A Toolkit for Simplified Web-Services Programming

Moshe Chai Barukh and Boualem Benatallah

School of Computer Science & Engineering,
University of New South Wales, Sydney – Australia
{mosheb,boualem}@cse.unsw.edu.au

Abstract. The Internet has truly transformed into a global deployment and development platform. For example, Web 2.0 inspires large-scale collaboration; Social-computing empowers increased awareness; as well as Cloud-computing for virtualization of resources. As a result, developers have thus been presented with ubiquitous access to countless web-services. However, while this enables tremendous automation and re-use opportunities, new productivity challenges have also emerged: The same repetitive, error-prone and time consuming integration work needs to get done each time a developer integrates a new API. In order to address these challenges, we designed and developed *ServiceBase*, a "programming" knowledge-base to abstract, organize, incrementally curate and thereby re-use service-related programming-knowledge. Empowered by this knowledge we then provide: (a) A set of APIs that expose a common and high-level interface to developers for integrating services in a simplified manner; (b) An extended version of the GIT repository, creating a *plug-n-play* environment for services; (c) A mind-map based visualization tool to help explore the base.

1 Introduction

It is no doubt that the inception of the Service Oriented Architecture (SOA) paradigm has significantly empowered the capabilities for modern software engineering. Moreover, coupled with other recent advances in web-technology, such as Web 2.0, Social and Cloud computing, the Internet has delivered developers with ubiquitous access to a plethora of rich and logical services along with computing resources, data sources, and tools – and the potential to build powerful systems. For instance, *ProgrammableWeb* now lists more than 6,700 APIs in its directory. Moreover, in order to increase the dispersion of APIs, organizations such as *Mashery*[1] and *Apigee*[2] are building on these trends to provide platforms for the management of APIs. However, while advances in Web-service and services composition have enabled tremendous automation and reuse, new productivity challenges have also emerged. The same repetitive, error-prone and time consuming integration work needs to get done each time a developer integrates a new API. Moreover, the heterogeneity associated with services also means service programming has remained a technical and complex task. For example, the developer require sound understanding of the different service types and access-methods, as well as being able to format input data,

[1] http://www.mashery.com
[2] http://apigee.com

X. Lin et al. (Eds.): WISE 2013, Part II, LNCS 8181, pp. 515–518, 2013.

or parse and interpret output data in the various formats, (e.g. XML, JSON, SOAP, Multimedia, HTTP, etc.). In addition, programmers may also need to develop additional functionality such as: user management, authentication-signing, access control and third-party authorization; tracing as well as version management, etc.

In order to address these challenges, we have designed and developed *ServiceBase*, a "programming" knowledge-base, where common service-related low-level logic can be abstracted, organized, incrementally curated and thereby re-used by other application-developers. Architecturally, we have drawn inspiration from *Freebase* and *Wikipedia*, where just as encyclopediatic information is distributed in the form of user-contributed content, similarly, technical knowledge about services could be both populated and shared amongst other developers. Empowered by that knowledge, we then provide a set of APIs that expose a common and high-level interface to developers for integrating services in a simplified manner, despite their underlying heterogeneity. This is facilitated by the implemented service-bus middleware that translates high-level methods to more concrete service calls, by looking-up and then building the necessary information from the knowledge-base. While this itself significantly simplifies service-programming, we have been motivated to provide further support in the form of a supporting programming toolkit for service-based programming. In particular we implement two extensions: (a) To ease access, navigation and exploration of the service-base from within a typical programming environment, we implement an extended version of the GIT repository, creating a plug and play environment of services. This means new services can be added to the knowledge-base (i.e. "plugged"), and registered services can be utilized in application development (i.e. "played"); and (b) In order to help visualize service meta-data, (for example determining what are the message-parameters and formats for some service), we have implemented a mind-map visualization of the service knowledge-base.

Inevitably, our work provides the potential for a vast increase in application-development productivity and greater code-maintainability. We have endeavored to fill the gap amongst non-highly skilled programmers who often resort to homebrewed systems; or even skilled programmers who often end up re-implementing systems (or part of systems) due to being unable to locate a suitable API, or if it being too hard to understand how to use. In order to substantiate our results, we demonstrate in this paper how a service-oriented application can be built using various services that cover the 3 main different service-types, (WSDL, REST & ATOM/RSS), and implemented faster and in much fewer lines of code than if using a traditional coding approach.

2 *ServiceBase* System Architecture

As illustrated in Figure 1, *ServiceBase* has been realized in three layers of abstraction: (i) Firstly, the *Programming Knowledge-Base* acts as the central repository for storing all service-related programming information. (ii) The *ServiceBus* then exposes a set of Java (client-library) and RESTful APIs that enable simplified programmatic access to registered services. Finally, (iii) we provide a *toolkit* in order to initiate a supportive programming environment. Consisting of: an extended GIT-Repository that enables *plug-n-play* capabilities via a command-line interface; as well as a web-based mind-map visualization GUI for exploring and traversing the graph service-base.

