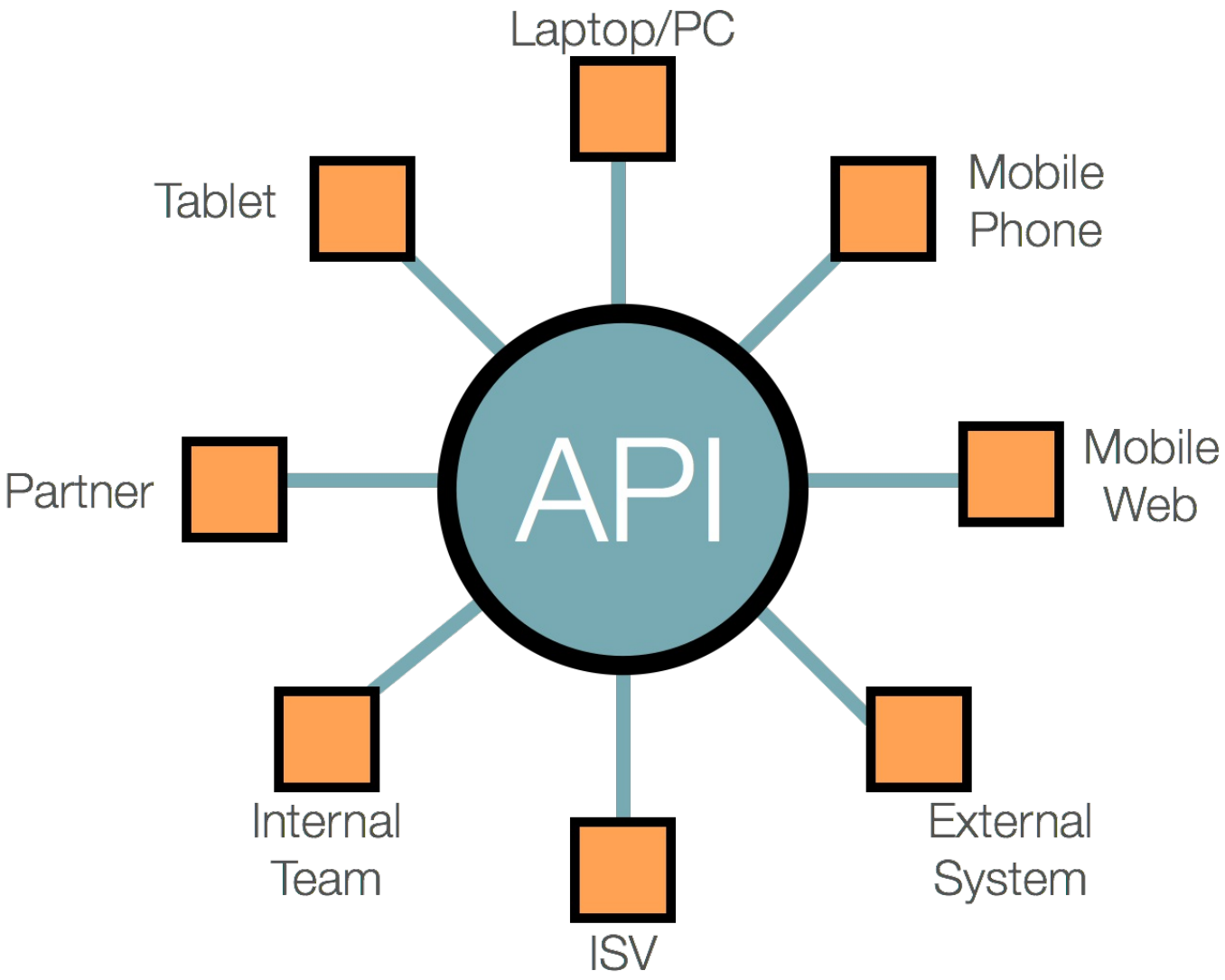
API

Application Programming Interfaces (API) describe the programmatic interfaces used for interacting with a software system. APIs can exist on many levels, from methods to classes to online services. APIs may be used by user interface code, by backend services communicating with each other, or by third parties. The primary purposes of APIs are to provide a predictable means of programmatic interaction and to hide implementation details from the API client. By hiding the internal implementation, APIs enable the API owner to change their underlying implementation without affecting any clients that may depend upon it.

