Zygote: A Framework for Prototyping Smart Devices

Deepak Karki, Aishwarya Kaliki
Department of Computer Science and Engineering
PES Institute of Technology
Bangalore, India

Dr. Ram P. Rustagi
Professor
PES Institute of Technology
Bangalore, India

***Abstract*—The Internet of Things (IoT), which has had an unprecedented effect in the manufacturing industry, is now becoming increasingly popular in everyday life. Even with the rising interest in IoT among developers, there is no simple way for them to prototype and realize their ideas. As of today, developing an IoT project requires cross domain knowledge of hardware, embedded systems, communication protocols, and cloud and web technologies. We propose and implement the Zygote framework, a browser-based tool to lower the entry barrier and enable rapid prototyping and building of IoT projects. Zygote reduces the time and effort required to do so by providing a visual interface to configure, control and deploy an IoT application. It also exposes a REST API for programmatic control as well as a dashboard for quick data visualization. With its modular architecture, it supports the creation of custom plugins as required by the developer. The Zygote framework enables users to build IoT prototypes in minutes and with virtually zero lines of code**

Keywords— IoT, rapid prototyping, visual programming, REST

I. INTRODUCTION

The Internet of Things (IoT) is a revolutionary movement that is changing the way we think about the devices in our daily lives. Applications are being developed to seamlessly connect previously isolated appliances, thereby improving our way of life. With smart systems that can observe pollution levels in water bodies, detect the presence of hazardous gases or monitor the vital signs of those in need of health assistance, the IoT movement is affecting individuals, communities and businesses as well. Its success is further advocated by the sheer number of new systems and applications being developed every day. IoT is expected to play a key role in development and sustenance of smart and interconnected cities [1].

Building an IoT application is a complex task. It usually means working with sensors, e.g. temperature sensors or humidity sensors, and actuators, e.g. motors and alarms. Typically, such components are unintelligent, i.e. they cannot function as part of an application on their own, and need a controller with the necessary programming to become usable. For example, a controller can be instructed to read the value of a temperature sensor once every second. Another controller can be programmed to activate an alarm every minute. Creating an application that is IoT-enabled also means linking these individual elements to allow them to share data, using one or more wireless communication protocols such as Bluetooth, ZigBee or WiFi. New and existing components can be interconnected and combined with innovative logic to perform complex functions such as conserving energy in homes and businesses, forecasting weather or detecting fires. To design such applications, IoT enthusiasts need working knowledge in the hardware and networking fields. Additionally if the data from an IoT system must be stored, processed or visualized, developers need to be familiar with databases, cloud technologies, web development and user interface design.

Developers, when deciding which platforms, protocols and technologies to use, require an understanding of how each one works. This research can sometimes make them lose focus on the true purpose of their application and the value addition they are trying to create. Another hurdle that arises is due to the transient nature of initial designs and prototypes. With every change in the product, whether in its uses or its implementation, a lot of time and effort is wasted on acquiring knowledge of underlying design. Similarly, hobbyists and novices just entering the field of IoT, need a steep learning curve with respect to the diversity of knowledge required. Beginners do not know enough in IoT related area to immediately be able to implement their ideas. The knowledge gap acts not only as a deterrent, it also drastically increase the development time

IoT developers thus need technologies that make the product development process easier and enable rapid prototyping of ideas and applications. The current demographic of the tools available for IoT developers is fragmented, falling into one of three primary categories:

