Amazon [Team] | White Paper

Prototype Development Guidelines for Physical Computing Experiences

v0.4

Abstract

In [Team], we operate under a mix of several corporate and operational constraints that often require us to favor speed or fidelity when designing new physical computing experiences. To optimize our process, we are recommending a prototyping architecture standard that contains the following:

1. grouped components set within presentation and logic layers on the client side and a data layer on the server side
2. a set of specific guidelines for each layer and its components
3. a set of general implementation guidelines

This organized yet flexible approach should simplify our prototyping process and empower us to produce high-quality output more efficiently.

Challenges

The charter of [Team] is to create conceptual but testable prototypes that are large, complex, and contain many forms of interaction. Our technologists must research and select hardware and software resources from an evolving array of possibilities, and quickly combine them into a workable solution without sacrificing appearance, durability, or functionality.

Delivering these prototypes against the backdrop of Amazon's design and development culture means working under shifting demands across varied environments. We have several stakeholders and partners for each project, with different interests, needs, and timelines. We may be tasked with creating an experience to demonstrate high concepts, secure executive buy-in, provide a system for user testing, or a mix of these requirements.

Additionally, our physical production environments are not conducive to experimentation. While there are many technologies present in many Amazon [division] facilities, they may not be standardized, and they are often unavailable or inaccessible to prototyping. Moreover, they would rarely be able to drive an exercise based on any new or emergent hardware we are exploring.

The situations in which we work and the technologies we often employ make having an architecture with guidelines that create efficiencies extremely advantageous.

Proposed Solution

Goals

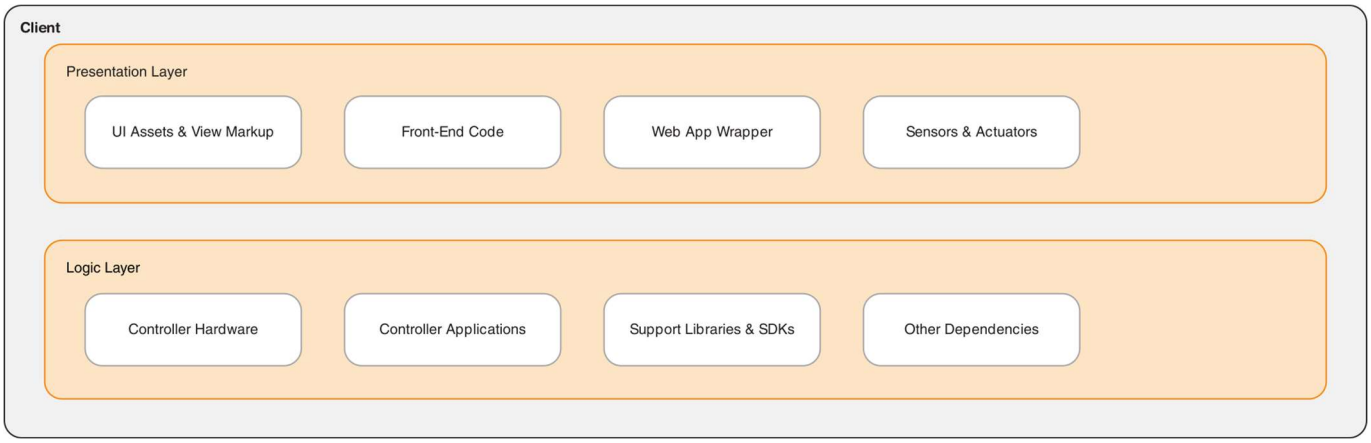
A successful architecture for the construction of physical computing prototypes will create a scenario where we can routinely do the following:

1. Write less code, on average, and write it more easily.
2. Write reusable modules of code.

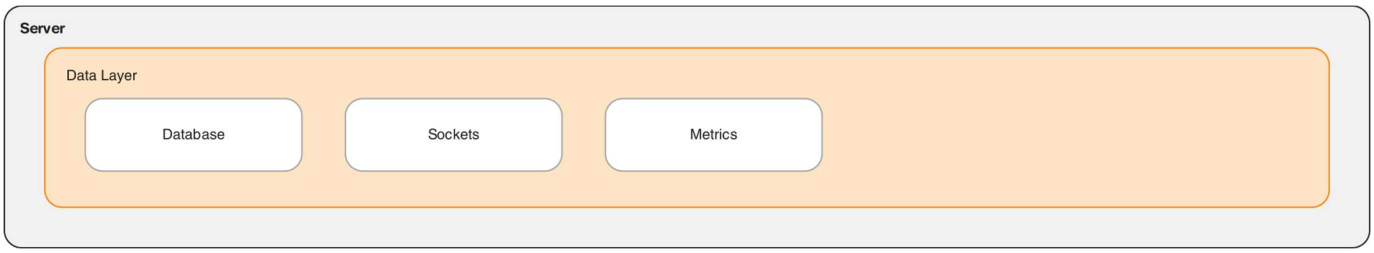
3. Create applications by wiring decoupled code modules together (versus writing a new, very custom codebase with intermingled functionality).
4. Provide a wired or wireless bridge for different parts of a prototype to connect and share data.
5. Leave the prototype open to integration of hardware that calls for a specific SDK or programming language.
6. Leverage a decoupled stack architecture where the controller logic can be written in any language.