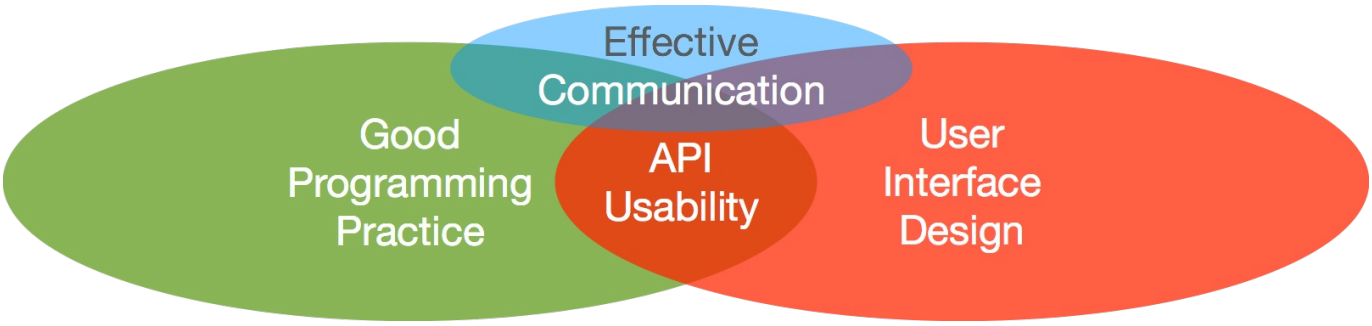


While APIs have been defined and used for decades, the advent of web-

based services has propelled API design into the the forefront of many organizations (typically through REST-based interfaces). This is because services can be hosted behind APIs giving API owners an easy mechanism for maintaining and updating their systems and controlling billing without having to release the actual source code for their systems.

One consequence of providing a public API though is that once clients start to use them they are difficult (or impossible) to change, leading to the saying "API are forever". Because of their long lifetimes, it is important that software engineers carefully design their interfaces to make them easy to understand, use, maintain, secure, and extend.

API design is an iterative process and good design requires collecting a variety of viewpoints (both internal to your organization and externally from engineers who will have to understand and use your API). There are typically many different design options for any given API, choosing the 'right' one will require balancing some of the concerns mentioned in the last paragraph.



One aspect of APIs that is often overlooked is their *usability*. The user of an API is the developer who has to learn to use the API to use it within their own system. By providing effective affordances (such as meaningful names, parameter types, and data formats) and thinking about how the APIs will be used in practice, API designers can ease adoption of their APIs and help prevent consumers from using their APIs incorrectly. Useable API design requires combining strong development practice with techniques from user interface design (such as prototyping and user

studies) and effective documentation.

API platforms

API Platforms are now an extremely popular way of building business value. These platforms emerged in the early 2000s when companies realized they could focus their development efforts on building a core set of functionality that could be extended by customers or third party organizations. By focusing on maximizing the quality and core capabilities of their platform, companies relieved themselves from building all possible niche versions of their product, instead letting other developers extend the platform for them.

API design is of critical importance for companies building API platforms as a set of APIs that is opaque and difficult to understand will not achieve wide adoption. This is especially important for organizations who intend their API to be their primary product.

Design principles

At a high-level the principles behind good API design mirror that of good design in general. The first two questions to consider when designing the API are:

1. What is the goal of the API?
2. Who is the API's intended consumer?

These questions will help drive more technical answers like which languages, protocols, and data formats should be used, along with how the API will be versioned, licensed, and authenticated.

If APIs had one design tenet it would be:

> APIs should be easy to use and hard to misuse.

APIs should not be designed in a vacuum. Starting with a one-page spec that describes the API goals and users is a great place to start. Next, as many stakeholders as possible should be looped into the design to ensure it makes sense and adequately addresses their needs. While automated testing plays an important role in API validation, it is also important to create several proof-of-concept client systems that use the APIs to ensure it can be effectively used. While all APIs will naturally have some fundamental shortcomings (an API cannot be all things to all users), by looking at the client code you can gain an appreciation for how code may be duplicated, awkward, or erroneous when used by API clients.

Some high-level API design advice includes:

- APIs should do one thing, and do it well (also known as the single responsibility principle).
- APIs should be kept as small and simple as possible; once an API is released it is hard to remove.
- APIs should *never* expose internal implementation details; these make it difficult for the consumer to use and for the maintainer when they want to change an internal implementation detail.
- The 'usability' of a system to another engineer is dictated by the quality of the API. This means names should be well considered and consistent and the APIs, their constraints, and their pre- and post-conditions should be documented. Consistent APIs reduce developer surprise and helps them to naturally move from one API to

the next.

References

- We will expand on design in general later in the course. Joshua Bloch, the designer of the architect of the `java.util` package has a great set of slides on how to design good APIs online.
- Many popular libraries have great APIs. While the JDK and Android are huge, smaller libraries like Guava and Joda-Time are also instructive.