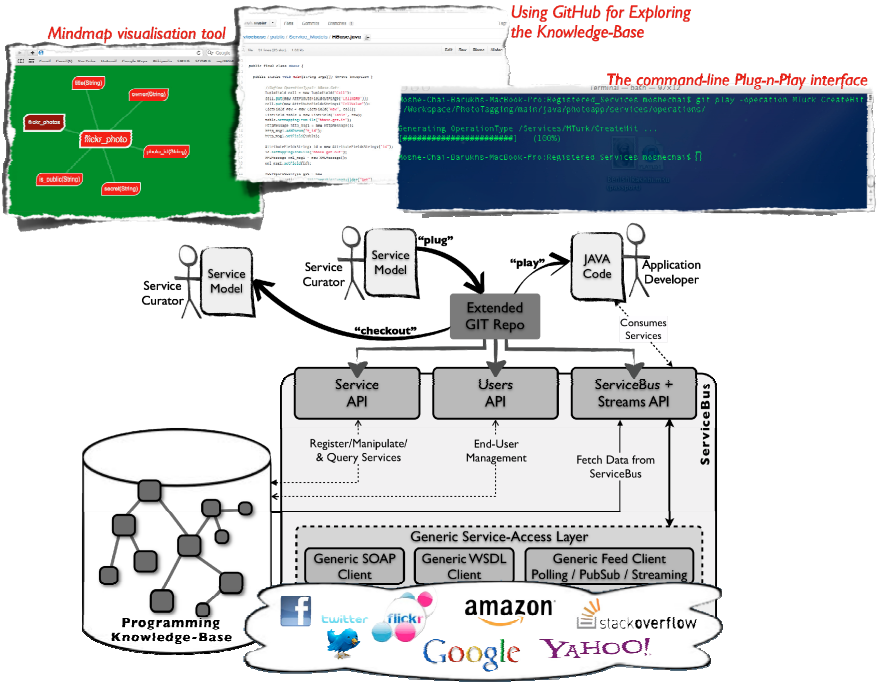

Fig. 1. ServiceBase System Architecture

The programmatic interface to *ServiceBase* offers the following APIs:

The **UsersAPI** provides a means for end-users and apps to register and identify. Identity management is achieved via *Facebook* login credentials as well as native login. 3rd-party application interaction is achieved via OAuth 2.0 Provider interface.

The **ServicesAPI** is used to *register* new services into the knowledge base; as well as *search*, *explore*, *update* and *delete* service-models that are already registered.

The **ServiceBusAPI** is the main gateway for application-developers to interact with typical request-reply ops via the `invoke(..)` methods. The main innovation is the common programming interface provided to interact with services, irrespective of their underlying heterogeneity (e.g. JSON/REST vs. XML/REST vs. SOAP/WSDL).

The **StreamsAPI** is used to enable interaction with real-time feeds-services. It provides a `subscribe(..)` method to any registered feed. The API then exposes three modes for working with subscriptions: (a) It provides *querying* (i.e. pull) of feed events; (b) It enables setting up *listeners* (i.e. asynchronous callback push) of events; and (c) It enables *navigating* streams using `pause()`, `stop()` and `rewind(date)`.

The **Plug-n-Play API** is used to even further simplify service-programming. We have done this by extending the standard GIT commands with `"plug"` (helps to automate adding new services); and `"play"` (generates a code-stub to use a service). Usage shown below:

```
git plug <service_model_file>

git play [-operation | -feed [-get | -callback]]
         <service_name><operation_name>|<feedtype_name>
         <destination_location>
```

3 Demo Scenario

We demonstrate the toolkit over a reasonably complex scenario involving 3 different service-types: Namely, Amazon *MTurk* expressed in WSDL, *Flickr* expressed in REST/ATOM; and Apache *HBase* Database-as-a-Service, expressed in REST.

Description. We want to build an application to employ the crowd for determining appropriate captions for photos added to some *Flickr* photo-stream. Upon an *MTurk* worker accepting the work, we would like to store this result in *HBase* for later retrieval. Finally, we want to write an app to query-on-demand the proposed captions and ask for verification – upon approval the captions can then be updated on *Flickr*.

Screencast. As described below, we demonstrate the implementation of this scenario in only three simple stages and in less than *100 lines* of code with no extra dependencies. We have only shown snippets of generated code-stubs below, however we demonstrate the entire implementation scenario and execution in the following screencast: `http://www.cse.unsw.edu.au/~mosheb/servicebase.html`

Stage 1. Subscribe to the required photo-stream feed, and then set-up a callback listener. The callback is formulated using two services, and for each the code stubs can be generated. I.e. (i) Listening to the stream for photo-uploads (Lines 11–19); and then (ii) Posting a HIT on *Mturk* (Lines 1–10). Therefore, the code in (i) calls (ii).

Stage 2. When each HIT has been completed by the crowd-worker, we invoke the *HBase* data-store service to curate the results. The HIT is in fact represented as an HTML form, which post data to the *HBase* services via the Service-bus RESTful API.

Stage 3. Finally, we write the verifier web-app. Again, most of the code could be auto-generated via the play function: i.e. getting the photos from *HBase* and displaying to the verifier. Then if verified, updating the photo by posting to *Flickr*.

```
1.  public class MTurk_CreateHit {
2.      public static Message invoke(String title, String desc, File qn,
            String duration, String lifetime, String reward) {

3.          WSDLService mturk = ServiceAPI.getService("AmazonMTurk");
4.          TupleField hit = mturk.getOperationType("createHit")
                                  .getInputMessage().getField();

5.          hit.getField("title").setValue(title);
6.          hit.getField("description").setValue(desc);
            ...
7.          hit.getField("question").setValue(qn);

8.          return ServiceBus.invoke(acs_key, mturk, "createHit", hit);
9.      }
10. }
                                        ❖ ❖ ❖
11. public class Flickr_GroupPoolFeed {
12.     public static void callback(Field field, String sub_id){

13.         TupleField photo = ((TupleField)field);
14.         String id = photo.get("photo_id").getValueAsString();
15.         String url = photo.get("photo_url").getValueAsString();

16.         File question_html = getHtmlQuestion(id, url);
17.         MTurk_CreateHit
                .invoke("Describe an Image",        //title
                        "Tagging of an Image",      //description
                        question_html,              //question HTML
                        "86400",                    //duration = 1 day
                        "7",                        //lifetime = 7 days
                        "1"                         //reward = 1 dollar
18.     }
19. }
```