Modular Services State Management and Leveraging Pods for Flutter

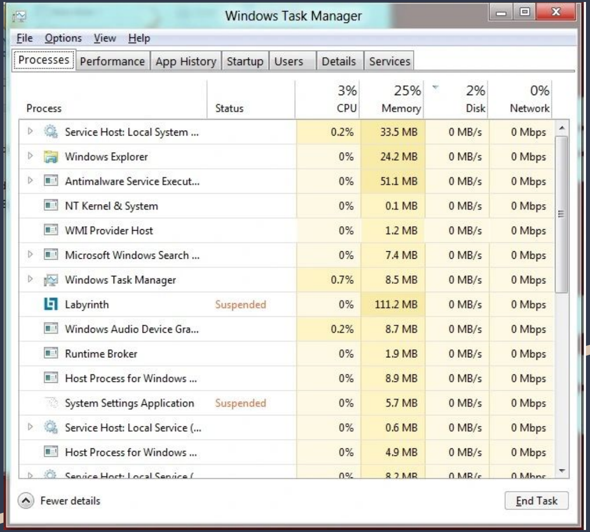
By github.com/robmllze

What is MSSM?

Modular Services State Management, or MSSM, is a state management approach that emphasizes the division of an application's state into distinct, modular services, each responsible for managing a specific domain of the application's data.

Each service could represent a feature of your application, for example, a *UserService* could manage incoming user data, and an *EventService* could manage incoming user event data, such messages.

How do Services Work?



Process	Status	3% CPU	25% Memory	2% Disk	0% Network
Service Host: Local System ...		0.2%	33.5 MB	0 MB/s	0 Mbps
Windows Explorer		0%	24.2 MB	0 MB/s	0 Mbps
Antimalware Service Execut...		0%	51.1 MB	0 MB/s	0 Mbps
NT Kernel & System		0%	0.1 MB	0 MB/s	0 Mbps
WMI Provider Host		0%	1.2 MB	0 MB/s	0 Mbps
Microsoft Windows Search ...		0%	7.4 MB	0 MB/s	0 Mbps
Windows Task Manager		0.7%	8.5 MB	0 MB/s	0 Mbps
Labyrinth	Suspended	0%	111.2 MB	0 MB/s	0 Mbps
Windows Audio Device Gra...		0.2%	8.7 MB	0 MB/s	0 Mbps
Runtime Broker		0%	1.9 MB	0 MB/s	0 Mbps
Host Process for Windows ...		0%	8.9 MB	0 MB/s	0 Mbps
System Settings Application	Suspended	0%	5.7 MB	0 MB/s	0 Mbps
Service Host: Local Service (...		0%	0.6 MB	0 MB/s	0 Mbps
Host Process for Windows ...		0%	4.9 MB	0 MB/s	0 Mbps
Service Host: Local Service (...		0%	8.2 MB	0 MB/s	0 Mbps

In the context of MSSM, a service functions similarly to a background process, akin to those in Windows. Such processes commence operation either upon the startup of Windows or when a user logs into the system.

Throughout their runtime, these services engage in various activities including streaming data from the web, processing inputs, communicating with other services, etc.

An application can be thought of a symphony of services, each with a specific responsibility, yet working together in unison.

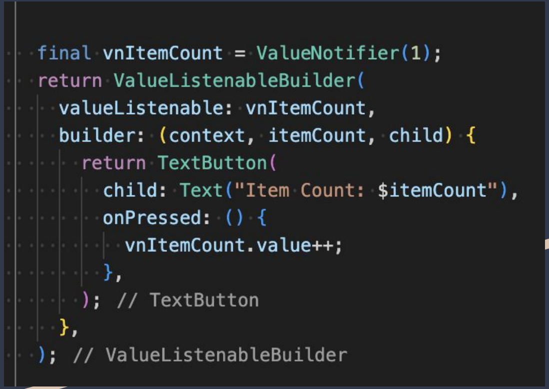
Data Nodes

Data nodes in the context of MSSM are essentially variables or objects that represent the current state of a service.