- **Hardware Enablers**: These are inexpensive development boards that have the added advantage of running on low power. These usually include the capability to connect with other boards through wireless protocols such as WiFi/Bluetooth/ZigBee. Examples of such tools are NodeMCU [2], Tessel [3] and Spark [4]. Each hardware board supports a programming language specific to the system (NodeMCU - Lua, Tessel - Javascript, Spark - Arduino). The primary goal of such hardware is to reduce the effort required to connect the board to other devices, by providing communication facilities and APIs to use them.
- **Cloud Solutions for the IoT**: Such technologies enable devices to connect to the Internet, and provide an infrastructure for the sharing of data between those devices. Through the use of public clouds, these applications also provide additional facilities such as data storage and analytics. Examples of such services are Xively [5] and Dweet [6]. These technologies involve the sensors being registered with the cloud and constantly pushing data to the server via a representational state transfer (REST) API for later retrieval.
- **Real-time Data Visualization Dashboards**: These provide a user interface for visualizing the data generated by an IoT application. They are usually web-based and include a library of widgets designed for different occasions. There are many such tools available in the market such as

978-1-4673-9777-3/16 $31.00 © 2016 IEEE
DOI 10.1109/ADCOM.2015.7

Freeboard[7] and Keen Dashboards[8]. Sometimes dashboards are bundled as a part of the aforementioned cloud solutions.

The limitation of these approaches is that each tool addresses only one sub-domain in the IoT ecosystem – hardware, cloud or data visualization and analytics. Integration of these products to build a complete IoT application requires lot of work and expertise. Furthermore, each category of tools has its own drawbacks that limit their usability in development. Hardware enablers each come with their own programming interface, which means developers need to learn a new language for every development board they include in their application. Interconnecting different types of hardware enablers is also difficult since every device is compatible with only a single wireless communication protocol and the developer has to design a common interface between them to facilitate any exchange of data. Cloud-based solutions are well suited for large scale IoT products when they are being deployed, but hard to use for small scale applications especially during the design and prototyping stage. Visualization dashboards are usually only simplex: communication of data is only one sided, from the application server to the dashboard.

There is no simple way to configure, test and deploy an IoT prototype without having to understand several different technologies. Frameworks such as MakerSwarm [9], Spark and Pinoccio [10] do exist, but they are part of a closed ecosystem. The support tools provided are also closed source and work specifically with the hardware given with the framework. Such solutions are not extensible and this becomes severely limiting when developers want to use the features of more than one framework.

On the other hand, there are a variety of open source and platform neutral tools available for IoT, such as the WoTKit [12], Node-Red [13] and *glue.things* [14]. WoTKit is a platform to collect, process and visualize sensor data. It has a web based programming tool which can be used to visually represent the logic to control sensor data and create mashups [11]. Node Red is a visual authoring tool for the internet of things. It is a browser based and implements a flow based programming model to interconnect hardware components. *glue.things* is a modified version of Node-RED that makes it simpler to create mashups of hardware and software services. Each one does well to solve a particular problem it targets, but fails to become an extensible full stack solution for IoT prototyping. WoTKit is a REST based service which does not allow applications to pull data from sensors. It just acts as a mediator between the sensors and clients. One can't set triggers or configure the sensors using it. Node-RED works well when interconnecting resources and setting triggers as it uses WebSockets for streaming data and events. However, the Node-RED interface can be used to interconnect sensors only on one hardware unit, i.e. it cannot be distributed. Though Node-RED runs on a variety of hardware platforms, the sensor plugins developed for one platform can't be used on another. In addition, advanced developers are forced to use the Node-RED UI as there is no REST API available out of the box, making it inflexible. Even *glue.things* suffers from these drawbacks as it is based on Node-RED.

Hence we need a framework that is truly extensible and open. Attempts have been made to design the ideal IoT platform [15]. In this paper we present Zygote, a prototyping platform for the IoT. Zygote is a full stack turnkey solution that integrates IoT hardware, cloud, a graphical authoring tool and a visualisation dashboard to provide seamless experience to IoT developers. It provides REST APIs to access and configure resources (sensors, actuators) and to setup data flows between resources; the data flow happens via WebSockets and supports both directions of flow. Zygote is extensible - the developers can either use the UI provided or can choose to develop their own using the REST API. It is platform neutral i.e. it does not create vendor lock-in. The current prototype implementation does not deal with security issues as is it not our key area of focus. Required security measures can be plugged right into the framework with some effort. The rest of the paper describes the design and implementation of the framework.

