Dependency Injection

What is a Dependency?

- a dependency is an object or a class that another class needs to function correctly.
- For instance, if we have a class that fetches data from a website, this class may depend on a service or a repository to access the data. These services or repositories are the dependencies of our class.
  In the context of Flutter, dependencies could be Flutter packages, third-party dependencies, or even classes we create within our own Flutter projects.

What is a Dependency Injection?

- Dependency injection is a programming technique that makes our code more maintainable by decoupling the dependencies of a class.

The primary goal of dependency injection is to provide a class with its dependencies, rather than having the class create these dependencies itself.
This way, we can manage dependencies in a more maintainable way, making our code easier to test and modify.
In Flutter, we implement dependency injection by passing instances of dependencies into the class that needs them.
DI Could be done through the**:**

- **constructor (constructor injection),**
- **a method (method injection),**
- **or directly into a field (field injection)**.

Why Use Dependency Injection in Flutter?

- Simplified Maintenance (Loose Coupling):
  - Without DI (Tight Coupling):
    - **Challenge:** Modifying UserService requires changing how it interacts with LocalStorage everywhere it's used. This can lead to ripple effects and make maintenance difficult.

Code Example WithOut DI:

```
class LocalStorage {
  Future<String> read(String key) async {
    // Simulate reading data from a local file (replace with actual
implementation)
    final data = await Future.delayed(Duration(milliseconds: 200), () =>
'John Doe');
    return data;
  }
}

// UserService tightly coupled to LocalStorage
class UserService {
  final LocalStorage storage;

  UserService(this.storage);

  Future<String> getUserName() async {
    final data = await storage.read('user_name');
    return data ?? '';
  }
}

// Usage
void main() async {
  final storage = LocalStorage();
  final userService = UserService(storage);

  final userName = await userService.getUserName();
  print('User Name: $userName'); // Output: User Name: John Doe (after
200ms delay)
```

```
}
```

- With DI (Loose Coupling):
  - **Benefit: By** injecting the DataStorage dependency through the interface, UserService becomes loosely coupled. You can now swap LocalStorage with another implementation (e.g., RemoteStorage) without modifying UserService. This improves code maintainability and flexibility.

Code example With DI:

```
// DataStorage Interface (defines the contract)
abstract class DataStorage {
  Future<String> read(String key);
}

// LocalStorage Implementation (concrete example)
class LocalStorage implements DataStorage {
  @override
  Future<String> read(String key) async {
    // Simulate reading from a local storage (replace with your actual
implementation)
    await Future.delayed(const Duration(milliseconds: 500)); // Simulate
delay
    return 'Alice'; // Example data
  }
}

// MockDataStorage for Unit Testing (simulates behavior)
class MockDataStorage implements DataStorage {
  @override
  Future<String> read(String key) async {
    // Simulate data storage behavior for testing
    return 'Test User'; // Example test data
  }
}

// UserService (depends on the DataStorage interface)
class UserService {
  final DataStorage storage;

  UserService(this.storage);

  Future<String> getUserName() async {
```

```
    final data = await storage.read('user_name');
    return data ?? '';
  }
}

// Usage (Real and Unit Test)
void main() async {
  // Real usage (injecting LocalStorage)
  final storage = LocalStorage();
  final userService = UserService(storage);
  final userName = await userService.getUserName();
  print(userName); // Output: Alice (assuming LocalStorage)

  // Unit test (injecting MockDataStorage)
  final mockStorage = MockDataStorage();
  final userService = UserService(mockStorage);
  final userName = await userService.getUserName();
  expect(userName, 'Test User'); // Verify test data (using a testing
framework like test)
}
```

