REST to JavaScript for Better Client-side Development

Hyunghun Cho
Samsung Electronics
hyunghun.cho@samsung.com

Sukyoung Ryu
KAIST
sryu.cs@kaist.ac.kr

ABSTRACT

In today's Web-centric era, embedded systems become mashup various web services via *RESTful web services*. RESTful web services use REST APIs that describe actions as resource state transfers via standard HTTP methods such as GET, PUT, POST, and DELETE. While RESTful web services are lightweight and executable on any platforms that support HTTP methods, writing programs composed of only such primitive methods is not a familiar concept to developers. Therefore, no single design strategy for (fully) RESTful APIs works for arbitrary domains, and current REST APIs are system dependent, incomplete, and likely to change. To help sever-side development of REST APIs, several domain-specific languages such as WADL, WSDL 2.0, and RDF provide automatic tools to generate REST APIs. However, client-side developers who often do not know the web services domain and do not understand RESTful web services suffer from the lack of any development help.

In this paper, we present a new approach to build JavaScript APIs that are more accessible to client-side developers than REST APIs. We show a case study of our approach that uses JavaScript APIs and their wrapper implementation instead of REST APIs, and we describe the efficiency in the client-side development.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages (e.g., description, interconnection, definition)*

Keywords

Web services; API Design; REST API; JavaScript API; Web IDL; JavaScript wrapper

1. INTRODUCTION

The Web-centric era brings various devices into the world of Web and connects them via internet. Many embedded systems and devices consume information from Web servers and produce information like Web servers. Based on such Web technologies, *Internet of Things (IoT)* becomes a new paradigm that connects various devices to process information and provide services.

To support a variety of devices seamlessly, device vendors provide Web servers in the RESTful architecture style [3] that is more lightweight than the conventional Web service style such as SOAP, WSDL, and UDDI [9]. Following the RESTful architecture style, REST APIs categorize resources using Uniform Resource Identifiers (URIs), HTTP, and standard media types like XML and JSON, and provide functionalities by using HTTP methods such as GET, PUT, POST, and DELETE. Because REST APIs couple servers and clients loosely unlike conventional web services, they can support richer services consisting of more complex architectures like 3-tier structures of servers, facades or gateways, and clients.

The simple structure of REST APIs allows embedded systems to provide convergence services that are not easily possible with traditional devices. For example, they can support cloud services that share data between different devices seamlessly without Infrastructure as a Service (IaaS) for file storage and synchronization [14], and they can provide a TV-centric N-Screen Service [16] that can run on multiple platforms. Also, they can support a service that controls and monitors various consumer electronic devices remotely as a home gateway server.

However, the primitive functionalities of REST APIs provide too low-level abstraction for client-side developers. Most client-side developers are not familiar with programming high-level logics in terms of low-level HTTP methods where every functionality is represented by state transfers of resources denoted by URIs.

In this paper, we present a new approach for client-side developers to develop client-side web services more easily. To help developers use their familiar programming styles, we provide JavaScript APIs to the developers and connect the JavaScript APIs and their corresponding REST APIs by wrapper implementation. Our case study of applying the approach to real-world application shows that the approach reduces the development cost and it also reduces the necessary changes of client code due to the changes of REST APIs.

The remainder of this paper is organized thus. In Section 2, we describe why we introduce JavaScript APIs to replace REST APIs for client-side developers. Section 3 presents our approach to build JavaScript APIs from REST APIs, Section 4 shows a real-world case study using the approach. Section 5 discusses related work and Section 6 concludes with future directions.

2. MOTIVATION

Designing (fully) REST APIs is a challenging task and designing standardized REST APIs among various devices is even more difficult. While many design decisions are possible for REST APIs, no clear guidelines or processes to design good REST APIs are available. Therefore, often vendors design different APIs for the same feature. For example, Table 1 shows that different consumer electronics A, B, and C design APIs differently for the same features.

At the same time, generally designed REST APIs provide good evolvability and maintainability for server developers but they are often difficult for client developers to understand and use. The more attributes a resource has, the more difficult to understand and use

WWW'14 Companion,, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04. http://dx.doi.org/10.1145/2567948.2579219.