II. ARCHITECTURE

The framework is made up of four key components: (i) a server component (zygote-server), (ii) a hardware specific component to run on end nodes (zygote-embed), (iii) a visualization dashboard (zygote-dashboard) and (iv) a browser based control panel (zygote-flowboard). A graphical representation of the architecture and the interconnection between the subsystems is shown in **Figure 1**.

A. Zygote-Server

The zygote-server acts as the centralized mediator between all the nodes connected to it. It has two key interfaces, the REST API interface for configuring nodes and resources on the nodes, and the WebSocket interface to facilitate the flow of real time data and event streams between the various nodes. The server's primary job is to wait for WebSocket connection requests and to route data and events between the connected nodes. The REST API interface is the only way for a client or user to interact with the framework. The interface provides support to query the metadata of connected nodes, create/delete resources on nodes, read/write/configure the resources and create event triggered data flows between resources across nodes.

B. Zygote-Embed

The zygote-embed subsystem is the code that runs on the end hardware platforms such as the Beaglebone Black [16] and Raspberry Pi [17]. These connect to the zygote-server via WebSockets. The key task of this subsystem is to inform the server about the system specific information and to make the on-board resources available to the server via WebSockets and remote procedure calls (RPCs).

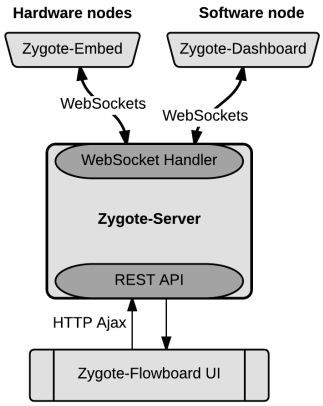

Figure 1. Zygote Architecture

Resources imply sensors and actuators in the case of hardware nodes.

C. Zygote-Dashboard

Most IoT project require some kind of data visualization and graphical interaction panel. The zygote-dashboard is a browser based application that allows the developer to dynamically create and reorder panels and widgets, without having to write any HTML or

Javascript code. Widgets are needed either for visualization of data or taking user input, and panels are used to contain widgets. The dashboard, also called a software node, connects to the zygote-server via WebSockets. Both zygote-embed and zygote-dashboard are called resource containers, where the resources in case of a software node are the widgets and in the case of hardware nodes, they are physical components such as sensors.

D. Zygote-Flowboard

The flowboard is a browser based visual programming tool that is also the brain of the application. It is a GUI layer over the REST API exposed by the server. It enable users to create IoT applications with virtually zero lines of code. The flowboard provides support to create/ delete resources and to develop data flows between them. The flowboard implements the flow-based programming model [18] to interconnect the various sensors, actuators and visualization widgets.

E. Using the Framework

Creating an IoT application using the Zygote framework is accomplished in 4 simple steps, with no complex installation, setup or code required:

- Start the zygote-server using the command line (Linux/Unix based machines).
- Connect the hardware to the server by providing its address and TCP/IP port in the hardware's terminal.
- Start zygote-flowboard in any browser, create resources and specify the flow of data.
- If visualization is required, open the dashboard in a browser window and create widgets.

III. Features and Benefits:

A. Simple and Intuitive

Zygote provides a very simple web-based interface to start the development of an IoT application. It provides a complete visual interface to configure and interact with nodes as well as a graphical programming interface to logically interconnect nodes and create data flows. All sensors and modules are abstracted into visual blocks that can be dragged and dropped onto a workspace and used in creating flows as required.

B. Resource Abstraction and Portability

The core of the embedded component (zygote-embed) is hardware agnostic and all resource objects, including system peripherals (such as GPIO, I2C, and PWM), have a generic *read( ), write( )* and *config( )* (RWC) interface. This abstraction allows the implementation to be different on different platforms, and yet not disturb the framework. To port the framework onto new hardware, two things are required: (i) a *hardware description file* describing the specifications of the new hardware platform and (ii) the device specific implementation of the RWC interface for system peripherals. The implementation could be just a wrapper over the available Node.js input/output (I/O) library available for the board.

