ANY2API – Automated APIfication
Generating APIs for Executables to Ease their Integration and Orchestration for Cloud Application Deployment Automation

Johannes Wettinger, Uwe Breitenbücher and Frank Leymann
Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, Stuttgart, Germany

Keywords: Cloud Computing, DevOps, API, APIfication, Service, Web, REST.

Abstract: APIs are a popular means to expose functionality provided by Cloud-based systems, which are utilized to integrate and orchestrate application as well as management functionality in a programmatic manner. In the domain of application management, they are used to fully automate management processes, for example, to deploy Cloud-based Web applications or back-ends for mobile apps. However, as not all required functionality is exposed through an API natively, such processes additionally involve a multitude of other heterogeneous technologies such as scripting languages and deployment automation tooling. Consequently, combining different technologies in an efficient manner is a complex integration challenge. In this paper, we present a generic approach for automatically generating API implementations for arbitrary executables such as scripts and compiled programs, which are not natively exposed as APIs. This *APIfication* tackles the aforementioned integration challenges by unifying the invocation of heterogeneous technologies while avoiding the costly and manual wrapping of existing executables because it does not scale. We further present the modular and extensible open-source framework ANY2API that implements our APIfication approach. Furthermore, we evaluate the approach and the framework by measuring the overhead of generating and using API implementations. In addition, we conduct a detailed case study to confirm the technical feasibility of the approach.

1 INTRODUCTION

A remarkable amount of today's applications, especially Web applications as well as back-end systems and platforms for mobile apps, provide *application programming interfaces* (APIs) (Richardson et al., 2013). The main purpose of an API is to provide a well-defined and documented interface, which is exposed to access and utilize application functionality in a programmatic manner. APIs hide and abstract from implementation-specific details such as invocation mechanisms and data models inherited from the technology stack on which a particular application is built upon. This is the foundation for integrating and orchestrating different applications and application components, enabling systematic development and reliable operations of distributed applications, mash-up applications, and mobile apps. Furthermore, APIs are used to integrate applications with business partners, suppliers, and customers (Rudrakshi et al., 2014). Even devices can be connected and interconnected to enable the *Web of things* (Guinard et al., 2010). Technically, APIs can be exposed and utilized in different forms. Both (i) libraries that are bound to a particular programming language and (ii) language-agnostic Web services, e.g., Web-based RESTful APIs (Richardson et al., 2013; Masse, 2011) or WSDL/SOAP-based services (W3C, 2007) are widespread forms of providing and using APIs. Popular providers such as Twitter, GitHub, Facebook, and Google offer such libraries[1] and Web services[2]. However, libraries and Web services are not mutually exclusive, meaning libraries often use Web services in the background, but adding an additional layer of abstraction to seamlessly integrate with the programming model of the corresponding language. Consequently, *Web APIs are a platform-independent and language-agnostic means for integration and orchestration purposes,* optionally enhanced by additional language-specific libraries. Regarding the terminology used in this paper, we consider a *Web API* as one particular kind of *API*. The use cases, examples, and implementations discussed in this paper mostly focus on Web APIs. However, the concepts

[1] Google APIs client libraries: http://goo.gl/uVvFf
[2] Google Compute Engine API: http://goo.gl/cj0BGl

Wettinger J., Breitenbücher U. and Leymann F..
ANY2API – Automated APIfication - Generating APIs for Executables to Ease their Integration and Orchestration for Cloud Application Deployment Automation.
DOI: 10.5220/0005472704750486
In *Proceedings of the 5th International Conference on Cloud Computing and Services Science* (CLOSER-2015), pages 475-486
ISBN: 978-989-758-104-5
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

and methods are suitably generic to be applied to other kinds of APIs, too.

The number of publicly available Web APIs is constantly growing[3]. As of today, the API directory *ProgrammableWeb*[4] lists more than 12000 APIs. Popular providers such as Google, Facebook, and Twitter are serving billions of API calls per day[5]. These statistics underpin the importance and relevance of APIs. Existing literature (Masse, 2011; Richardson et al., 2013) and frameworks such as Hapi[6] (Node.js) and Jersey[7] (Java) provide holistic support, best practices, and templates for building Web APIs. While this is state of the art for creating Web applications and back-ends for mobile apps, Web APIs as a platform-independent and language-agnostic means for integration and orchestration purposes are heavily utilized for automating the deployment and management of Cloud applications (Mell and Grance, 2011; Wettinger et al., 2014a), which leads to significant cost reductions and enables applications to scale: Cloud providers offer management APIs that can be programmatically used in a self-service manner, e.g., to provision virtual servers, deploy applications using platform services, or to configure scaling and network properties.