Feature	A	B	C
Retrieving device status	`/allctrl/status`	`/GetDeviceInfo`	`/sd`
Manipulating power status	`/allctrl/power`	`/SendCommandToDevice`	`/sd`
Retrieving power status	`/allctrl/power`	`/GetDeviceInfo`	`/sd`
Retrieving available commands	N.A.	`/GetCommandListOfDevice`	`/sd`

Table 1: Different URI designs of the same features by different consumer electronics

```
var addr = "http://127.0.0.1/";

function ajaxCall(method, uri, headers, queries,
                  payload, successCallback,
                  isAsync) {
    var httpObj = new XMLHttpRequest();
    var url = addr + uri;
    if (queries != null) {
        var url = "?" + queries;
    }
    for (var i = 0; i < headers.length ; i++) {
        httpObj.setRequestHeader(headers[i].param,
                                 headers[i].value);
    }
    httpObj.onreadystatechange = function() {
        if (httpObj.readyState == 4 &&
            xmlhttp.status == 200) {
            successCallback(xmlhttp.responseText);
        }
    }
    httpObj.open(method, url, isAsync);
    httpObj.send(payload);
}
```

Figure 1: REST API calls with JavaScript Ajax APIs

REST APIs. Because of the characteristics of REST APIs, client developers should implement actions from resource states either by creating new related resources or updating existing resources. Thus, a REST API user should create a resource using the attributes of the existing resources, but it is not trivial because most REST APIs are not self descriptive in the sense that REST APIs do not explicitly describe which attributes are related to other resources. While some attributes are mandatory that a client should set values explicitly, REST APIs do not denote such information clearly.

For client developers to understand the application domain and use REST APIs more easily, we develop JavaScript APIs that are friendlier to client-side developers than REST APIs. Most web applications use two kinds of APIs–REST APIs to use server resources and JavaScript APIs to use client device resources; clients send and receive messages with servers via REST APIs and hosts use JavaScript APIs to communicate with clients. For example, Figure 1 illustrates that a JavaScript web application uses Ajax APIs like `XMLHttpRequest` to invoke REST APIs. Note that the sixth parameter of the `ajaxCall` function, `successCallback`, receives a function as an argument value. Modern web applications use JavaScript extensively to take advantage of its expressive power. Therefore, they use corresponding pairs of REST APIs and JavaScript APIs to communicate between server-side and client-side applications. In this paper, we present an approach to provide JavaScript APIs corresponding to REST APIs to enable web applications to maintain bidirectional communications via JavaScript APIs with server-side processes as in the WebSocket interface [17].

3. OUR APPROACH

We build object-oriented JavaScript APIs from REST APIs so that client-side developers can communicate with servers only in JavaScript. While we can construct JavaScript APIs as one-to-one correspondence to REST APIs as in Figure 1, such APIs are not much more useful than REST APIs. Instead, we provide more reusable object-oriented JavaScript APIs by analyzing the functionalities of REST APIs. The following steps describe a general mechanism to design JavaScript APIs from REST APIs:

1. Collect use cases of a REST API. Often programming guidelines of REST APIs provide usage examples of REST APIs.

2. Extract possible states of a client and the state relationships. Because REST APIs do not keep client states in a server, clients should keep their state information, which allows us to extract client states and state transition conditions in a straightforward way. First, when a URI provides a GET method, name the state of the corresponding resource accordingly. When the attributes of the state are referred by POST, PUT, and DELETE methods of other REST APIs, extract the relationships between states. Also, because URIs make a resource hierarchy, extract the relationships that lower resources in the hierarchy exist only when higher resources do.

3. Introduce an entity that is responsible for the extracted states and their transitions. According to the object-oriented design principles [7], name an interface as a logical object that may have states as their instances or refer them. One interface may have multiple states but we follow the Single Responsibility Principle that assigns one state to one interface.

4. Add state transitions as methods to the introduced entity. To name the methods, consult the REST APIs which use POST, PUT, and DELETE methods for the create, update, and delete methods, respectively.

5. Make the number of method parameters smaller than the parameter number of the corresponding REST API. A REST API can have parameters in several ways: 1) as a part of a URI like `/api/resources/resourceIdValue`, 2) as a query argument like `/api/resource?parameter=value`, 3) as a parameter of a custom HTTP header, and 4) as a JSON content as shown in Figure 2 collectively. The design principles of REST APIs [11] recommend to use URI or header parameters for general-purpose attributes, query arguments for attributes corresponding to auxiliary functions, and JSON contents for other kinds of attributes. Therefore, we decide whether to map the parameters of REST APIs either to attributes of entities or to method parameters in JavaScript APIs according to the principles. We allow reuses of general-purpose attributes by introducing separate interfaces, map auxiliary attributes to methods' optional parameters, and other attributes to mandatory parameters. Because too many method parameters degrade the usability of APIs, we introduce interfaces

```
PUT http://127.0.0.1/api/foos/resourceId?query1=value1&query2=value2    HTTP/1.1
Accept: application/json
Header-Param1 : headerValue1
Header-Param2 : headerValue2

{ bodyParam1 : "bodyValue1",
  bodyParam2 : "bodyValue2" }
```