C. Protocol Agnostic

While most IoT solutions run all their sensor and actuator nodes on a particular wireless protocol (like ZigBee, WiFi, Z-wave, BLE, and nRF24L01), Zygote effortlessly enables communication between two off-the-shelf components running different protocols. Here, Zygote acts like an IoT hub: the developer just needs to plug the required hardware(sensore/actuator) into the board and adds its specification into the initialization file. The embedded component of Zygote then starts the necessary services and makes the node accessible to the developer via the flowboard.

D. Seamless Cross-Node Resource Access

Resource access across nodes is necessary to make more meaningful IoT applications. All the nodes are connected to the server via WebSockets and the server knows of all the resources on all the nodes. Hence, if a node requests the execution of an operation on a resource, and if the resource does not belong to the node requesting the operation, it is forwarded to the server. The server routes the request to the concerned node, which executes the operation and returns the result to the server. This result is returned to the first node that had requested the operation. All this complexity is hidden within the framework and is invisible to the developer, for whom there is no visible difference between carrying out an operation on a resource that is part of the same node on which the request was generated or on a remote resource on a separate node.

E. Modular Architecture with Plugin Support

For experienced developers, who have created their own sensor board or wireless enabled system, they can easily integrate their platform into the Zygote framework by adding a single plugin file and dropping it in a specific folder. The framework automatically detects it and makes it available on the flowboard. Thus, new sensors and actuators can easily interact with those already supported by Zygote, making the framework easily extensible. The plugin has to expose a RWC interface and can internally use the system peripheral resources (such as GPIO, PWM and UART) and services (such as WiFi and Bluetooth). Once a plugin is written, it can work on any node that is supported by the framework.

F. REST + WebSockets

With a REST-based architecture that also supports WebSockets, Zygote gives developers the best of both worlds. WebSockets are used for inter-node communication which involves streaming data, sending events and performing RPCs. Developers will no longer be forced to choose between using either complex push mechanism or the inefficient pull mechanism for communication in their IoT application. Rather, any combination of the two can be used as required, due to WebSockets and the RPC engine. While the WebSockets make the framework flexible, the REST interface provides a clean API for clients to interact with the framework. New REST endpoints are dynamically generated when resources are created. The endpoints enable users to access and manipulate various resources across the nodes.

IV. Implementation and Usage

A. Zygote-Embed

This is the component of the framework that resides on the end hardware nodes. There are seven modules that make zygote-embed: the main module, the set of services, the platform specific libraries, a local controller, a remote controller, the execution unit and the generic resource class.

The **main.js** file is executed to start the zygote-embed process. It takes command line parameters of the server address, type of the board and desired node name. Then a HTTP POST request is made (with the node's hardware spec file) to the server to register itself

with the desired node name. If successful, the node initializes the required data structures and starts system services.

After initialization, the **remote controller** module takes over and establishes a connection with the zygote-server. This module is responsible for (i) receiving incoming requests from the server and passing it to the correct module based on the type of request, (ii) providing other modules with access to external resources. The **local controller** module maps the URL of the resources to the actual resource object. Whenever there is a request to create/delete or access the resource, the remote controller passes the request to the local controller. For example, if the user requests the creation of a temperature sensor (with endpoint identified by '*2*'), the server will generate a URL */node-name/temp/2* for this new resource. The creation request will be forwarded to the local controller, which will create the resource object and store the mapping between the URL and the object. The resource can be accessed by using this URL which is unique across all nodes in a network. The **platform library** is the collection of hardware specific implementation of the system peripheral resources (GPIO, PWM, UART, etc.). These modules expose the generic RWC interface to ensure that all the plugins and programs that use the interface remain system agnostic. The **resource wrapper** is the interface that supports access of both local and remote resources with the same interface. For example, a resource wrapper object "*res*" can be created as follows:

var res = new Resource("beaglebone", "temp/1");

