Best practices for IoT development

Applying lessons from mobile development to your IoT projects

Andrew Fisher

September 02, 2015

When developing in the Internet of Things (IoT) space, you must consider connectivity, security, and privacy. And, given the nature of always-connected devices, you must also consider power consumption, chip speed, memory, and firmware updates. All of these critical factors determine whether your IoT product might be successful or not. This article examines good mobile development patterns and how you can apply them to IoT development to help make your IoT "thing" more likely to succeed.

So you've decided to develop an IoT product. As you get started, you can learn from many practices that have evolved out of Mobile Development. In particular, consider these practices:

- De-couple APIs from services
- Iteratively prototype your solution
- Expect connectivity issues

Avatars, services, and de-coupled APIs

If you have created any type of web application, then you're already familiar with the idea of a service. A service is the capability that is provided by the application and is our traditional view of a system, for example, a weather data service.

You can learn more about avatars in this W3C position paper, "Leveraging cyber-physical objects through the concept of avatars" by Medini, et al.

Consuming this service are *avatars*, which are Internet-connected "things" (that can be software or physical smart things). These avatars then interact with parts or the whole of this service. Each avatar interacts with the service independently, and yet collectively they make the service powerful.

Let's consider the weather data service example. Avatars might be a consumer website or mobile app that shows current weather readings. Avatars might also be a connected weather station that reports data into the weather service every minute, sharing the current temperature, humidity, and wind speed.

Because the current generation of mobile services are so powerful (Google and particularly Google Now, Facebook, Twitter, among other consumer applications), the services are accessible

in a variety of ways, such as desktop applications, mobile websites, specific apps, third-party apps, and even browser plug-ins. Each of these avatars can use as much or as little of the service as required in order to fulfill the contextual niche that they operate in. Thus, it's possible to embed a "tweet this link" button (one service) on any web page (an avatar) or see a list of search results (another service) from your home screen on Android (an avatar).

The key facilitator of this approach is designing the service to have accessible APIs that each avatar can interact with effectively and contextually.

A great modern example of this approach in the hardware space is FitBit. FitBit has a line of products that all serve to make health and fitness more measurable. While differences exist between certain products (step trackers versus stair trackers for example), the core service is the same. These products aggregate data from the sensors around activity and provide that aggregated data back to the user to be able to make informed decisions around their health.

From a services perspective, FitBit provides the following services:

- Track activity data (type, duration, calorie expenditure, time it occurred, heart rate)
- Track sleep data (sleep versus wakefulness duration)
- Track calorie and water intake
- Track weight data
- Report on data in the system

From an avatar perspective, different avatars behave in different ways for a FitBit device:

- Physical fitness trackers record information and pass it back to the service
- A mobile app provides the basic view of data as it is aggregated
- A web application provides more detailed and historical trend views
- Third-party systems can push data into the platform (like food consumption)
- Third-party systems can read aggregate data (like steps) to add to their own services

As you can see, the decoupling of the service and the production of a good API provides these benefits to FitBit:

- The service can be expressed through different avatars and therefore provide value at every point (on your wrist, in your pocket, on your desktop)
- Third-party systems can add extra value to FitBit with their own services.
- As fragmentation occurred in the mobile landscape (as more companies competed with Apple), FitBit can respond quickly and easily with apps targeted to any device that needed an avatar (even Windows phones).

I recommend that you read Mike Kuniavsky's book titled Smart Things if you are interested in learning more about avatars and services.

This de-coupling approach is increasingly common because applications must cater to both mobile and desktop web experiences. If you set up your IoT service in a similar way, you can take it in new directions as market opportunities present themselves.

IoT products require software and hardware prototypes

Anyone who builds physical products knows that you start with very simple prototypes and keep refining them based on feedback and their performance in the real world. IoT products have the additional complexity that the software and network elements need to be prototyped as well.

This challenge is magnified when you consider the numerous contexts in which your IoT product might be used. It might just be a sensor, but how do you interact with it? Is it by using a mobile or web application? Is configuration different than reporting? How usable are these interfaces? The list of questions can seem endless.

Mobile products have a long history of dealing with this challenge due to the nature of how mobile environments can cause context changes that affect the usability of a service. As a result, mobile development typically proceeds with basic prototypes that offer more and more fidelity as they get refined. Usually the process involves these phases:

1. Create a simple interactive web application that uses a development framework to rapidly wireframe the core aspects of the interaction. Stub out all of the service calls to placeholder content that is real enough to simulate how the information needs to be rendered and interacted with.
2. After it is refined, start integrating simple aspects of the service so that you can determine whether the responses are appropriate and contextually relevant.
3. Start designing interfaces beyond functional prototypes, and be sure to consider interaction methods and requirements for guidance and feedback
4. Continue refining and integrating capabilities until you release the product

By iteratively prototyping your IoT product, you can be sure that it is usable and the right aspects of your service are being incorporated. Note that not every context requires every aspect of your service. Look at a modern mobile banking application and you'll see that beyond account balance and some payment/transfer interactions, you don't typically have the full banking platform at your disposal. In the context of mobile banking usage, users typically just check whether they have enough money to go out for a meal or pay a bill rapidly.