Figure 2: Sample REST API request

```
[Constructor (DOMString id, HeaderParams headers)]
interface Foo {
    readonly attribute DOMString id;           // originated from resourceId
    readonly attribute HeaderParams headers;
    void update(BodyParams params,     // grouped JSON data as the BodyParams dictionary
                optional DOMString query1,   // originated from query1
                optional DOMString query2);  // originated from query2
};
[Constructor (DOMString header1, DOMString header2)]
interface HeaderParams {
    DOMString header1; // originated from Header-Param1
    DOMString header2; // originated from Header-Param2
};
dictionary BodyParams {
    attribute DOMString param1; // originated from bodyParam1
    attribute DOMString param2; // originated from bodyParam2
};
```

Figure 3: JavaScript API corresponding to the REST API in Figure 2 written in Web IDL

```
var foo = new Foo("resourceId", new HeaderParams("headerValue1", "headerValue2"));
foo.update({
        param1 : "bodyValue1",
        param2 : "bodyValue2"
    }, value1, value2);
```

Figure 4: Sample use of the JavaScript API in Figure 3 in JavaScript

that contain similar attributes and use such interfaces as parameters. Figure 3 presents an example JavaScript API design corresponding to the REST API request in Figure 2 and Figure 4 shows a sample use of the JavaScript API in JavaScript.

After designing the structure of JavaScript APIs, we refine the APIs in an interface description language. In our mechanism, we use Web IDL [8] to describe JavaScript APIs because Web IDL comes with useful tools such as widlproc [10], an open-source tool that converts APIs in Web IDL into XML formats to validate the APIs and to enable automatic generation of stub codes and JavaScript APIs specifications. Figure 3 shows an example Web IDL description of a JavaScript API.

Then, we implement wrappers that map REST APIs to JavaScript APIs. While we can use any implementation language for the wrappers depending on web platforms that provide JavaScript APIs, we use JavaScript as an implementation language so that we can reuse the wrappers in multiple web platforms. We implement wrappers using the JavaScript Ajax APIs as shown in Figure 1.

Finally, we refine the constructed JavaScript APIs iteratively by replacing the uses of REST APIs with the JavaScript APIs incrementally. Instead of designing the entire functionalities of existing REST APIs at once, we design JavaScript APIs corresponding to a core subset of the REST APIs first, check and revise the designed JavaScript APIs, and add more functionalities iteratively.

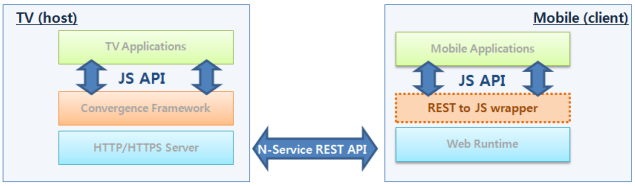

Figure 5: System architecture of the N-Service API

4. CASE STUDY

In this section, we show a real-world case study of our approach described in Section 3 in detail. We use one of the convergence application APIs provided by Samsung Smart TV to illustrate how we convert REST APIs to JavaScript APIs so that we provide JavaScript APIs both to hosts and clients seamlessly. The TV convergence application API (a.k.a N-Service) provides an interface for inter-application communication, which allows Smart TVs incorporating web platforms and nearby mobile devices to develop N-Screen services together. In existing approaches, applications on Smart TVs provide JavaScript APIs for the functionalities of a host, and applications on mobile devices use REST APIs provided by Smart TVs. On the contrary, using our approach, both clients and hosts use JavaScript APIs to implement their applications as shown in

Method	URI	Use case
POST	/ws/apps/{appId}/connect	connect to a host application
POST	/ws/apps/{appId}/disconnect	disconnect from a host application
GET	/ws/apps/{appId}/info	get the information of a host application
POST	/ws/apps/{appId}/queue	push a message to a host application or upload a file
GET	/ws/apps/{appId}/queue/devices/{deviceId}	pop a message of a specific client device from a host application
POST	/ws/apps/{appId}/queue/devices/{deviceId}	push a message to a specific client device
POST	/ws/apps/{appId}/queue/groups/{groupId}	push a message to a specific group
GET	/ws/apps/{appId}/queue/groups/{groupId}	retrieve group members
POST	/ws/apps/{appId}/queue/groups/{groupId}/join	join a group
POST	/ws/apps/{appId}/queue/groups/{groupId}/leave	leave a group

