Backend for Frontend

You are designing an application using a Microservices Architecture. You realize that the Domain Microservices that encapsulate individual business functions do not map cleanly to the channel-specific needs of your client applications. All of the server-side functionality a client application needs should be accessible through a single API.

How do you represent a channel-specific service interface that is consistent with an overall microservices architecture but allows enough uniqueness that it can be adapted to the needs of a specific client type?

There are several issues that can make this implementation challenging:

- Microservices break one set of functionality into multiple APIs.
- The easiest way for a client to interface with a server is through a single API.
- Different types of clients—browser, mobile, CLI—want different APIs for the same functionality.
- Business logic is rarely specific to a single client type. It should be implemented such that all client types can share it.

Therefore,

Build a "Backend for Frontend" (e.g. a *BFF* or Dispatcher) that acts as a single API for a client. Implement different BFFs for different types of clients, each with an API customized to what that client type needs.

A *BFF* can perform the following actions:

- **Orchestration** – It can orchestrate several calls to business microservices that result from a single client action.
- **Translation** – It can translate the results of a microservice into a channel-specific representation that more cleanly maps to needs of the user experience of that client.
- **Filtering** – It can filter results from a business microservice that are not needed by a particular client type.

A BFF should not contain any business logic. Because it's specific to a single client type, any business logic implemented in BFFs won't be shared across all client types.

This pattern was introduced in Newman and is a key part of building microservices architectures that have dynamic front-ends following a Multichannel Architecture such as Native Mobile Applications or Single Page Applications. Both of these patterns introduce unique translation or filtering requirements that often necessitate the use of a Backend for Frontend. For instance, in a native application you may need to filter long data sets in order to fit on the limited screen real estate of the mobile application. Likewise a Single Page Application may require orchestration of several calls to individual Domain Microservices to return a particular set of information needed to represent a screen flow.

An important point about implementing the Backend for Frontend pattern is that in most cases, it is the same team that is responsible for building both the client application and the Backend for Frontend. This leads to some particular development efficiencies; the first is that it is often convenient to use the same programming language for both. Thus, many times JavaScript developers building a Single Page Application will also want to develop their Backend for Frontend services using JavaScript and Node on the server side. Likewise, Java developers building an Android native application may want to develop their Backend for Frontend services with Java.

Backend for Frontend's often use Page Caches to store long results obtained from Domain Microservices so that the results can be obtained a page at a time from a client. A Service Registry may help the client code that makes up the bulk of a *Backend for Frontend* to be resilient in the face of changes to the physical address of the *Business Microservices* that it depends on.

Sam Newman introduced this pattern in Building Microservices