Separating Client Apps from the Services They Consume During Development

Client Apps typically rely on remote services to power their apps. However, development schedules between the client app and the services don't always fully align. For a high velocity inner dev loop, client app development must be decoupled from the backend services while still allowing the app to "invoke" the services for local testing.

Options

Several options exist to decouple client app development from the backend services. The options range from embedding mock implementation of the services into the application, others rely on simplified versions of the services.

This document lists several options and discusses trade-offs.

Embedded Mocks

An embedded mock solution includes classes that implement the service interfaces locally. Interfaces and data classes, also called models or data transfer objects or DTOs, are often generated from the services' API specs using tools like nswag (RicoSuter/NSwag: The Swagger/OpenAPI toolchain for .NET, ASP.NET Core and TypeScript. (github.com)) or autorest

(Azure/autorest: OpenAPI (f.k.a Swagger) Specification code generator. Supports C#, PowerShell, Go, Java, Node.js, TypeScript, Python, Ruby (github.com)).

A simple service implementation can return a static response. For RESTful services, the JSON responses for the stubs can be stored as application resources or simply as static strings.

```
public Task<UserProfile> GetUserAsync(long userId,
CancellationToken cancellationToken)
{
    PetProfile result =
Newtonsoft.Json.JsonConvert.DeserializeObject<UserProfile>(
        MockUserProfile.UserProfile, new
Newtonsoft.Json.JsonSerializerSettings());

    return Task.FromResult(result);
}
```

More sophisticated can randomly return errors to test the app's resiliency code paths.

Mocks can be activated via conditional compilation or dynamically via app configuration. In either case, it is recommended to ensure that mocks, service responses and externalized configurations are not included in the final release to avoid confusing behavior and inclusion of potential vulnerabilities.

Sample: Registering Mocks via Dependency Injection

Dependency Injection Containers like Unity (Unity Container Introduction | Unity Container) make it easy to switch between mock services and real service client implementations. Since both implement the same interface, implementations can be registered with the Unity container.

```
public static void Bootstrap(IUnityContainer container)
{

#if DEBUG
    container.RegisterSingleton<IUserServiceClient,
MockUserService>();
#else
    container.RegisterSingleton<IUserServiceClient,
UserServiceClient>();
#endif

}
```

Consuming Mocks via Dependency Injection

The code consuming the interfaces will not notice the difference.

```
public class UserPageModel
{
    private readonly IUserServiceClient userServiceClient;

    public UserPageModel(IUserServiceClient userServiceClient)
    {
        this.userServiceClient = userServiceClient;
    }

    // ...
}
```

Local Services

The approach with Locally Running Services is to replace the call in the client from pointing to the actual endpoint (whether dev, QA, prod, etc.) to a local endpoint.

This approach also enables injecting traffic capture and shaping proxies like

Postman (Postman API Platform | Sign Up for Free) or Fiddler (Fiddler | Web Debugging Proxy and Troubleshooting Solutions (telerik.com)).

The advantage of this approach is that the APIs are decoupled from the client and can be independently updated/modified (e.g. changing response codes, changing data) without requiring changes to the client. This helps to unlock new development scenarios and provides flexibility during the development phase.

The challenge with this approach is that it does require setup, configuration, and running of the services locally. There are tools that help to simplify that process (e.g. JsonServer, Postman Mock Server).

High-Fidelity Local Services

A local service stub implements the expected APIs. Just like the embedded mock, it can be generated based on existing API contracts (e.g. OpenAPI).

A high-fidelity approach packages the real services together with simplified data in docker containers that can be run locally using docker-compose before the client app is started for local debugging and testing. To enable running services fully local the "local version" substitutes dependent cloud services with local alternatives, e.g. file storage instead of blobs, locally running SQL Server instead of SQL AzureDB.

This approach also enables full fidelity integration testing without spinning up distributed deployments.

Stub / Fake Services

Lower fidelity approaches run stub services, that could be generated from API specs, or run fake servers like JsonServer (JsonServer.io: A fake json server API Service for prototyping and testing.) or Postman. All these

services would respond with predetermined and configured JSON messages.

How to Decide

	Pros	**Cons**	**Example when developing for:**	**Ex W to**
Embedded Mocks	Simplifies the F5 developer experience	Tightly coupled with Client	More static type data scenarios	Te (e te int te
	No external dependencies to manage	Hard coded data	Initial integration with services	
		Mocking via Dependency Injection can be a non-trivial effort		
High-Fidelity Local Services	Loosely Coupled from Client	Extra tooling required i.e. local infrastructure overhead	URL Routes	W cc ar av

	Easier to independently modify response	Extra setup and configuration of services		
	Independent updates to services			
	Can utilize HTTP traffic			
	Easier to replace with real services at a later time			
Stub/Fake Services	Loosely coupled from client	Extra tooling required i.e. local infrastructure overhead	Response Codes	W Cc av
	Easier to independently modify response	Extra setup and configuration of services	Complex/variable data scenarios	W Cc ar av
	Independent updates to services	Might not provide full fidelity of		

	expected API		
	Can utilize HTTP traffic		
	Easier to replace with real services at a later time		

Last update: August 26, 2024