JSON Interoperability in MIT App Inventor

Thoughts on Terseness Versus Learnability

Evan W. Patton
Massachusetts Institute of Technology
Cambridge, MA, USA
ewpatton@mit.edu

Danny Tang
Massachusetts Institute of Technology
Cambridge, MA, USA
data1013@mit.edu

Abstract

Block languages abstract away the syntax of languages and allow for people to focus on the semantics of a program. Text languages, however, can make use of a variety of syntactical sugar to provide abbreviated means of unpacking complex data structures. We present a use case involving a complex data structure in the JavaScript Object Notation and show how nested elements would be accessed using the MIT App Inventor platform. We then introduce a new block to show how further abstractions within the blocks language can simplify access while making it more readable, more compact, and easier to construct.

Keywords mobile apps, JavaScript Object Notation, blocks

1 Introduction

One way in which one can bootstrap a mobile app is by rendering content provided by a web-based Application Programming Interface (API). The MIT App Inventor platform has support for making web API calls, and for parsing the results in different serializations, such as the JavaScript Object Notation (JSON). However, manipulating the content of these results has been limited to a small set of operators, which can be chained together in complex ways to access internals.

We introduce a use case to demonstrate the limitations of MIT App Inventor with respect to handling JSON data structures, and propose a new set of blocks that will aid in accessing data in nested data structures from JSON-based Web APIs. Further, we consider that manipulation of data structures is a task better suited to text languages and seek thoughts from the blocks-based programming community on how terseness of the blocks representation may or may not affect learnability of material.

2 Related Work

Most blocks language implement some basic form of object that can be manipulated by blocks. Many of these have a closed set of properties, such as App Inventor's components or Scratch's sprites. However, some languages allow for user-defined or open-ended sets of properties on objects

BLOCKS+, November 04, 2018, Boston, MA, USA
2018.

Listing 1. JSON data for a hypothetical web API.

```
{
  "data": {
    "stations": [{
      "num_docks_available": 2
    }, {
      "num_docks_available": 0
    }]
  }
}
```

in the language. For example, Gameblox[1] allows users to add custom properties to sprites, which can be accessed in the blocks. Thunkable × allows one to define types with closed sets of properties. NetsBlox[2] allows users to define messages with a fixed set of fields that can be passed between different workspaces. GP[3] provides an implementation most similar to the one we propose for MIT App Inventor, and allows for setting and getting arbitrary keys on a dictionary instance.

3 Use Case: Processing Web API Response[4]

Lauren is a student who bikes to her university regularly using a bike share. She finds that the bike share system provides a JavaScript Object Notation (JSON) based API for accessing status information about the stations in the system. An abbreviated example is provided in Listing 1.

4 Example Blocks in App Inventor

Lauren uses MIT App Inventor to create an app to access the bike share system's data API and process it to signal her when a dock is available near her destination. Given the structure of the JSON, she composes the blocks structure shown in Figure 1, based on an associative list representation of the data. This method of decomposing the data structure in order to access its internal elements requires 13 blocks. It also requires understanding the type of the data at each step in order to ensure that the correct "not found" value is

[1]https://gameblox.org
[2]https://editor.netsblox.org/
[3]https://gpblocks.org/
[4]This use case idea was inspired by members of MIT App Inventor's Master Trainers group.