Table 2: List of REST APIs for clients (excerpt)

Interface	Attribute/Method	
NServiceDeviceManager	void getNServiceDevices(successCallback, errorCallback) Get a client device object NServiceDevice connected to a host	
	void registerManagerCallback(callbackFn(ManagerEvent)) Register a callback function that receives events like client connections and disconnections	
	Number broadcastMessage(DOMString message) Send a message to all the clients connected to a host	
	Number multicastMessage(DOMString groupID, DOMString messasge) Send a message to a group	
NServiceDevice	DOMString getUniqueID()	Get the unique id of a client
	DOMString getDeviceID()	Get the device id of a client
	DOMString getName()	Get the name of a client
	Number getType()	Get the type of a client
	Number sendMessage(DOMString message)	Send a message to a client
	void registerDeviceCallback(callbackFn(NServiceDeviceEventInfo)) Register a callback function that receives messages sent by a client from a server	
	void unregisterDeviceCallback() Unregister a callback function that receives messages sent by a client	
ManagerEvent	Number eventType	Type of an event like connection and disconnection
ManagerEvent	DOMString deviceName	Name of a client that signals an event
ManagerEvent	DOMString uniqueID	Unique id of a client that signals an event
ManagerEvent	Number deviceType	Type of a client that signals an event
NServiceDeviceEventInfo	Number eventType Type of an event like sending a message and leaving a group	
	NServiceDeviceMessageInfo eventData or NServiceDeviceGroupInfo eventData Message content	
NServiceDeviceGroupInfo	DOMString groupName	Group name that a client joins or leaves
NServiceDeviceMessageInfo	DOMString message	Message sent by a client
	DOMString context	Context sent by a client

Table 3: List of JavaScript APIs for hosts (excerpt)

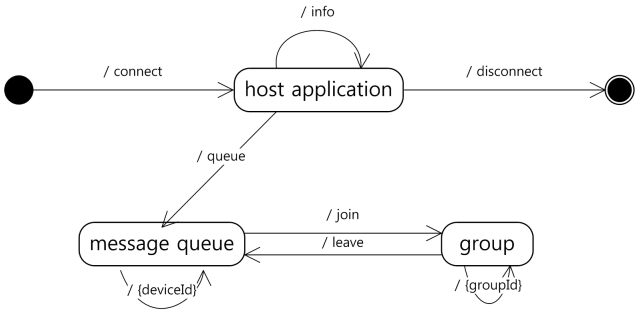

Figure 6: State diagram from use cases

```
POST /ws/app/sampleWidget/connect HTTP/1.1
SLDeviceID: 12345
VendorID: VenderMe
DeviceName: IE-Client
ProductID: SMARTDev
```

Figure 7: N-Service REST API (excerpt)

Figure 5. The resulting JavaScript APIs of our case study are open to the public at the Samsung developer site [15], and anyone can use them on Android.

According to the steps to design JavaScript APIs from REST APIs specified in Section 3, we first collect use cases of REST APIs from programming guidelines to understand their functionalities. Generally, REST APIs often come with documentation that describes their intended uses, and a Samsung development guide provides such a guideline for client to TV application communication [12]. Table 2 shows a list of REST API use cases where `{appId}`, `{deviceId}`, and `{groupId}` denote the id of a host application, the id of a connected device to a host, and id of a generated group, respectively.

In the second step, we extract possible states of a client and the state relationships as illustrated in Figure 6. Following the general mechanism, name the states received by GET methods in Table 2 `host application`, `message queue`, and `group`, and map the REST APIs to transit between the states. Because `queue` and `groups` exist under `{appid}` in the URI hierarchy, we reflect the relationships in the transitions shown in the state diagram.

The third step introduces entities that are responsible for the states and their transitions. From a guideline of JavaScript APIs for host applications [13] shown in Table 3, introduce necessary entities while preferring to reuse existing entities rather than to introduce new entities. We chose to introduce a new `NServiceHost` interface to take care of host applications, and to reuse existing `NServiceManager`, `NServiceDeviceGroup`, and `NServiceDevice` interfaces to manage message queues and groups.

The fourth step designs methods for state transitions. For this case study, we can introduce JavaScript methods for the edge labels in Figure 6 `connect`, `disconnect`, `info`, `queue`, `join`, and `leave`. Because client applications invoke REST APIs by Ajax APIs, we design the methods as asynchronous functions by using success callback functions and error callback functions as parameters.