In an OS, services expose data that can be routinely checked for changes through an event loop. This data can subsequently be processed then returned to available services for further actions or integrated into Widgets for display in the UI.

In a Flutter application utilizing MSSM, a practical approach is to define each data node within a service as a *ValueNotifier*. This enables tracking changes to the data node and seamlessly integrating the updated data into the UI using a *ValueListenableBuilder*.

ValueNotifier & ValueListenableBuilder

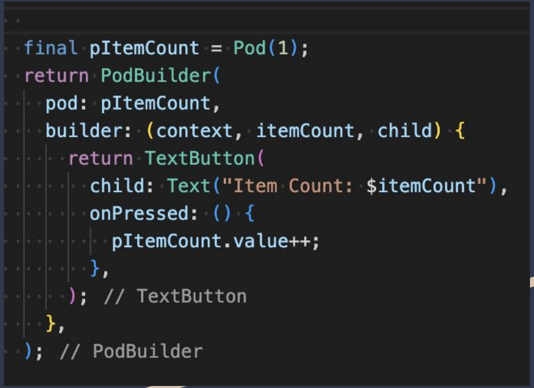


```dart
final vnItemCount = ValueNotifier(1);
return ValueListenableBuilder(
  valueListenable: vnItemCount,
  builder: (context, itemCount, child) {
    return TextButton(
      child: Text("Item Count: $itemCount"),
      onPressed: () {
        vnItemCount.value++;
      },
    ); // TextButton
  },
); // ValueListenableBuilder
```

A *ValueNotifier* is a Flutter class designed for holding a single value and notifying listeners when this value changes. It's part of Flutter's foundation library and is particularly useful for managing state in a more granular way.

ValueListenableBuilder is a specialized widget that listens to a *ValueNotifier* and rebuilds its child widget whenever the ValueNotifier's value changes.

Pod & PodBuilder



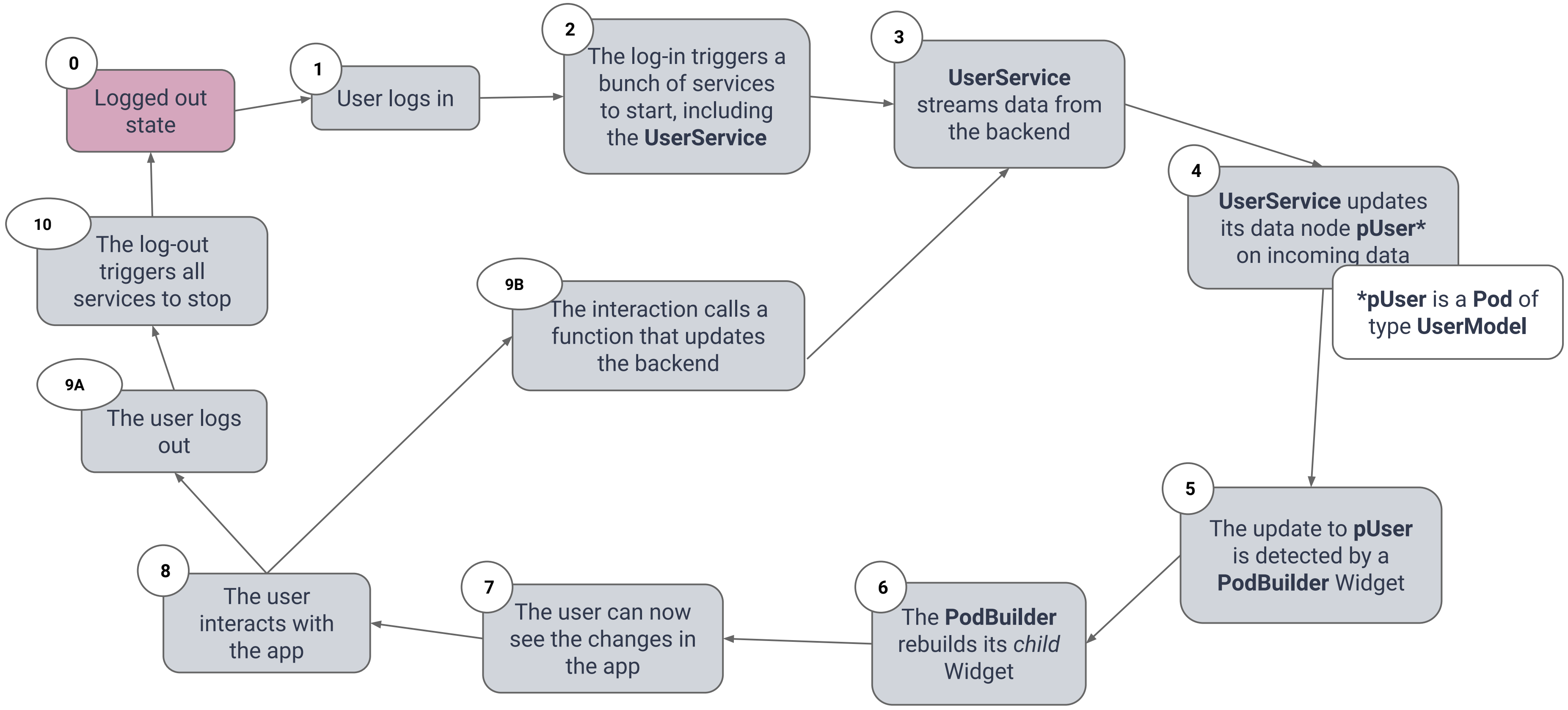
```
final pItemCount = Pod(1);
return PodBuilder(
  pod: pItemCount,
  builder: (context, itemCount, child) {
    return TextButton(
      child: Text("Item Count: $itemCount"),
      onPressed: () {
        pItemCount.value++;
      },
    ); // TextButton
  },
); // PodBuilder
```