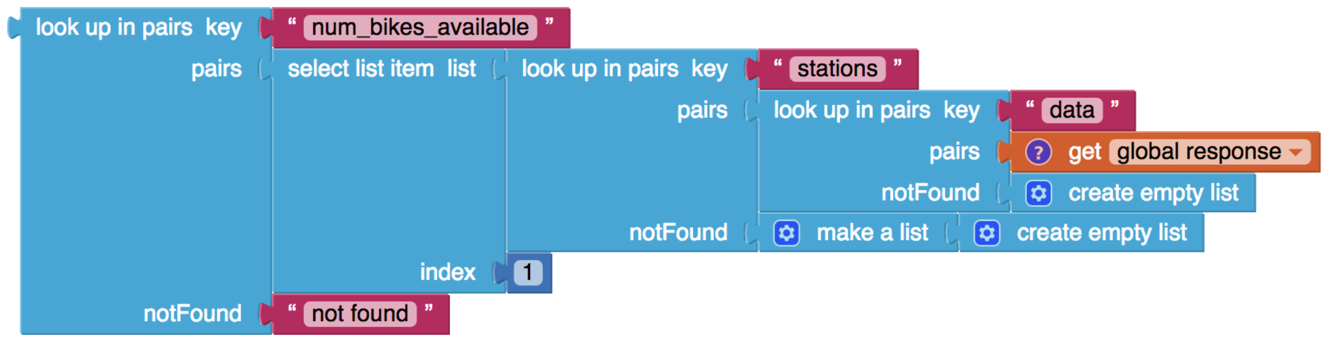

Figure 1. Recursive lookup over the data in Listing 1.

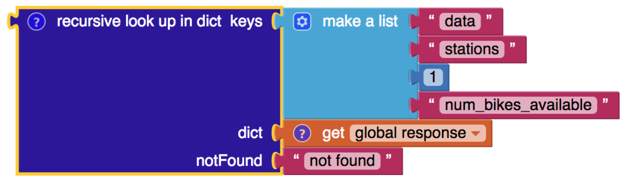

Figure 2. Recursive lookup block that walks both dictionaries and lists to retrieve the appropriate value. If at point the lookup fails to find an item at the given key or index, the "not found" value will be returned.

returned. If the wrong type is returned, a `RuntimeException` will occur.

Later, she learns of a block called "recursive lookup in dictionary" that is less verbose and updates her code to use it (Figure 2). This new approach requires 8 blocks, a reduction of 38%. A second benefit of this approach is that, while it still requires "knowing" the types keys to use at each step in the path from the root to the leaf, this information can be gleaned from looking at the data structure. It does not require knowing that if a lookup is to fail, it needs to return a list versus a string, which is the case in first example.

Another benefit of the new block is that experimentation is less painful. Changing the data structure or the path of interest might require significant modification of the order of lookup blocks (either "lookup in pairs" or "select item in list"). However, for the latter set of blocks, reordering the keys in the list is straightforward, as is extending the list of keys to an arbitrary depth. Each additional step in depth requires only a single block versus an additional three in the first example.

5 Discussion

Text languages typically provide terse representations for accessing complex data structures. For the example data given in Listing 1, accessing the first station's number of bikes could be accomplished in JavaScript through the token sequence `data.stations[0].num_docks_available`.

While our approach in Figure 2 is still more verbose, it is a marked improvement in expressivity over the version expressed given current capabilities in the language (Figure 1).

Further abstracting the blocks, while useful for the purposes of reducing code complexity and space, still require significant cognitive resources to reason about the abstraction, for example knowing at which steps an entity is a dictionary or list, or what the next key or index is that needs to be processed. We could further reduce the barrier to entry of integrating JSON-based web APIs by enabling the developer to visually explore the data structure and dynamically build the blocks for "querying" the data structure in a more visually connected way, for example, by greying out the areas of the structure not returned by the given key path. The introduction of new visual editors or exploratory tools for complex data structures that internally map the logic to blocks may make it even easier for developers to specify the path(s) through data.

Another complex operator for consideration would be the concept of "any" or "all" special blocks that allow the code to select the first (or random) entity or enumerate all entities for the purposes of extracting whole chunks of data matching a similar path. In essence, they would wrap functions such as "random" or "map" without needing to explicitly introduce these concepts into the language.

6 Conclusion

We presented a use case for manipulating data provided by a web API in the JSON serialization. We contrasted two different approaches to accessing the same data in an app built with MIT App Inventor. Even with these representations, text approaches are shorter and potentially easier to understand than their blocks-based equivalents. Further, these blocks make it easier to interact with existing infrastructure, which can help with bootstrapping larger, more complex projects. We recommend that the blocks programming community further explore other representations for manipulating data structures like JSON in blocks.