We can do this by adopting an architecture standard that centers around a separation of concerns in the prototype which includes an encapsulated **presentation layer, logic layer, and data layer**. This architecture allows for interchangeability as well as a certain level of technical agnosticism and provides the opportunity to create guidelines both per layer and across the stack.

Suggested Architecture



The **presentation layer** contains view functionality with sensors and actuators that facilitate human-computer interaction. The **logic layer** consists of controller hardware running applications and any dependencies to process the I/O from the sensors and/or actuators. It also maintains communication with the server in the data layer.



Our prototypes are already served by our proprietary, wireless server appliance which serves a **data layer**. It runs locally to provide private database, API, and socket communication functionality instead of cloud services or a remote server over the Internet.

Occasionally, a scenario will call for certain devices (such as mobile devices, or microcontrollers with no user interface) that do not directly fit this approach. But given our typical projects, it should be flexible enough for most hardware requirements. The key assumption on which this architecture rests is that a prototype will consist of two key components:

1. A visible user interface that runs on a traditional computer display that runs in the presentation layer.
2. A process running in the logic layer which is driven by a PC tethered to a microcontroller, and handles application and input/output logic respectively, or by a single system-on-chip computer that handles both.

Architecture Guidelines by Layer

Presentation Layer

UI Assets & View Markup

Build the user interface in HTML and CSS, making it more accessible to designers and perhaps a broader range of technicians and developers. We can utilize frameworks like Bootstrap, or develop our own design components.

Front End Code

Write the front end in JavaScript, serving the HTML and CSS of the UI, and leveraging parts of the same Bootstrap and/or custom, reusable frameworks or components we create.

Web App Wrapper

If the prototype application should function like installable software, use JavaScript runtimes like Electron or NWJS. If the logic layer is written in Node.js, it is also possible for both layers to run in the same wrapper and be executed with a double click. When working in Java, the same wrapper or packing functionality exists with Processing.

Sensors and Actuators

The hardware included for input and output should be highly reliable. Use devices from reputable manufacturers that supply datasheets and authoritative code samples. Also, consider the physical environment in which the end solution will be deployed, and what technology is available there. If it is advantageous or even mandatory that we utilize similar technology, we may be locked into using certain devices for input or output.

Logic Layer

Microcontrollers and Other Controller Hardware

New types of microcontrollers, system-on-chip (SoC) computers, and associated peripherals are being released frequently at very low cost and varying levels of quality. So ensuring that the hardware of choice performs with stability is important. Also, when wired connections between sensors and/or actuators to microcontrollers are finalized, move from jumper wires to soldered connections on proto-boards wherever possible, documented either in writing or a Fritzing diagram. This will aid team members in reconnecting portions of a prototype if they are moved or accidentally disconnected, or if the connection scheme is not apparent because of complexity or the length of time since previous use.

Controller Application(s)

A specific sensor, actuator, or piece of controller hardware may require a certain language for the controller application. Whatever that language, it needs to be able to issue GET and POST HTTP requests and utilize WebSockets to function with the server running in the data layer. But to provide as much accessibility within Amazon to the code we write, we should almost always be selecting between JavaScript and Java. (See *Appendix A: Implementation Recommendations* below.)

Support Libraries, SDKs, and Other Dependencies

If the hardware requires us to use a certain library or SDK, use the latest stable version of these dependencies. If you have no prior experience with the dependency in question, test it in a small experiment before use in a prototype.

Data Layer

To provide a local, private communication and control platform for multiple sets of hardware in a single prototype experience, we have built a Node.js application on low-cost Raspberry Pi hardware with API, socket communication, and database functionality. It is physically connected to a Wi-Fi router with a static IP and is accessible to all Wi-Fi enabled clients connected to the router's network. And since the Raspberry Pi has GPIO pins, the server can directly run a separate set of sensors and actuators.

When writing code for your prototype, encapsulate logic so anything connected via WebSockets can easily trigger actions elsewhere, make API requests and react to the responses, and make metrics entries in the database for distinct events.

General Guidelines

Deliverables

We are technologists in a design-heavy team, and we should favor tools, technologies, and methods that enable us to rapidly build strong UX prototypes versus writing production code. However, we should strive to build deliverables that make handoff as seamless as possible, even if the tools and architecture we use do not directly translate into production environments.