- Effortless Unit Testing:
  - Without DI
    - **Challenge**: Testing code that relies on external resources (like databases or APIs) can be cumbersome. You might need to set up complex mockups or manipulate real data.
      - Code Example Without DI

```
// NetworkService (Tightly Coupled to HttpClient)
class NetworkService {
  final HttpClient client; // Tightly coupled

  NetworkService(this.client);

  Future<User> fetchUser(int userId) async {
    final response = await
client.get(Uri.parse('https://api.example.com/users/$userId'));
    return User.fromJson(jsonDecode(response.body));
  }
```

```
}

// Unit Test (Complex Setup)
test('NetworkService fetches user correctly', () async {
  // Complex setup to mock HttpClient behavior
  final mockHttpClient = MockHttpClient();
  // ... configure mock behavior

  final networkService = NetworkService(mockHttpClient);
  final user = await networkService.fetchUser(1);

  expect(user.id, 1);
  expect(user.name, 'Alice');
});
```

- With DI
  - **DI Solution**: DI allows you to inject mock or test doubles (simulations) of dependencies during unit testing. This isolates your code's logic and simplifies testing.
  - Example code:

```
// HttpService Interface (defines the contract)
abstract class HttpService {
  Future<Response> get(Uri url);
}

// HttpClientAdapter (Concrete Implementation)
class HttpClientAdapter implements HttpService {
  final HttpClient client;

  HttpClientAdapter(this.client);

  @override
  Future<Response> get(Uri url) async {
    return await client.get(url);
  }
}

// MockHttpService (Mock Implementation for Testing)
class MockHttpService implements HttpService {
```

```
  @override
  Future<Response> get(Uri url) async {
    // Simulate successful response with example data
    return Response(200, jsonEncode({'id': 1, 'name': 'Alice'}));
  }
}

// NetworkService (Depends on HttpService Interface)
class NetworkService {
  final HttpService client;

  NetworkService(this.client);

  Future<User> fetchUser(int userId) async {
    final response = await
client.get(Uri.parse('https://api.example.com/users/$userId'));
    return User.fromJson(jsonDecode(response.body));
  }
}

// Unit Test (Simple and Isolated)
test('NetworkService fetches user correctly', () async {
  // Inject MockHttpService for testing
  final mockService = MockHttpService();
  final networkService = NetworkService(mockService);

  final user = await networkService.fetchUser(1);

  expect(user.id, 1);
  expect(user.name, 'Alice');
});
```

In short, DI helps you write well-structured, maintainable, and testable Flutter apps.

Different Types of Dependency Injection in Flutter

Three types: Constructor injection, method injection and field injection.

1. Constructor Injection

- Constructor injection is one of the most common forms of dependency injection. In this approach, we pass the dependencies of a class through its constructor. This is a straightforward and effective way to provide a class with its dependencies, and it's often the preferred method for implementing dependency injection in Flutter.

Code Example:

```
class UserRepository {
  // Example dependency
  final ApiClient apiClient;

  // Constructor with dependency injection
  UserRepository(this.apiClient);

  // Methods that use apiClient...
}

class UserBloc {
  final UserRepository userRepository;

  // Constructor with dependency injection
  UserBloc(this.userRepository);

  // Methods that use userRepository...
}

void main() {
  // Create dependencies
  final apiClient = ApiClient();
  final userRepository = UserRepository(apiClient);

  // Create object with dependencies injected
  final userBloc = UserBloc(userRepository);

  // Now you can use userBloc...
```

```
}
```

2. Method Injection

- Dependencies are injected using setter methods. This can be useful in some scenarios, but it can be less transparent than constructor injection.

Code Example:

```
class UserRepository {
  // Example dependency
  final ApiClient apiClient;

  // Constructor with dependency injection
  UserRepository(this.apiClient);

  // Methods that use apiClient...
}

class UserBloc {
  UserRepository _userRepository;

  // Method to set dependency
  void setUserRepository(UserRepository userRepository) {
    _userRepository = userRepository;
  }

  // Methods that use _userRepository...
}

void main() {
  // Create dependencies
  final apiClient = ApiClient();
  final userRepository = UserRepository(apiClient);
  final userBloc = UserBloc();

  // Set dependency using method injection
  userBloc.setUserRepository(userRepository);

  // Now you can use userBloc...
}
```

3. Field Injection

- Field Injection where we inject a dependency directly into a field of a class. This can be useful in certain scenarios, but it's generally less preferred than constructor or method injection because it can make our code harder to understand and test.