It now represents the temperature sensor on the "beaglebone" node that has the id "1". The wrapper object also implements the RWC interface. So calling *res.read()* invokes the read function on the temperature sensor object, irrespective of whether it is a local resource. If it is an external resource, the framework will internally perform a RPC, completely abstracted from the developer's view. The flow execution requests that arrive at the remote controller from the zygote-server are passed to the **execution unit**. This is the module that executes the flows sent by the server. The flow is Javascript code sent as a string with additional relevant data as a part of a JSON object. In the flow, all resources access requests are made using the resource wrapper object. Therefore, the flow does not have to worry about where each resource exists. The additional data contains information on the trigger on which the flow should execute, which is either timer based or event based (executed on an event generated by a resource). The execution unit parses the JSON object and sets up the environment to execute the flow on the trigger.

B. Zygote-Server

It exposes a REST API for developers to interact with the system and a WebSocket interface to interconnect the various nodes. The main module (server.js) is used to start the server and initialize the WebSocket handler. The server consists of various controller modules which handle incoming HTTP requests and forward them to the connected nodes via the required socket handler. Users can interact with the nodes (which can be considered instances of zygote-embed and zygote-dashboard) using REST API. The flowboard too internally uses the REST API to communicate with the server.

The **container** module is used to: (i) handle the registration of new nodes, both hardware and software nodes, by accepting POST requests to *host:port/containers/* with the JSON *spec* file of the resource as the request body, (ii) return the list of the active nodes when a GET request is made to the server at *host:port/containers/* and (iii) return the spec file of a particular node when the following GET request is made, *host:port/containers?container=node-name*.

The **resource type** module is used to: (i) handle the creation and deletion of resources on resource containers, which are done by sending a POST or DELETE request to *host:port/node-name/resource-type* with the request body as *{"ep" : "ep-val"}*. If the request is successful, there is a newly created REST endpoint "*host:port/node-name/resource-type/ep-val*" to interact with the resource, where *resource-type* is any one of the different kinds of resources the board supports: GPIO, PWM, etc. and (ii) return the list of active endpoints when a GET request is made to *host:port/node-name/resource-type*.

The **resource instance** module is user to perform RPC on the resource objects on the remote nodes. A HTTP GET request to *host:port/node-name/resource-type/ep?key=val* gets translated to a read request on the resource with parameter *{"key" : "val"}*. A HTTP POST request to *host:port/node-name/resource-type/ep {"k" : "v'}* gets translated to a write request on the resource with parameter *{"k" : "v'}*, where that is the request body. A HTTP PUT request works in the same manner as POST, except config gets called on the resource object.

The **flowboard** module is used to: (i) handle the creation of flows when a POST request is made to *host:port/flowboard* with the flow structure as the JSON request body and (ii) handle the deletion of active flows when a DELETE request is made to *host:port/flowboard* with the the list of flow ids to delete as the request body.

The **dashboard** module is used to: (i) handle incoming requests to read, write and configure widgets in the form of GET, POST and PUT requests respectively to the URL *host:port/dashboard/panel-id/widget-id* and (ii) return the event map structure to the client on a GET request to *host:port/dashboard/events*. The event map is a JSON structure which has the mapping between widget type and the events it can emit. For example, a toggle button widget can emit an "*toggle*" event.

The **socket handler** is the key module that helps in cross talk between the various nodes connected. It holds a collection of open WebSockets, one for each node connected. It also behaves as a RPC engine, by allowing one node to access resources on other nodes and also helping in completing REST API requests. When a function is to be called on a remote resource by a one of the nodes, then a RPC event is emitted to the socket handler via the WebSocket connection, with associated JSON data. The JSON data holds the name of node, the url of the resource, the operation to be performed and finally any associated data with respect to the operation. The socket handler passes the data to the correct node. The target node executes the operation on the resource and returns the value, which is then returned to the first node that initiated the RPC.