However, because such management APIs typically provide basic functionality only, they have to be combined with further configuration management systems to realize non-trivial deployment scenarios: a huge number of reusable artifacts such as scripts (e.g., Chef cookbooks (Nelson-Smith, 2013), Juju charms[8], Unix shell scripts) and templates like Docker container images (Turnbull, 2014) are shared by open-source communities to be reused in conjunction with provider-supplied services. While APIs can be orchestrated easily due to well-known and common protocols (e.g., HTTP), the technical integration with these different artifacts and heterogeneous management systems is a very error-prone, time-consuming, and complex challenge (Wettinger et al., 2014a). Thus, to build, deploy, and manage non-trivial Web applications, it is of vital importance to handle the invocation of different artifacts, technologies, and service providers in a technically uniform manner to focus on the orchestration level, neglecting lower-level technical differences.

Unfortunately, many of these individual artifacts are *executables* that cannot be utilized through an API without a central middleware component (Wettinger et al., 2014a) such as a service bus that (a) maps generic API calls into executable-specific invocations, (b) translates inputs and results of the invocation, and (c) makes them available through an API endpoint. However, this central middleware approach comes with three major drawbacks: (i) the individual artifacts are not packaged with their API to be utilized at runtime and, thus, they are not self-contained; (ii) in order to utilize the executables through an API, a central middleware component is inevitably required in addition to the individual artifacts to be invoked which results in additional costs and maintenance effort; (iii) in case a new kind of executable comes in, the central middleware has to be adapted, extended, and redeployed accordingly with potential risks such as downtime, functional failures, and unintended side effects. Today, this is nothing exceptional because open-source communities constantly share new kinds of artifacts such as Chef cookbooks, Juju charms, and Docker container images to name a few examples from the domain of application deployment automation.

The main goal of our work is overcome these drawbacks by introducing an automated approach to *generate API implementations (APIfication)* that are packaged including the corresponding artifacts such as the executable and all its dependencies in a portable manner. This makes them truly self-contained without depending on a central middleware. The generated API implementations simplify the orchestration of different kinds of artifacts and their integration with existing provider-hosted APIs. Therefore, the major contributions of this paper are as follows: (i) We present an automated *APIfication method,* respecting the requirements we derived from a use case and motivating scenario in the field of Cloud computing and deployment automation. (ii) We introduce an *APIfication framework* to implement the method we presented before and provide a prototype implementation to demonstrate the feasibility. (iii) We *validate* the proposed APIfication approach using a prototype implementation and perform an *evaluation* to analyze the efficiency of our approach. (iv) We conduct a *case study* in the field of deployment automation and discuss further use cases of the APIfication approach in other fields such as e-science.

The remainder of this paper is structured as follows: Section 2 describes the problem statement, including a use case and motivating scenario in the field of deployment automation. Based on the generic APIfication method presented in Section 3, we propose and discuss an APIfication framework in Section 4. Our prototype implementation ANY2API as well as its validation and evaluation are discussed in Section 5. Moreover, we present a case study in that section. Section 6 outlines further use cases to apply our APIfication approach.

[3]ProgrammableWeb statistics: http://goo.gl/2eQ01o
[4]ProgrammableWeb: http://www.programmableweb.com
[5]ProgrammableWeb calls per day: http://goo.gl/yhgyyW
[6]Hapi: http://hapijs.com
[7]Jersey: http://jersey.java.net
[8]Juju charms: https://manage.jujucharms.com/charms

Finally, Section 7 and Section 8 discuss related work, future work, and conclude the paper.

2 PROBLEM STATEMENT & USE CASE

As discussed in Section 1, APIs serve as a platform-independent and language-agnostic means for integration and orchestration purposes. There are several frameworks based on different programming languages and technology stacks established to develop APIs, especially Web APIs. However, an individual API still needs to be implemented manually using these development frameworks. While this is state of the art for creating new applications such as Web applications or back-ends for mobile apps, for some use cases the individual development of an API is not feasible or even impossible. This is due to scaling issues (e.g., creating APIs for a huge amount of individual executables) or missing expertise, meaning the person, who needs to utilize certain functionality is not able to develop a corresponding API. In the following we discuss an important use case that requires API implementations to be generated in an automated manner.

