processing capabilities, so this approach needs to be tuned to the application requirements and one solution might not fit all cases.

It's also recommended to only perform the synchronization task when the network is available. For example, you can use the `connectivity_plus` plugin to check if the device is connected to WiFi. You can also use `battery_plus` to verify that the device is not running low on battery.

In the previous example, the synchronization task runs every 5 minutes. In some cases, that might be excessive, while in others it might not be frequent enough. The actual synchronization period time for your application depends on your application needs and it's something you will have to decide.

Storing a synchronization flag

To know if the data requires synchronization, add a flag to the data class indicating if the changes need to be synchronized.

For example, `bool synchronized`:

```dart
@freezed
class UserProfile with _$UserProfile {
  const factory UserProfile({
    required String name,
    required String photoUrl,
    @Default(false) bool synchronized,
  }) = _UserProfile;
}
```

Your synchronization logic should attempt to send it to the API service only when the `synchronized` flag is `false`. If the request is successful, then change it to `true`.

Pushing data from server

A different approach for synchronization is to use a push service to provide up-to-date data to the application. In this case, the server notifies the application when data has changed, instead of being the application asking for updates.

For example, you can use Firebase messaging, to push small payloads of data to the device, as well as trigger synchronization tasks remotely using background messages.

Instead of having a synchronization task running in the background, the server notifies the application when the stored data needs to be updated with a push notification.

You can combine both approaches together, having a background synchronization task and using background push messages, to keep the application database synchronized with the server.

Putting it all together

Writing an offline-first application requires making decisions regarding the way read, write and sync operations are implemented, which depend on the requirements from the application you are developing.

The key takeaways are:

- When reading data, you can use a `Stream` to combine locally stored data with remote data.
- When writing data, decide if you need to be online or offline, and if you need synchronizing data later or not.

- When implementing a background sync task, take into account the device status and your application needs, as different applications may have different requirements.

Unless stated otherwise, the documentation on this site reflects the latest stable version of Flutter. Page last updated on 2024-12-11. View source or report an issue.