In addition to prototyping your IoT service and avatars, you can also iteratively prototype the hardware that you use in your product:

1. Start with simple, off-the-shelf hardware. Although everyone thinks the Arduino is a hobbyist platform, it is cheap, produced well, and is reliable enough to rapidly produce something you can interact with. For a few dollars, you can determine whether your idea has any merit without investing in custom system board design.
2. Refine the prototype by using as many off-the-shelf components as possible. Keep it at a scale where it can be rapidly torn apart and rebuilt. Prototyping system boards, through hole components, and pluggable modules are great for this. Use components that are extremely well known, so you can prototype rather than get lost in implementation detail.
3. Decouple your critical components from the system boards that they connect with. Using components that use standard protocols, such as I2C and SPI, means that you can rapidly switch out your Arduino for a BeagleBone or a Raspberry Pi.

4. Rely on off-the-shelf hardware for as long as possible, before you decide to switch, and then think about it again.

It's worth noting that you can use a Raspberry Pi, Arduino, ESP8266, or similar component as the core of your IoT product. Yes, there are good reasons to use specific chips or system boards when you need to. But, if you're making an Internet connected garden sensor, an ATMEGA328 (the core of the Arduino) or an ESP8266 will give you plenty of headroom. By using off-the-shelf components, you can get the benefit of scale that already exists and the benefit of basic problems already being solved.

Hardware these days is easily reverse engineered. In an IoT product, the value is in the service, not the individual avatars. As such, expensive, custom-designed hardware components are going to raise the cost of your avatars without increasing the value of your service meaningfully (and your product price can be undercut). Keep your components and your controller board modular. Then, if you do need to move to a more powerful or less expensive system board, you don't have to rebuild everything from scratch again.

A final point on prototyping is to ensure that you're logging usage and error details. This logging helps you understand what is working and not working within the system and also to debug things more rapidly, which are critical requirements when you're in the prototyping phase. This logging can be maintained all the way to production, noting the obvious privacy concerns that end consumers will have with this logging and which you'll need to address appropriately.

Offline First designs can address connectivity issues

To learn more about offline-first apps, read the article, "Designing Offline-First Web Apps" by Alex Feyerke.

Many engineers who live in large cities, particularly Western cities, take the ubiquity of mobile and wifi networks for granted. This mistaken assumption can render your product entirely unusable. Modern mobile and web development have a practice that is called "Offline First," which is designed to build applications that will function as well as possible without the network to back them up. All features might not be available, but the loss and restoration of connectivity needs to be a seamless process that doesn't touch the end consumer.

From a mobile perspective, these general principles are used to facilitate an offline first design:

- Assume that the network can disappear at any moment - even (and especially) midway through a transfer.
- Use smaller messages, more frequently, rather than single, large requests and responses.
- Use local storage for buffering messages that need to go across the network. Messages that are stored locally can be queued when offline or can be sent again if a failure occurs.
- Ensure that the application assesses connectivity, and don't expect the user to be aware of it.
- Provide mechanisms for data synchronization behind the scenes, such as making system state updates first and then synchronizing versioned data later (so the user can get on with their task).

From an IoT perspective, many of these things hold true, however many sensor devices assume permanent connectivity to wifi. Of course, if you are in a highly congested piece of airspace (such as a residential building in the center of a large city such as Hong Kong), it is possible to have 50 - 100 wifi networks all in contention for the same signal space (and interfering with each other). Anyone who has attended a developer conference knows how appalling wifi is when even 200 developers with notebook computers, tablets, and mobile phones all create hotspots. I've seen perfectly good wireless modules refuse to associate at conferences due to noise and contention.

So what can you take away from these practices from an IoT perspective? Consider these design points:

CoAP is the "Constrained Application Protocol" and is designed as a web transfer protocol in networks in the Internet of Things. You can read more about CoAP in the IETF proposed standard, RFC7252.

To learn more about MQTT, try out this developerWorks tutorial, "Explore MQTT and the Internet of Things service on IBM Cloud."

- Assume network failure at any point.
- Write locally first, and then send the data.
- On the service side, use message queuing systems such as RabbitMQ to process your messages.
- Use lightweight and resilient protocols for messaging, such as CoAP and MQTT, rather than the heavier HTTP.

Build alerting systems into the service so that you can diagnose when a node falls off the network for a length of time. This alert needs to be informative, but not overwhelming for the user (as it might be that a device's battery failed).

As more devices enter the network space, network reliability cannot be guaranteed, even in home network environments. You must ensure offline capability and network resilience as a critical component of your IoT development.

Conclusion

In this article, I examined how decoupling APIs from services can help create powerful IoT apps. Also, by using simple and modular hardware, such as Arduino boards, you can more easily develop your IoT solution. And finally, by using the mobile concept of offline first, you can more readily ensure the networking success of your IoT device. You now have a good idea of how to apply mobile development best practices to Internet of Things (IoT) development to help make your IoT "thing" more likely to succeed.

Related topics

- Internet of Things on IBM Cloud: Rapidly compose and extend apps that take advantage of data and analytics from your connected devices and sensors.
- Build your own wearable with IBM Watson IoT Platform and IBM Cloud (Mike Spisak and Rhonda Childress, developerWorks, February 2015); Build a hybrid mobile app that connects to a wearable device and sends sensor data from the device to the cloud.
- The Internet of Things service: Get simple but powerful application access to IoT devices and data.