2.1 Use Case: Deployment Automation

A major use case originates in the DevOps community (Hüttermann, 2012), proposing the implementation of fully automated deployment processes to enable continuous delivery of software (Humble and Farley, 2010; Wettinger et al., 2014b). This is the foundation for rapidly putting changes, new features, and bug fixes into production. Especially users and customers of Cloud-based Web applications and mobile apps expect fast responses to their changing and growing requirements. Thus, it is a competitive advantage to implement automated processes to enable fast and frequent releases (Hüttermann, 2012). As an example, Flickr performs more than 10 deployments per day[9]; HubSpot with 200-300 deployments per day goes even further[10]. This is impossible to achieve without highly automated deployment processes. The constantly growing DevOps community supports the implementation of automated processes by providing a huge variety of individual approaches such as tools and artifacts to implement holistic deployment automation. Reusable executables such as scripts, configuration definitions, and templates are publicly available to be used for deployment automation. Juju charms and Chef cookbooks are examples for these (Nelson-Smith, 2013; Sabharwal and Wadhwa, 2014). Such executables usually depend on certain tools. For instance, Chef cookbooks require a Chef runtime, whereas Juju charms need a Juju environment. This makes it challenging to reuse different kinds of heterogeneous artifacts in combination with others. Especially when systems have to be deployed that consist of various types of components, typically multiple tools have to be combined because they focus on different kinds of middleware and application components. Thus, there is a variety of solutions and orchestrating the best of them requires to integrate the corresponding tools, e.g., by writing scripts that handle the underlying lower-level invocations, parameter passing, etc. However, this is a difficult, costly, and error-prone task as many of the executables cannot be utilized through an API without relying on a central middleware component. Consequently, all artifact- and tooling-specific details (invocation mechanism, rendering input and output, etc.) have to be known and considered when integrating and orchestrating different kinds of executables. We tackle these issues with our work presented in this paper by generating APIs for individual executables. The generated APIs hide and abstract from artifact- and tooling-specific details, thereby significantly simplifying the integration and orchestration of very different kinds of artifacts.

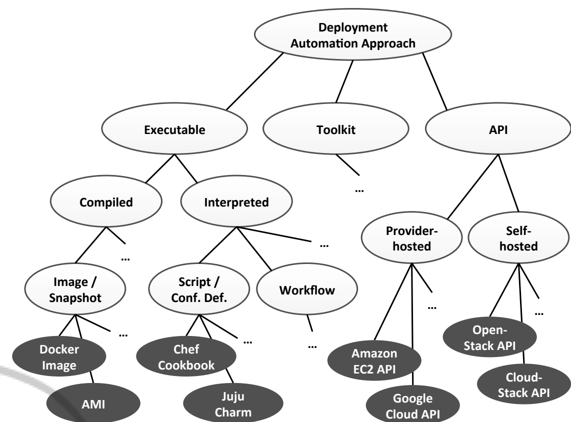

Figure 1: Deployment automation classification.

Figure 1 shows an initial classification of deployment automation approaches. Executables are categorized in *compiled* and *interpreted* artifacts. Examples for compiled executables are pre-built virtual machine snapshots and container images such as Amazon machine images (AMI)[11] or Docker container images[12].

[9] Flickr deployments per day: http://goo.gl/VEmVqE

[10] HubSpot deployments per day: http://goo.gl/4AQy1h

[11] AMIs: http://goo.gl/S1Zx8Q

[12] Docker Hub Registry: https://registry.hub.docker.com

In contrast to those, scripts and configuration definitions such as Chef cookbooks and Juju charms are interpreted at runtime. Beside executables, existing *APIs* can be utilized in two flavors: (i) provider-hosted APIs are offered by Cloud providers to provision virtual servers, storage, and other resources; (ii) self-hosted APIs are offered, e.g., by open-source Cloud management platforms such as OpenStack (Pepple, 2011). Our work focuses on transforming existing individual executables into self-hosted APIs by generating corresponding API implementations. As a result, full deployment automation can be achieved by integrating and orchestrating provider-hosted and self-hosted APIs without considering the tooling- and artifact-specific details of different kinds of executables. Moreover, this approach broadens the potential variety of tools and artifacts because their implementation-specific differences are completely hidden by using the generated API implementations.

Technically, the integration and orchestration of generated and existing APIs can be implemented using arbitrary scripting languages such as JavaScript, Ruby, or Python; alternatively, service composition languages such as BPMN (OMG, 2011) or BPEL (OASIS, 2007) may be used. For scripting languages, provider-independent and provider-specific *toolkits* are available to implement deployment plans that orchestrate and integrate different APIs. Examples are fog[13] and Google's API libraries[14]. Furthermore, general-purpose libraries to interact with different kinds of Web APIs are available for all major scripting languages: restler (JavaScript)[15], node-soap (JavaScript)[16], rest-client (Ruby)[17], Savon (Ruby)[18], etc.