C. Zygote-Dashboard

The zygote-dashboard is the web-based component of the framework that deals with visualizing the data generated by the IoT application. It allows users to create widgets that can be configured to communicate with the other elements of the application, either to receive data and display it as the user requires, or to send user generated input to a specified resource. Users can also create panels that act as containers for widgets: by separating widgets into different panels, users can categorize them as required. For example consider a dashboard for a smart home IoT application, that allows users to control lights through IoT-enabled switches. In such a case, button widgets can be created to act as light switches. However, with the number of lights that are present in an average home, this can quickly become confusing. With panels, users can group switches by the

location the corresponding lights are present, which increases clarity and ease of use.

The dashboard consists of two key modules: the dashboard-UI module and the adapter module.

The user interface module consists of the HTML, CSS and Javascript code that define the web page that interacts with the user. The dashboard UI is completely decoupled from the zygote framework, hence it can be used as a separate library in other applications as well (by changing the adapter module).

The UI can be further broken down into three subsystems: (i) the canvas element, (ii) the panel objects and (iii) the widgets.

The canvas element covers the whole page and is responsible for creating and managing panels. The primary function of the panel element is to create and contain widgets and to allow the user to move the widgets within the panel. Widgets are either input or output elements: input elements accept user data or events, and output elements are meant for real time data visualization. The widgets are also considered resources and implement the generic RWC interface.

All objects in the UI implement the event emitter design pattern. The canvas element emits a "*new-panel*" event when a panel is created; the panel emits a "*new-widget*" event when a widget is created under the panel. The newly created objects are available to the listeners as event associated data. This means that the adapter module just has to listen to the required events. The UI is indifferent to the functioning of the adapter: in the future even if the adapter logic changes, the UI can remain untouched.

The **adapter module** is what glues the dashboard to the Zygote framework. The module connects to the zygote-server as a software node when the page is loaded. The adapter consists of three subsystems: (i) the socket handler, (ii) the execution unit and (iii) the event hook module.

The socket handler is the subsystem that maintains the WebSocket connection to the server. It is similar to the remote handler module in zygote-embed. It is responsible for making widgets accessible to the external world, receiving flows, helping in RPCs and updating the server of the new panels and widgets created by the user. The execution unit is exactly the same as the one available in zygote-embed, it is used for flow execution triggered by timer or events. The event hook module is what talks to the dashboard UI. It listens to events generated by the UI and stores the changes to keep track of the latest panels and widgets created/destroyed. When the server requests the socket handler for updates, this module gives the latest UI snapshot as a JSON structure.

D. Zygote-Flowboard

The zygote-flowboard is a graphical programming tool for the Zygote framework. It provides users with a way of creating logical representations for the resources in their application, and connecting them to show the flow of data.

The zygote-flowboard uses the REST API exposed by the Zygote server to periodically get a list of all resource containers connected to the framework at that point, including zygote-dashboard instances. This is the simplest component of the Zygote framework. It basically acts as the visual layer to make use of the REST interface provided by the server. Advanced users can develop their own mechanisms for controlling the active nodes and don't have to rely on the flowboard UI.

The UI is split into two functional parts: the palette and the workspace. The palette on the left holds the list of active nodes and the resource blocks belonging to each node. When a resource is created, the corresponding block is added under the right container. The list of available resources can be updated by clicking the refresh button provided. The workspace is where the users can drag and drop blocks and interconnect them to create the flows. These blocks can belong to one of the following categories:

- **Start Blocks**: This block depicts the beginning of a new flow, and is used to set flow parameters such as flow name and trigger type. The flow once complete, can be activated through this block.
- **Stop Blocks**: Every flow has to terminate with this block. This is also used to deactivate an active flow.
- **Function Blocks**: These blocks are generic blocks with configurable number of outputs for each block. It makes flow development flexible by allowing them to write custom javascript code to manipulate the data streams.
- **Resource Blocks**: These blocks are created by the users themselves and represent the resources on the nodes. Users can use them in the flow if they want to read from or write to a resource.