Code Example:

```
class UserRepository {
  // Example dependency
  final ApiClient apiClient;

  // Constructor with dependency injection
  UserRepository(this.apiClient);

  // Methods that use apiClient...
}

class UserBloc {
  // Field to hold dependency
  late UserRepository userRepository;

  // Methods that use userRepository...
}

void main() {
  // Create dependencies
  final apiClient = ApiClient();
  final userRepository = UserRepository(apiClient);
  final userBloc = UserBloc();

  // Assign dependency directly to field
  userBloc.userRepository = userRepository;

  // Now you can use userBloc...
}
```

Comparison and Use Cases for Each Type DI

Each type of dependency injection has its own use cases and advantages.

Constructor injection is generally the most preferred method because it clearly indicates the dependencies of a class and ensures that a class always has access to its dependencies once it's created.

Method injection can be useful when a class needs to use different implementations of a dependency at different times, or when a dependency is not needed immediately when the class is created.

Field injection is less common and generally less preferred because it can make our code harder to understand and test. However, it can be useful in certain scenarios, such as when we need to inject dependencies into Flutter widgets, which don't have a traditional constructor.

Dependency Injection Packages in Flutter

Some of the most popular dependency injection packages in Flutter include Provider, GetIt, and Riverpod.

These packages provide different ways to manage dependencies in our Flutter projects, and each has its own strengths and use cases.

Let’s Take detailed Look into GetIt

OverView

GetIt is a service locator for Dart and Flutter projects. It provides a simple and effective way to manage singleton instances and implement dependency injection.

With GetIt, we can register our dependencies as singletons and easily access them anywhere in our code. GetIt also supports asynchronous initialization and factory registrations, which can be useful for managing more complex dependencies.

For Example:

Problem: Imagine you're building a Flutter app with many screens. Each screen needs to get data from the internet using a special class called ApiService. The problem is, if you have to create a new ApiService object every time a screen needs data, your code gets messy and repetitive. This can also lead to mistakes if you forget to set things up the same way each time.

```
// This creates a new ApiService for each screen, causing redundancy
Screen1: ApiService apiService = ApiService();
data = apiService.getData();

Screen2: ApiService apiService = ApiService(); // Duplicate creation
data = apiService.getData();
```

The Solution: There's a handy package called GetIt that can help us solve this problem. GetIt acts like a central storage for important things our app needs, like the ApiService. We can set it up once in the main part of our app (the main() function) to store a single ApiService object. This way, any screen in our app can easily access this same object whenever it needs data.

```
// Configure GetIt in main() (once)
GetIt.instance.registerSingleton<ApiService>(ApiService());

// Access the single ApiService instance in any screen
final data = GetIt.instance<ApiService>().getData();
```

Let’s Dive into Code example For DI Using GetIt.

Using get_it for Setter Injection and Constructor Injection as **field injection is not supported** in this package.

Get_it can be used for both **method injection** and **constructor injection**. Method injection is a type of dependency injection where the dependencies are provided through setter methods. In contrast, constructor injection is a type of dependency injection where the dependencies are provided through a class constructor.