2.2 Motivating Scenario: Facebook App

Considering the deployment automation use case discussed before, this section presents a comprehensive example as motivating scenario: the automated deployment of a Cloud-based Facebook application. The structure and parts of the application are shown in Figure 2. A *canvas frame*[19] is used to create and embed a corresponding application on the Facebook platform. The *canvas URL* points to an externally hosted Web application that is run based on a PHP runtime environment. It provides both the user interface and the underlying application logic. The PHP runtime itself is provided by an Apache HTTP server in conjunction with a PHP module. Both are deployed on a virtual machine, running Ubuntu 14.04 as operating system, which itself runs in the Cloud, hosted on Amazon's public infrastructure (EC2[20]). The scenario covers a typical setting used to deploy and run Web-based social applications as it employs and combines modern social media platforms such as Facebook as well as Cloud infrastructures such as Amazon EC2. It could be further refined, e.g., by connecting the Web application to a database that is provided by a database-as-a-service offering hosted on a different Cloud infrastructure.

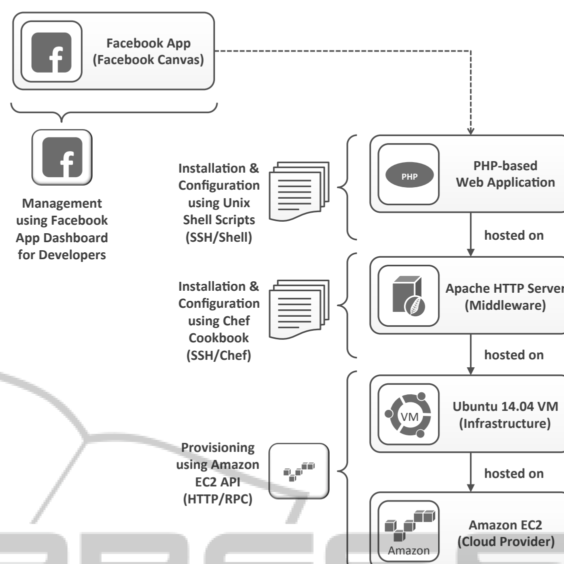

Figure 2: Facebook application stack.

To provision the complete application stack in an automated manner, different types of interfaces and invocation mechanisms have to be integrated. The virtual machine with its operating system is acquired by using the HTTP/RPC API provided by Amazon EC2. A Chef cookbook is executed on the virtual machine through an SSH connection to install the middleware of the application stack (Apache HTTP server). Furthermore, SSH is used to run custom Unix shell scripts to install and configure the actual Web application. However, remotely running executables such as Chef cookbooks and Unix shell scripts is not as straightforward as calling a well-defined API endpoint: (i) an executable needs to be placed on the virtual machine, e.g., using file transport protocols such as FTP and SCP. Moreover, (ii) the executable may require a particular runtime environment to be installed on the virtual machine such as a Chef runtime for Chef cookbooks. An SSH connection can be used to drive

[13]fog: http://fog.io
[14]Google APIs Client Libraries: http://goo.gl/uVvFf
[15]restler: https://github.com/danwrong/restler
[16]node-soap: https://github.com/vpulim/node-soap
[17]rest-client: https://github.com/rest-client/rest-client
[18]Savon: http://savonrb.com
[19]Facebook canvas frame: http://goo.gl/5guKas
[20]Amazon EC2: http://aws.amazon.com/ec2

the installation. Afterward, (iii) the execution of the scripts needs to be parameterized, which may be done by setting environment variables or storing configuration files. The final challenge is (iv) retrieving the results of the invocation, e.g., by reading, parsing, and potentially transforming the console output or files that were written to disk. In comparison to a simple API call, these steps are more complex and error-prone because lower-level implementation details such as different transport protocols and invocation mechanisms have to be considered and combined with each other. The overarching provisioning logic orchestrating all API calls as well as the preparation and invocation of the executables could be implemented by a script using a general-purpose scripting language such as Ruby or Python. However, such a script would be polluted with lower-level implementation details such as establishing SSH connections and placing files on the virtual machine. Furthermore, service composition languages such as BPEL or BPMN cannot be used without manually creating wrapping logic for the different executables involved. This is due to their focus on Web service orchestration. Consequently, the implementation details of the underlying APIs and executables directly influence which orchestration approaches can be used. This clearly contradicts with the idea of loose coupling, i.e., selecting an orchestration approach and implementing the orchestration logic without considering the implementation details of the underlying, lower-level technologies.

To tackle these challenges we propose an automated approach to generate APIs for arbitrary executables. The approach is based on the *APIfication method* we present in Section 3. In the context of our motivating scenario discussed in this section, the approach can be used to completely wrap the script invocation by generating an API that hides the (i) placement, (ii) installation of required runtime environments, (iii) parameterization and execution of the executable, as well as (iv) transforming and returning the results. Consequently, the orchestration logic deals with API calls only, without getting polluted, error-prone, or unnecessarily complex because of implementation details of the underlying executables.