Finally, the fifth step determines method parameters according to the parameter kinds of the corresponding REST APIs. Because REST APIs often use many parameters, we introduce new interfaces that contain similar attributes and use them as parameters. For example, among the parameters shown in Figure 7, an excerpt from the N-Service REST API, we encapsulate the application id `sampleWidget` and the header parameters `SLDeviceID`, `VendorID`, `DeviceName`, and `ProductID` in newly introduced interfaces `NServiceHostInfo` and `NServiceOwnDeviceInfo` as their attributes appropriately.

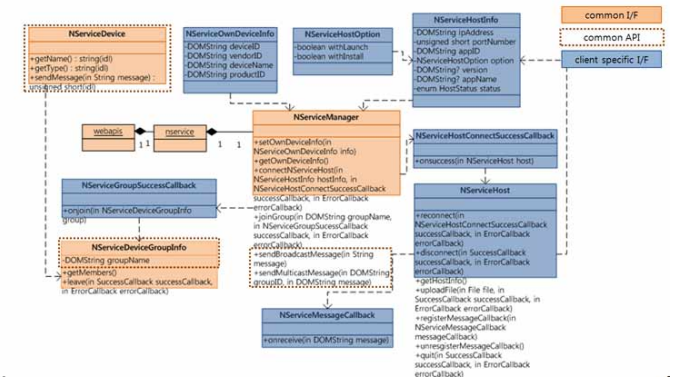

Figure 8: ER diagram of the N-Service API

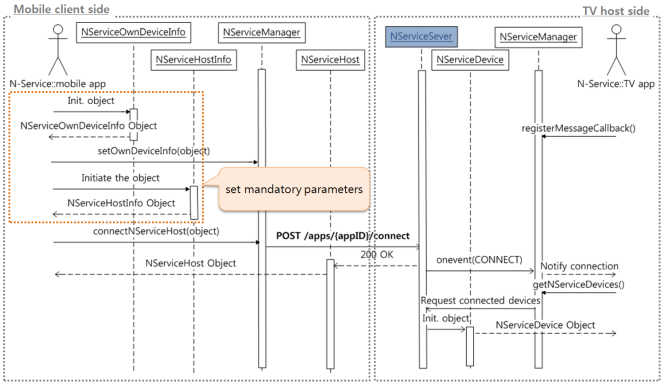

Figure 9: API sequence flow of the N-Service

The ER diagram between client APIs and host APIs in Figure 8 shows that both APIs work seamlessly in JavaScript. The `common I/F` boxes denote the interfaces that already existed in host applications and reused or extended for client applications, and the `client specific I/F` boxes denote newly introduced interfaces for clients. As the `common API` boxes show, both clients and hosts can use the same JavaScript APIs. To check the validity of the designed JavaScript APIs, we use sequence flows that illustrate API call flows. Figure 9 describes a valid order of JavaScript API calls for a part of the APIs shown in Figure 8.

We refine the designed APIs in Web IDL as in Figure 10. The detailed description of the APIs in an interface description language enables to validate the completeness of APIs and to generate documentation of them automatically by various tools.

Then, we implement wrappers that provide JavaScript APIs defined in Figure 10. The generated interfaces and API descriptions guide how to implement wrappers, but we should check security policies of web applications on target platforms. Also, because various platforms including web browsers, Android platforms, Tizen, and Firefox OS, the implementation language may depend on target platforms. While testing sample client applications that use wrappers, we found one security issue with wrappers. Because many platforms use the same origin policy to avoid the security vulnerability by cross-site scripting, client applications using wrappers cannot call REST APIs as Ajax calls because REST APIs and wrappers are

```
[Constructor (NServiceOwnDeviceInfoInit init)]
interface NServiceOwnDeviceInfo {
    attribute DOMString deviceID;
    attribute DOMString vendorID;
    attribute DOMString deviceName;
    attribute DOMString productID;
};

[Constructor (NServiceHostInfoInit init)]
interface NServiceHostInfo {
    attribute DOMString ipAddress;
    attribute unsigned short portNumber;
    attribute DOMString appID;
    readonly attribute DOMString? version;
    readonly attribute DOMString? appName;
    readonly attribute NServiceHostStatus status;
};

[NoInterfaceObject] interface NServiceManager {
    boolean setOwnDeviceInfo(
        NServiceOwnDeviceInfo info);
    NServiceOwnDeviceInfo? getOwnDeviceInfo();
    void connectNServiceHost(
        NServiceHostInfo hostInfo,
        NServiceHostConnectSuccessCallback onsuccess,
        optional ErrorCallback? onerror);
};
```

Figure 10: N-Service API in Web IDL

in different domains. To solve the problem, we made the web server providing REST APIs use the W3C standard Cross Origin Resource Sharing (CORS) [6].