In addition to refining the physical equipment—and facilitating vendor relationships for the production of that equipment—we can contribute to the ideal handoff scenario by creating well-documented applications with an architecture that is clear, intuitive, and allows for the mimicking of real production data. If we maintain that focus, we should be able to cleanly pivot into new projects after handoff and maintain pace and fidelity moving forward.

Forward Thinking

Even when locked into hardware or software for prototypes, we should make a case for the deployment of newer, better, or more robust technology if it best demonstrates our proposed solution, provides more benefit for the company, and is cost-effective over the long term.

Conclusion

The goal of this standard is to inform an extensible, composable client side where the presentation and business layers can function as a modular framework without being overly coupled, and with a data layer on the server side that allows us to share information and mimic production data operations, all while collecting metrics on interactions.

This approach should also give us the ability to rapidly connect or even reuse prototyping components with less code while retaining an easy way to employ convention in the user interface, which will be a crucial efficiency when delivering stable prototypes at high fidelity in such a challenging environment.

Appendix A: Implementation Recommendations

While the primary technologist on a project should remain free to make a case for and use the best language or platform for their project, we will normally be selecting JavaScript or Java as the foundation for the logic layer, as they are the two languages most common to development and engineering across engineering teams at Amazon. When using these languages, here are some recommendations for implementation.

JavaScript: Node.js

The Node-serialport library, on which serial communication to controller hardware and several JavaScript IoT frameworks are based is, simple, powerful, and has very thin native bindings. This makes it an excellent choice for serial communication. But to make an Arduino function (for example), you would need to write all communication methods by hand. By loading Firmata onto an Arduino, you can use the Node.js module for Firmata to write JavaScript applications that communicate directly with Arduino over serial, without writing an Arduino sketch.

```
var Board = require("firmata");
var board = new Board("path to serialport");

board.on("ready", function() {
  // Arduino is ready to communicate
  var pin = 13;
  var state = 1;

  board.pinMode(pin, board.MODES.OUTPUT);

  setInterval(function() {
    board.digitalWrite(pin, (state ^= 1));
  }, 500);
});
```

There are many Node.js modules for popular components like Neopixels that further simplify matters. But to make it as easy as possible to work with the type of sensors and actuators we commonly use, consider adding a JavaScript robotics and IoT Framework such as Cylon or Johnny-Five. It uses Firmata to talk to Arduino (or any number of microcontrollers or system-on-chip computers) and allows you to easily write lean, modular Node.js applications that act as hubs between those controllers and other components, comprising an almost “plug and play” development scenario.

```
var Cylon = require("cylon");

Cylon.robot({
  connections: {
    leap: { adaptor: "leapmotion" },
    arduino: { adaptor: "firmata", port: "/dev/ttyACM0" }
  },

  devices: {
    led: { driver: "led", pin: 13, connection: "arduino" }
  },

  work: function(my) {
    my.leapmotion.on("frame", function(frame) {
      if (frame.hands.length > 0) {
```

```
        my.led.turnOn();
      } else {
        my.led.turnOff();
      }
    });
  }
}).start();
```

Java or C

There will be cases where we need to work with a technology such as RFID and use proprietary hardware from a vendor partner. In such cases, an SDK is typically provided and is often available in one of three languages: C, C#, or Java. Our engineering partner will most likely be working in Java for the production version of our prototype, and being OS/platform independent is normally the preferred choice.

Using Java or C over C# can also provide the following advantages:

- We can design within the real-world parameters that our engineering partners will be dealing with.
- Our handoff prototype will be closer to the final product than if written in C#. This will help ensure that the user experience considerations are retained in the final product.
- We can help establish trust in these situations by opting for the language that our partners are more familiar with, providing more transparency into work than an obscure or more platform-dependent language.

When working in Java, consider using Processing, the open-source programming language, and IDE. With its Java syntax and "setup/loop" graphics programming model, it is a versatile prototyping tool maintained and extended by a massive online community. There are both foundation and contributed libraries to easily include functionality for serial, video, network, graphics, audio, and hardware I/O functionality. And if a desired Processing library doesn't exist, you can typically opt for a standard Java library to accomplish your task.

Creating user interfaces is possible for C applications that require more than command line input and output, but it is not quite as straightforward. There are toolkits like QT to consider. Another option is to connect a C server-style app with a JavaScript client using TCP sockets.

Revision History

Version	Date	Notes	Author/Role
0.1	1/14/2020	Initial document created	Gregory Martin, Senior Technologist
0.2	1/19/2020	Included and edited notes on Java per feedback, moved technical details to appendices, formatting changes, updated Application Architecture Overview chart	Gregory Martin, Senior Technologist
0.3	8/20/2020	Added recommendation to solder wiring and document the wiring scheme	Gregory Martin, Senior Technologist
0.4	3/3/2021	Simplified the proposed recommendation, backing away from some specifics and accommodating the broad types of our work	Gregory Martin, Senior Technologist