3 APIfication METHOD

The APIfication approach presented in this section is based on the assumption that each executable has some metadata associated with it. These metadata are either natively attached and/or they are explicitly specified and additionally attached to the executable. Metadata indicate which input parameters are expected, where results are put, which dependencies have to be resolved before the invocation, etc. The main purpose of a generated API implementation is to enable the invocation of the corresponding executable through a well-defined interface, independent from the underlying technology stack. Furthermore, a generated API implementation enables the invocation of the corresponding executable not only locally in the same environment (e.g., same server), but enables the execution using remote access mechanisms such as SSH and PowerShell in remote environments. This is to decouple the environment of an API implementation instance from the environment of the actual executable that is exposed by the API. Distributed environments as they are, for instance, used in the field of Cloud computing are thereby supported. An API call could be made from a workstation (running a script that orchestrates multiple APIs) to an API implementation instance that is hosted on premises (e.g., a local server); the actual executable (e.g., a Chef cookbook to install a middleware component) runs on a Cloud infrastructure. However, one could also run all parts on a single machine, e.g., a developer's laptop.

Figure 3 shows an overview of the *APIfication method*, outlining the individual steps and their ordering to generate API implementations in an automated manner. In the **first step,** the executable targeted for the APIfication is selected. Then, the interface type (e.g., RESTful API) and the API implementation type (e.g., Node.js or Java) is selected **(step 2 & 3).** The type of interface including the communication protocol (HTTP, WebSocket, etc.) and the communication paradigm (RPC, REST, etc.) can be chosen when generating an API implementation. This choice may be driven by existing expertise, alignment with existing APIs, or personal preferences. Similarly motivated, the type of the underlying implementation (Java, Node.js, etc.) for the generated API can be chosen when generating an API implementation. A generated API should be language-agnostic to allow the usage of arbitrary languages (scripting languages, programming languages, service composition languages, etc.) to orchestrate and integrate different APIs. Thus, Web APIs are the preferred and universal type of APIs because they can be utilized in nearly any kind of language.

After the selection part, the executable including its metadata is scanned to discover input and output parameters **(step 4).** If the scan did not discover all parameters, the following (optional) step can be used to refine the input and output parameters for the generated API **(step 5).** However, this is not required if the metadata associated with the executable are sufficient as this is, e.g., the case for many open-source deployment automation artifacts such as Chef cookbooks and Juju charms. Consequently, the method can be applied

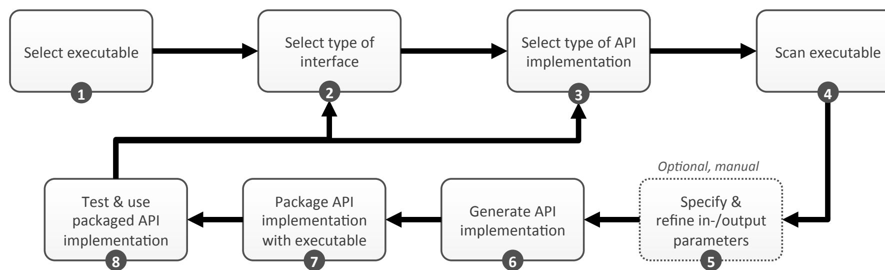

Figure 3: APIfication method.

to a huge amount and variety of such artifacts in an automated manner. Then, the API implementation is generated **(step 6).** To enable an API implementation to be hosted in different environments, it must be packaged in a portable manner **(step 7).** Thus, the implementation must be self-contained without depending on central middleware components, which dynamically provide data format transformations, parameter mappings, etc. at runtime. All these and related functionality are incorporated in the API implementation when it is generated at build time. The portability aspect is key for automated deployment processes because they need to run in very different environments (development, test, production, etc.). These environments may be hosted on different infrastructures (developer laptop, test cloud, etc.), so portability of the generated API implementations is key in this context. Technically, containerization technology (Scheepers, 2014; Turnbull, 2014) may be utilized for this purpose: each API implementation gets packaged as a portable container image that can be instantiated in different environments.

Later, the generated implementation may be refined or updated by going back to the selection steps for the interface type and the API implementation type. The APIfication method presented in this section addresses the challenges we identified in Section 2, including the deployment automation use case and the motivating scenario. However, the method itself is still abstract and can be implemented in various ways. The following section presents a modular and extensible framework to implement the APIfication method.