But… let's use the Pod instead!

The Pod offers advantages over the ValueNotifier in Flutter due to its advanced features, including support for state updates during the build phase, improved handling of complex data types like lists and maps, and more efficient management of multiple state objects in *PodListBuilder* or *ResponsivePodListBuilder*.

Therefore, instead of using a *ValueNotifier*, you can use a *Pod*, and instead of a *ValueListenableBuilder*, you can employ a *PodBuilder*.

See: https://pub.dev/packages/xyz_pod


0
Logged out state
1
User logs in
2
The log-in triggers a bunch of services to start, including the **UserService**
3
UserService streams data from the backend
4
UserService updates its data node **pUser*** on incoming data
***pUser** is a **Pod** of type **UserModel**
5
The update to **pUser** is detected by a **PodBuilder** Widget
6
The **PodBuilder** rebuilds its *child* Widget
7
The user can now see the changes in the app
8
The user interacts with the app
9A
The user logs out
9B
The interaction calls a function that updates the backend
10
The log-out triggers all services to stop

Example: UserService Flow with pUser Pod in MSSM

Example 1 - Pod and PodBuilder

Task:

Build a Flutter application that includes a Text field and a Button. When the button is pressed, the Text field should increment its value starting at 0. Use a Pod and PodBuilder.

Example 1 – Pod and PodBuilder

Approach:

1. We declare a data node named pCount in our code.
2. We utilize a PodBuilder widget, assigning pCount to its pod property.
3. Within the builder property of PodBuilder, we create a Text widget displaying the count.
4. We increment the value of pCount using the update function.
5. The PodBuilder invokes its build function each time pCount is updated.

Note: Always dispose Pods with *.dispose()* when they are no longer needed to free up resources.

```dart
import 'package:flutter/material.dart';
import 'package:xyz_pod/xyz_pod.dart';

final pCount = Pod<int>(0);

void main() {
 runApp(const App());
}

class App extends StatelessWidget {
 const App({super.key});
 @override
 Widget build(_) {
   return MaterialApp(
     home: Material(
       child: Column(
         children: [
           PodBuilder(
             pod: pCount,
             builder: (context, child, count) {
               return Text("Count: $count");
             },
           ),
           TextButton(
             onPressed: () => pCount.update((e) => e + 1),
             child: const Text("Increment"),
           ),
         ],
       ),
     ),
   );
 }
}
```

Example 2 - Pod and PodListBuilder

Task:

Create a Flutter app with a Text field and two Buttons. One button increases the value of pCount1 by 1, and the other decreases the value of pCount2 by 2. Calculate and display the total using a Pod and PodListBuilder for this task.