Once a flow is completed, it can be sent to the zygote-server by clicking on the Start block. With this action, Javascript code equivalent to the flow is generated, packed into a JSON object with additional information about trigger and container and then sent to the server. The server posts the flow to the correct container, where it can be executed when triggered.

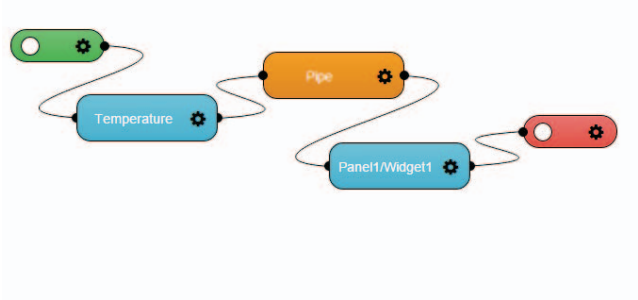

Figure 2. A sample flow that depicts the movement of data from the temperature sensor to a widget on the zygote-dashboard.

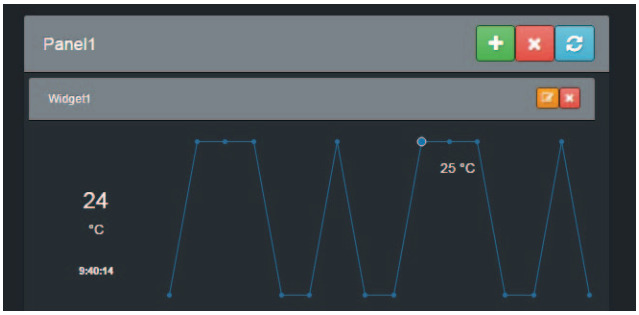

Figure 3. A sample dashboard with a sparkline widget that visualizes the data from the temperature sensor.

E. *Working With the Framework*

In this section we run though the development of a sample IoT application using the Zygote framework. The application will receive data from a temperature sensor and visualize it on a sparkline.

First, start the zygote-server on a Linux machine by going to the server's source folder and type the shell command: *$node server.js*. This starts the server on port 3000 with IP address of the machine. In our setup the server has IP address 10.0.0.2.

Next connect the hardware running an instance of zygote-embed (in this application, it is a Beaglebone Black) to the server by typing: *$node main.js bbb:home 10.0.0.2:3000* on Beaglebone Black's terminal. The first parameter '*bbb*' is the hardware platform identifier and '*home*' is the requested node name. The second parameter represents the server ip address and port.

Setup the WiFi-enabled microcontroller (we used Arduino) with the temperature sensor. The Arduino can be made WiFi-enabled using a serial to WiFi converter. In our setup, this component has the IP address 10.0.0.9.

Open the zygote-dashboard and create a sparkline widget. On the flowboard, create a temperature sensor resource by specifying the IP address of the Arduino. Create a flow to send the data from the temperature sensor to the sparkline widget, by using a pipe. This is basically a function block that does no processing except pass the data from its input end to the output end. The function definition consists of only one line of code. Set the trigger to activate the flow every ten seconds. Launch the flow.

Figure 2 shows the flow created on the flowboard and **Figure 3** shows the temperature data as seen on the dashboard.

As shown by the above example, the developer does not need to configure the system or write complex code. The Zygote framework provides that interface. The developer only needs to visually create the resources and string them together to create an IoT prototype.