4 APIfication FRAMEWORK

In order to implement the APIfication method introduced in Section 3, we present a modular, plugin-based, and extensible framework in this chapter to support the individual steps of the method. Figure 4 shows several artifacts organized in multiple registries that are linked to the steps of the method, associated with certain actions (check, use, create). When selecting an executable for its APIfication, the available *invokers* are checked **(action A)** if there is at least one invoker available that is capable of running the given type of executable (e.g., a Chef cookbook). Figure 5 outlines the registry, in which the invokers are stored: each invoker supports at least one *executable type*. For instance, the *Cookbook Invoker* can be used to run Chef cookbooks. The generator registry (Figure 6) is checked **(action B)** when selecting the interface type and the API implementation type. As an example, a Chef cookbook may be selected in conjunction with `HTTP+REST` as interface type and `Node.js` as implementation type. In this case all checks would succeed because the *Cookbook Invoker* is available and the *REST API Generator* can be used to generate an `HTTP+REST` interface; this is possible because the chosen generator can deal with `Node.js` as implementation type. Consequently, the generator uses the invoker to provide an API implementation that can run the given Chef cookbook.

Next, the given executable with its metadata is analyzed by a corresponding *scanner* **(action C)** from the scanner registry (structured similarly as invoker registry) to create an *API I/O specification* **(action D).** A scanner is a specialized module in the framework that is able to scan executables of a certain type such as a *Chef cookbook scanner* to scan cookbooks. Figure 7 shows an example for a specification (produced by a scanner) for a MySQL cookbook: it contains the input and output *parameter names,* their *data types,* and the *mapping information* to properly map between API parameters to the executable parameters at runtime. The *mode* of a parameter indicates whether this parameter is used as input or it is used to return some output of an invocation. Optionally, a *default value* can be associated with a parameter, which is used in case no value is defined at runtime for the corresponding parameter. In case the data type is `object`, a schema definition, e.g., XML schema (World Wide Web Consortium (W3C), 2012) or JSON schema (Internet Engineering Task Force, 2013) can be attached to the parameter. This is to specify the expected data structure for values (objects) of a particular parameter in more detail. The mapping of parameters spec-

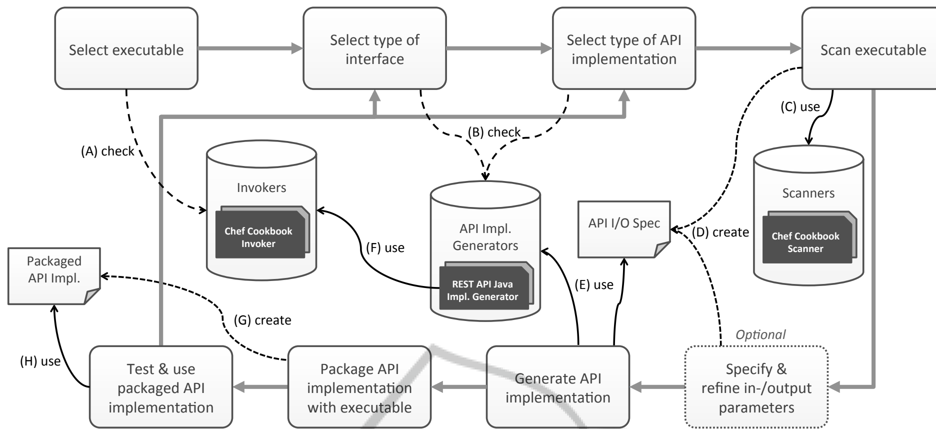

Figure 4: APIfication framework with technical examples.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Invoker	Executable Type
Cookbook Invoker	Chef Cookbook
Charm Invoker	Juju Charm
Docker Invoker	Docker Image
	Dockerfile
...	

Figure 5: Invoker registry.

ifies the target for input parameters and the source for output parameters at runtime. To refer to Figure 7: the API parameter `version` is mapped to the Chef attribute `mysql/version`, whereas the console output of the executable (`STDOUT`) is mapped to the API parameter `logs`. Optionally, the specification can be refined manually in the following step, which is not required if the executable's metadata is sufficient. The `invoker_config` parameter (mapped to the environment variable `INVOKER_CONFIG`) is a special one, provided by the framework; it cannot be modified or deleted during the (optional) manual refinement step. The parameter is used to configure the underlying invoker itself when using the generated API to run the executable. This is, for instance, needed to support remote access mechanisms, enabling the execution in remote environments. As an example the `invoker_config` parameter can hold the following JSON object to use SSH to run the executable remotely:

```
{
  "remote_access": "ssh",
  "remote_host": "173.194.44.88",
  "ssh_user": "ubuntu",
  "ssh_key": "-----BEGIN RSA PRIVATE KEY ..."
}
```

This sample configuration (given at runtime and transparently forwarded to the invoker) triggers the invocation of the underlying executable on the machine

Generator	Interface Type	Implementation Type
REST API Generator	HTTP+REST	Java
	HTTP+REST	Node.js
	HTTP+REST	Ruby
SOAP/WSDL API Gen.	HTTP+WSDL+SOAP	Java
	HTTP+WSDL+SOAP	Node.js
JSON-RPC API Gen.	HTTP+JSONRPC	Java
Node.js JSON-RPC API Gen.	HTTP+JSONRPC	Node.js
...		