1. Constructor injection

Code example:

```dart
import 'package:get_it/get_it.dart';

final getIt = GetIt.instance;

// Interface for a Network Client
abstract class NetworkClient {
  Future<String> fetchDataFromUrl(String url);
}

// Concrete implementation using a real HTTP client (replace with your
actual client)
class HttpNetworkClient implements NetworkClient {
  @override
  Future<String> fetchDataFromUrl(String url) async {
    // Implement your logic to fetch data from the URL using a real HTTP
client
    return Future.value("Fetched data from $url (replace with actual
network call)");
  }
}

// Service class that requires network data
class MyService {
  final NetworkClient networkClient;

  // Constructor Injection
  MyService(this.networkClient);

  Future<String> retrieveDataFromNetwork(String url) async {
    final data = await networkClient.fetchDataFromUrl(url);
    return "Retrieved data from network: $data";
  }
}

void main() async {
  // Configure dependencies
  getIt.registerSingleton<NetworkClient>(HttpNetworkClient());
```

```
  // Usage with Constructor Injection
  final myService = MyService(getIt<NetworkClient>());
  final result = await
myService.retrieveDataFromNetwork("https://example.com/data");
  print(result); // Output: Retrieved data from network: Fetched data from
https://example.com/data (replace with actual network call)
}
```

2. Method injection

Code Example:

```
import 'package:get_it/get_it.dart';

final getIt = GetIt.instance;

// Interface for a Network Client
abstract class NetworkClient {
  Future<String> fetchDataFromUrl(String url);
}

// Concrete implementation using a real HTTP client (replace with your
actual client)
class HttpNetworkClient implements NetworkClient {
  @override
  Future<String> fetchDataFromUrl(String url) async {
    // Implement your logic to fetch data from the URL using a real HTTP
client
    return Future.value("Fetched data from $url (replace with actual
network call)");
  }
}

// Service class that requires network data
class MyService {
  NetworkClient? _networkClient;

  void setNetworkClient(NetworkClient networkClient) {
```

```
    _networkClient = networkClient;
  }

  Future<String> retrieveDataFromNetwork(String url) async {
    if (_networkClient == null) {
      throw Exception("Network client not injected!"); // Handle missing
dependency
    }
    final data = await _networkClient!.fetchDataFromUrl(url);
    return "Retrieved data from network: $data";
  }
}

void main() async {
 getIt.registerSingleton<NetworkClient>(HttpNetworkClient());

  // Usage with Setter Injection
  final myService = MyService();
 // Comment this line for not set network client
  myService.setNetworkClient(getIt<NetworkClient>());
  final result = await
myService.retrieveDataFromNetwork("https://example.com/data");
  print(result); // This might throw an exception if the dependency is not
set
}
```

Registering a New Instance and the Same Instance

With get_it, you can register a new or the same class instance. Registering a new instance means that every time you call get_it(), you will get a new instance of Service. Registering the same instance means you will get the same service instance every time you call get_it().

Code reference below:

```
void setup() {
  getIt.registerFactory<Service>(() => ServiceImpl()); // New instance
  getIt.registerSingleton<Service>(ServiceImpl()); // Same instance
}
```

In the above example,

we register ServiceImpl as a **factory**, meaning a new instance will be created every time we call get_it().

We also register ServiceImpl as a **singleton**, meaning the same instance will be returned whenever we call get_it().

Different Registration Types

Get_it offers different registration types such as lazy singletons, factory functions, and constructor injection.

A lazy singleton is a singleton that is only created when it is first requested.

A factory function is a function that returns a new instance of a class.

Constructor injection is a form of dependency injection where dependencies are provided through a class constructor.

For example:

We have one login screen where field validations are implemented. When the user attempts to login without entering any information, the validator is triggered through singleton registration. Upon successful validation, the LoginService is invoked to authenticate the user credentials, through lazy singleton registration. Once authenticated, the NetworkCallService is called to make API calls, through factory registration, for getting user data.

Code reference:

```
// Lazy Singleton (created on first use)
GetIt.I.registerLazySingleton<LoginService>(() => LoginService());

// Factory (creates new instance each time)
GetIt.instance
    .registerFactory<NetworkCallService>(() => NetworkCallService());
// Singleton (single instance throughout the app)
GetIt.instance.registerSingleton<Validator>(Validator());
```