Example 2 – The ugly way...

Approach:

1. We use a PodBuilder for each Pod, nesting one within the build function of the other.
2. We calculate the sum of the two counts and display it using a Text widget.

Issues:

- This approach, while functional, can be confusing and difficult to follow due to the nested structure of PodBuilders.
- It might lead to increased complexity as more Pods or calculations are added, potentially making the code harder to maintain and debug.
- Nesting PodBuilders can also result in less efficient updates and rebuilds, impacting app performance when dealing with large and complex UIs.

```dart
final pCount1 = Pod<int>(0);
final pCount2 = Pod<int>(0);

Column(
 children: [
   PodBuilder(
     pod: pCount1,
     builder: (context, child, count1) {
       return PodBuilder(
         pod: pCount2,
         builder: (context, child, count2) {
           final sum = count1 + count2;
           return Text("Sum: $sum");
         },
       );
     },
   ),
   TextButton(
     onPressed: () => pCount1.update((e) => e + 1),
     child: const Text("Increment 1"),
   ),
   TextButton(
     onPressed: () => pCount2.update((e) => e - 2),
     child: const Text("Decrement 2"),
   ),
 ],
);
```

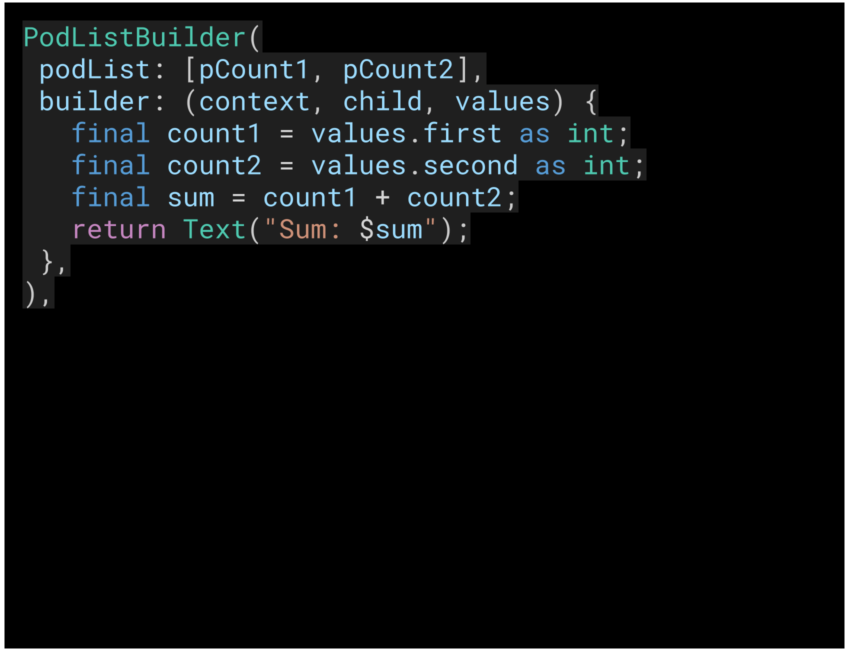
Example 2 – The right way...

Approach:

1. We use a PodListBuilder, which simplifies the process by directly managing a list of Pods.
   We provide the list of pods in the podList property.
2. In the builder function, we access count1 as the first element and count2 as the second element from the values in the same order as they were supplied in the podList.
3. We calculate the sum of the two counts and display it using a Text widget.

Advantages:

- Improved readability and maintainability, making it easier for developers to understand the code.
- Reduced complexity as the nesting of PodBuilders is avoided, leading to more efficient updates and a cleaner codebase.
- Enhanced performance, especially in scenarios with numerous Pods, as updates are managed more efficiently.



```
PodListBuilder(
 podList: [pCount1, pCount2],
 builder: (context, child, values) {
   final count1 = values.first as int;
   final count2 = values.second as int;
   final sum = count1 + count2;
   return Text("Sum: $sum");
 },
),
```

Example 3 – Service Pods

Now imagine we have this service responsible for syncing the user data on the database with its data node, pUser. It takes some time to start up, hence the async create function.