Figure 6: Generator registry.

Param. Name	Mode	Data Type	Default	Param. Mapping
version	in	string	"5.1"	CHEF_ATTR:mysql/version
port	in	number	3306	CHEF_ATTR:mysql/port
logs	out	string	–	STDOUT
...				
invoker_config	*in*	*object*	–	*ENV:INVOKER_CONFIG*

Figure 7: API I/O spec for MySQL cookbook.

associated with the given IP address (`remote_host`) through SSH. Beside the special `invoker_config` parameter, the API I/O specification tells the corresponding *generator* how to create a proper API implementation **(action E).** A generator is a specialized module that performs the actual work to generate an API implementation. One part of the generation process is to put the corresponding invoker into the generated API implementation. The invoker is provided by the invoker registry to run the given executable **(action F).** Finally, the API implementation is packaged with the executable in a self-contained manner **(action G).** With this, the APIfication procedure for the given executable is finished, so the generated and packaged API implementation can be tested and used **(action H).** Figure 8 outlines the structure of a generated and packaged API implementation: the invoker (e.g., the cookbook invoker) is retrieved from the in-

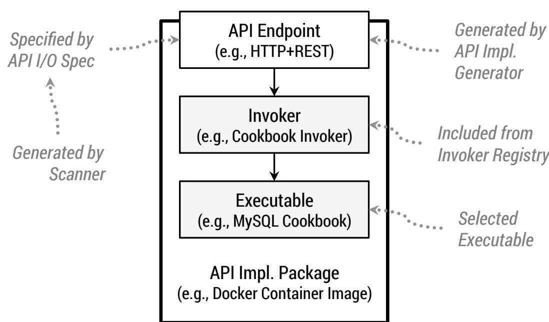

Figure 8: Generated API implementation package.

voker registry to invoke the selected executable such as the MySQL cookbook at runtime. The API endpoint is specified by the API I/O specification, which itself is generated by a scanner module provided by the framework. A generator module uses the specification to generate the implementation of the API endpoint. Finally, all parts are packaged in a self-contained manner, e.g., in a Docker container image. The following Section 5 presents the validation and evaluation of the APIfication method and framework we discussed in Section 3 and this section, based on a prototype implementation we provide.

5 VALIDATION & EVALUATION

In order to evaluate our APIfication method and framework, we developed ANY2API[21] as a prototype implementation. The following Section 5.1 presents and discusses the implementation. We performed experiments to measure the overhead both at build time and runtime (Section 5.2). Finally, Section 5.3 presents a comprehensive case study in the field of deployment automation.

5.1 ANY2API Implementation

ANY2API is a modular and extensible implementation of the APIfication framework presented in Section 4. Technically, it is based on Node.js, so most parts of it are implemented in JavaScript. Therefore, we use the Node Package Manager (NPM)[22] and the associated NPM registry to manage and publish Node.js modules. However, this does not imply that all parts of the framework have to be implemented in JavaScript. As an example, *invoker modules* expose several scripts that can (but do not have to) be implemented in JavaScript. Technically, these are specified as *NPM scripts*[23] in the `package.json` file of a module:

```
"scripts": {
  "prepare-executable": "node ./prep-exec.js",
  "prepare-runtime": "sh ./prep-runtime.sh",
  "start": "java -jar ./invoke.jar"
}
```

Such a script can then be called using the `npm run` command, e.g., to trigger an invocation of an executable that is packaged with a generated API implementation: `npm run start`. This command is executed by the generated API implementation, which itself can be of an arbitrary implementation type such as a JAR file (Java) or a Node.js module (JavaScript). Moreover, the API implementation needs to set predefined environment variables before running the script such as `PARAMETERS` to parameterize the invocation accordingly. These environment variables contain JSON objects that are parsed and processed by the invoker. As an example, the input parameters for invoking a MySQL cookbook may be rendered as follows:

```
{ "version": "5.1", "port": 3306 }
```

At build time (i.e., when generating an API implementation) the `prepare-executable` script is triggered to prepare the packaged executable. Such preparations may include resolving all dependencies of a particular executable to package the executable in a truly self-contained manner. At runtime (i.e., when an invocation of the executable is triggered) the `prepare-executable` script is executed before the `start` script to install prerequisites required for the invoker to run such as a Java runtime environment.