V. Results and Conclusion

We have demonstrated the construction of a temperature sensor IoT device using the Zygote framework by writing less than five lines of code. This was made possible due to the plug and play nature of the framework, which also provides graphical configuration and control for the application users. Using this approach any other idea can be realized into a working prototype in a matter of minutes. We have used this framework to build WiFi based presence detector, web controlled RGB bulb, visualization of luminosity and a REST API controlled servo motor in less than an hour. This is the primary goal of Zygote: rapid prototyping of IoT applications. Developers save on the time required to understand and configure the hardware platform they are using. Since the platform and other features of their application can be changed without much effort, prototyping the new and improved designs throughout the development process becomes more cost-effective and less time consuming.

Another advantage Zygote provides is that it can be used by developers at all skill levels to build IoT applications. By taking care of component interconnection and by providing a graphical interface for specifying the flow of data, using the framework drastically reduces the experience required in networking and embedded systems. With the inbuilt dashboard component, developers can visualize the data from their application without needing to learn web and UI development.

During the implementation of the zygote framework, we faced a few challenges as well. First, we had to come up with a design that allows access to the resources on the hardware across the Internet. Here, we had to deal with firewalls and other security barriers. Another issue we faced was developing a technique to handle the transient nature of the nodes, i.e. the nodes may join and leave the network dynamically. We also needed a way to set triggers and on certain events, especially to trigger an action on one node based on an event occurrence on another. Our initial prototype worked only in the LAN (local area network) and did not support transient nodes or the setting of triggers. After a few rounds of analysis and prototyping we came up with the architecture described in this paper – which involved events and data being streamed via WebSockets, REST based access of sensors/actuators and JSON based schema for hardware description.

VI. Future Work

As Zygote is still in its early stages of development, there is a lot of scope for growth. Adding support for more hardware platforms and communication protocols is a task of high priority. As of now the framework supports a simulated hardware and the Beaglebone Black, and support for the Raspberry PI is currently ongoing. The wireless protocol supported as of now is WiFi, and we will soon be adding support for Bluetooth and ZigBee as well. Another feature that is in the plan for the future is IoT security, to prevent malicious access of nodes over the network. The user interface can be extended to support more devices and widget types.

References

[1] Lea, Rodger, and Michael Blackstock. "Smart Cities: an IoT-centric approach."*Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing*. ACM, 2014.

[2] NodeMCU : http://nodemcu.com/. Accessed 2015-05-08

[3] Tessel by Technical Machine: https://tessel.io/. Accessed 2015-05-08

[4] Spark | Build your connected product: http://www.spark.io/. Accessed 2015-05-08

[5] Xively : https://xively.com/. Accessed 2015-05-08

[6] Dweet.io : http://dweet.io/. Accessed 2015-05-08

[7] Freeboard : https://freeboard.io/. Accessed 2015-05-08

[8] Keen Dashboards: https://keen.io/. Accessed 2015-05-08

[9] MakerSwarm https://goo.gl/CAQc5M. Accessed 2015-05-08

[10] Pinoccio: https://pinocc.io/. Accessed 2015-05-08

[11] Guinard, Dominique, Vlad Trifa, Thomas Pham, and Olivier Liechti. "Towards physical mashups in the web of things." In *Networked Sensing Systems (INSS), 2009 Sixth International Conference on*, pp. 1-4. IEEE, 2009.

[12] Blackstock, Michael, and Rodger Lea. "WoTKit: a lightweight toolkit for the web of things." *Proceedings of the Third International Workshop on the Web of Things*. ACM, 2012.

[13] Node-RED: http://nodered.org/. Accessed 2015-05-08

[14] Kleinfeld, Robert, Lukasz Radziwonowicz, and Charalampos Doukas. "glue. things–a Mashup Platform for wiring the Internet of Things with the Internet of Services."

[15] De Roeck, Dries, et al. "I would DiYSE for it!: a manifesto for do-it-yourself internet-of-things creation." *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design*. ACM, 2012.

[16] Beaglebone Black: http://beagleboard.org/black. Accessed 2015-05-08

[17] Raspberry PI: https://www.raspberrypi.org/ Accessed 2015-05-08.

[18] Morrison, John Paul. "Flow-based programming." Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems. 1994.