How can we use Pods to display the user data, once the service is created?

```
final pUserService = Pod<UserService?>(null);
final userService = await UserService.create();
pUserService.set(userService);
```

```
class UserService {
 final pUser = Pod<UserModel?>(null);
 UserService._();

 static Future<UserService> create() async {
   // Do some stuff here that takes time...
   return UserService._();
 }

 Future<void> startService() async {
   // Stream the database and update this.pUser each time we get new data.
 }
 Future<void> stopService() async {
   // Stop the stream.
 }

 void disposeResources() {
   this.pUser.dispose();
 }
}

class UserModel {
 final String id;
 final String email;
 const UserModel({
   required this.id,
   required this.email,
 });
```

Example 3 – The longer but intuitive way...

Approach:

1. We monitor pUserService, and when it’s value becomes non-null, we observe the pUser pod within pUserService.value.
2. We can then access the value of pUser and display the user email.

```
final pUserService = Pod<UserService?>(null);
final userService = await UserService.create();
pUserService.set(userService);
```

```
PodBuilder(
 pod: pUserService,
 builder: (context, child, userService) {
   if (userService != null) {
     return PodBuilder(
       pod: userService.pUser,
       builder: (context, child, user) {
         if (user != null) {
           return Text("User: ${user.email}");
         }
         return null;
       },
     );
   }
   return null;
 },
),
```

Example 3 – The way that won’t work...

Approach:

1. We monitor pUserService and pUserService.value?.pUser.
2. When the value of pUserService becomes non-null, the PodListBuilder will rebuild its child.
3. Since we’re monitoring pUserService.value?.pUser, and it’s initially null, we’re actually not monitoring pUserService.value?.pUser. We’re monitoring null.
4. This means the user will always be null even after the UserService started.

```
final pUserService = Pod<UserService?>(null);
final userService = await UserService.create();
pUserService.set(userService);

PodListBuilder(
 podList: [
   pUserService,
   pUserService.value?.pUser,
 ],
 builder: (context, child, values) {
   final user = values.second as UserModel?;
   if (user != null) {
     return Text("User: ${user.email}");
   }
   return null;
 },
),
```

Example 3 – The best way...

Approach:

1. We create a function that returns a list of Pods. We call this function the responder.
2. We use a ResponsivePodListBuilder instead.
3. Whenever any of the pods initially returned by the responder change, the ResponsivePodListBuilder triggers the responder once more, generating an updated list of Pods for observation.
4. The ResponsivePodListBuilder operates efficiently without any excess

```
final pUserService = Pod<UserService?>(null);
final userService = await UserService.create();
pUserService.set(userService);
```

```
TPodList userPlr() => [
       pUserService,
       pUserService.value?.pUser,
     ];

ResponsivePodListBuilder(
 podListResponder: userPlr,
 builder: (context, child, values) {
   final user = values.second as UserModel?;
   if (user != null) {
     return Text("User: ${user.email}");
   }
   return null;
 },
),
```

Conclusion

Embrace MSSM with Pods for your next Flutter project to harness the full potential of modular services state management, improving your app's architecture and user experience!

- **Modular Approach:** MSSM simplifies state management by dividing application state into distinct, modular services, enhancing maintainability and scalability.
- **Service-Oriented Architecture:** Each service operates like a background process, managing a specific domain of application data and improving overall application performance through efficient data handling and processing.
- **Data Nodes and Flutter Widgets:** Utilizing ValueNotifiers and Pod structures for data nodes facilitates responsive and efficient UI updates, with Pods offering advanced features for state management.
- **Pod Advantages:** Pods provide significant benefits over traditional methods, including support for complex data types, efficient management of state updates, and streamlined codebase through PodListBuilder and ResponsivePodListBuilder.
- **Practical Implementation:** Demonstrated through Flutter app examples, incorporating Pods significantly simplifies state management, enabling more readable, maintainable, and performant applications.