While we did not evaluate the efficiency of using JavaScript APIs compared to REST APIs quantitatively, we found that JavaScript APIs lowered the entry barrier of client developers that they develop more various web applications using the JavaScript APIs. Such applications including Colapicto, Draw Together, Family album, Continuous Youtube play, and Puzzle game are publicly available [15].

With the case study described in this section and other case studies in progress, we learned the following lessons:

- We found a security issue with the REST API design and resolved it using the W3C standard mechanism.
- The object-oriented JavaScript APIs are more usable than REST APIs based on state manipulation.
- The client code using JavaScript APIs are more readable than directly invoking REST APIs in JavaScript.
- The JavaScript APIs provide reasonable documentation as programming guidelines because of their structure based on state transitions.
- We could test the APIs more effectively using JavaScript only code and APIs.

5. RELATED WORK

In this paper, we focus on REST APIs without any support for "hypermedia" and "code-on-demand." According to the Richardson Maturity Model [4] that grades RESTfulness of REST APIs, the target REST APIs of this paper are at Level 2 relying on HTTP verbs and, Level 3 with Hypermedia Controls requires support for hyper media types. Because REST APIs at Level 3 are self descriptive, they are readily applicable to automatically generate client code using Collection+JSON [1] and HAL [2], for example. While upgrading the maturity level of REST APIs for embedded systems may be one way to leverage REST APIs, we suggest an alternative way to provide services and improve them under the current restrictions of embedded systems without changing them.

6. CONCLUSION

We present an approach to enhance the productivity of client developers by providing JavaScript APIs instead of REST APIs that are not familiar to client developers. We describe a general mechanism to design JavaScript APIs from REST APIs systematically and specify the JavaScript APIs in Web IDL to enable automatic validation and documentation generation. A real-world case study demonstrates that our approach is practically applicable and improves programmability of client developers. We believe that JavaScript APIs are more adaptable to future changes in REST APIs so that JavaScript developers can maintain their code more easily.

We plan to improve our mechanism based on our experiences with case studies. We will implement a tool to automatically generate stub code from JavaScript API descriptions written in Web IDL with annotations for REST APIs using widlproc. We also plan to detect misuses of JavaScript APIs due to the changes of REST APIs to make the adjustments of the wrapper implementation effective. Finally, we will apply the mechanism not only to clients but also for servers such as Node.JS [5].

7. ACKNOWLEDGMENTS

We thank the Samsung Web API team members for the case study conducted together and their helpful comments on earlier versions of this paper. This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2011-0016139 and NRF-2008-0062609) and Samsung Electronics.

8. REFERENCES

[1] Amundsen, M.: Collection+JSON. https://github.com/collection-json
[2] Chambrier, N.: Hypertext application language. https://hpmjs.org/package/hal
[3] Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)
[4] Fowler, M.: Richardson maturity model. http://martinfowler.com/articles/richardsonMaturityModel.html/
[5] Joyent: Node.js. http://nodejs.org/
[6] van Kesteren, A.: Cross-origin resource sharing. http://www.w3.org/TR/cors/
[7] Martin, R.C.: Principles of object oriented design. http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign
[8] McCormack, C.: Web interface definition language. http://www.w3.org/TR/WebIDL
[9] Moller, A., Schwartzbach, M.I.: An Introduction to XML and Web Technologies. Addison-Wesley (2006)
[10] Renouf, T., Byers, P.: widlproc. https://github.com/dontcallmedom/widlproc
[11] Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media (2007)
[12] Samsung: Client (HHP) to TV application communication. http://www.samsungdforum.com/Guide/ref00003/convergence_app_clienttotvappcomm.html
[13] Samsung: Nservice. http://www.samsungdforum.com/Guide/ref00008/nservice/dtv_nservice.html
[14] Samsung: Samsung Link. http://link.samsung.com
[15] Samsung: Samsung Web API. http://developer.samsung.com/samsung-web-api
[16] Samsung: Smart TV Convergence App API. http://www.samsungdforum.com/Guide/ref00003/index.html
[17] W3C: The WebSocket API. http://www.w3.org/TR/websockets/