Generators and *scanners* are implemented as Node.js modules, too. Each *generator module* exposes a `generate` function to produce an API implementation based on the given API I/O specification. Each *scanner module* exposes a `scan` function, which analyzes the given executable to generate an API I/O specification. This specification (after optional, manual refinement) can then be used in conjunction with a generator to produce an API implementation. To actually use and interact with the framework, the `any2api-cli`[24] module provides a command-line interface (CLI) to scan executables as well as to generate packaged API implementations:

```
# any2api -o ./mysql_spec scan ./mysql_cookbook
# any2api -o ./api_impl gen ./mysql_spec
```

The first command scans an existing Chef cookbook, generating an API I/O specification. Based on this specification, the second command generates a corresponding API implementation. By default, a Node.js-based API implementation exposing a RESTful interface is generated. A `Dockerfile`[25] (build plan

[21] ANY2API: http://any2api.org
[22] NPM: https://www.npmjs.org
[23] NPM scripts: http://goo.gl/0ss4NL
[24] `any2api-cli`: https://github.com/any2api/any2api-cli
[25] `Dockerfile` reference: http://goo.gl/p5Tfdz

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

to create a self-contained and portable container image) is included in each generated API implementation. Consequently, Docker can be used to create API implementation packages. Moreover, public and private Docker registries[26] can be utilized to store, manage, and retrieve potentially different versions of pre-built API implementations. Following this approach, a huge variety of existing tools that are part of the Docker ecosystem can be used to manage instances of generated API implementations. As an example, CoreOS[27] may be utilized to host API implementations in a managed cluster of Docker containers.

Currently, two *scanner modules* are implemented for analyzing Chef cookbooks and Juju charms. The *Chef invoker module* enables the invocation of Chef cookbooks, both in local and remote environments using SSH transparently. Using the *REST generator module,* Node.js-based RESTful API implementations can be generated. Further modules are currently being developed such as a Juju invoker, a Docker invoker, a Docker scanner, as well as alternative generators to support different type of interfaces (SOAP/WSDL, JSON-RPC, XML-RPC, etc.) and alternative implementation types (Java, Ruby, etc.).

5.2 Measurements

In order to evaluate the efficiency of our approach compared to the plain usage of the corresponding executable, we measured the overhead of the APIfication. Therefore, we generated API implementations for a selection of the most downloaded Chef cookbooks[28], covering the automated installation and configuration of very common and widely used middleware components, including `mysql`, `apache2`, `php`, `nginx`, `postgresql`, and others. As an example, `apache2` and `php` are required for the automated deployment of the Facebook application we outlined in the motivating scenario (Section 2.2). First, we measured the overall duration it takes to scan the executable (Chef cookbook) and to generate a corresponding API implementation (Node.js-based RESTful API). Second, we check the additional size of the generated API implementation without the corresponding executable. This is to estimate the disk space that is additionally required at runtime when using an instance of an API implementation. Third, we measured the execution duration and memory usage for running the corresponding executable both *with* and *without* using the generated API implementation. The evaluation was run on a clean virtual machine (4 virtual CPUs clocked at 2.8 GHz, 64-bit, 4 GB of memory) on top of the VirtualBox hypervisor, running a minimalist Linux system including Docker. The processing and invocation of a particular Chef cookbook was done in a clean Docker-based Ubuntu 14.04 container, with exactly one container running on the virtual machine at a time. We did all measurements at container level to completely focus on the workload that is linked to the executable and the API implementation.

Table 1 shows the results of our evaluation. The measured average duration to scan and generate an API implementation is in the range from 7 to 90 seconds. This duration is the overhead at build time, including the retrieval of the executable and all its dependencies. The additional size of the generated API implementation leads to slightly more disk space usage at runtime. Moreover, there is a minor overhead in terms of execution duration and memory consumption at runtime. In most of today's environments this overhead should be acceptable, considering the significant simplification of using the generated APIs compared to the plain executables. In addition, when using the plain executables directly, much of the complexity hidden by the generated API implementation has to be covered at the orchestration level. So, the overall consumption of resources may be the same or even worse, depending on the selected means for orchestration. Furthermore, instances of API implementations can be reused to run an executable multiple times and potentially in different remote environments. Through this reuse, the overhead can be quickly compensated in large-scale environments.

5.3 Deployment Automation Case Study

We used the presented APIfication approach to ease implementing and generating workflows for the deployment of Cloud applications based on the OpenTOSCA ecosystem (Binz et al., 2013; Kopp et al., 2013). This ecosystem is based on the TOSCA standard (Binz et al., 2014), which enables describing Cloud applications and their management in a portable fashion. To define management tasks imperatively, e.g., to migrate application components, the ecosystem employs management plans based on the workflow language BPEL (OASIS, 2007). Therefore, the orchestration of management scripts, APIs, and other executables is a major challenge. The presented APIfication approach eases developing management workflows significantly as it reduces the required effort and complexity of integrating different technologies. Using our approach, modeling management workflows requires the orchestration of APIs only, which is much more straightforward compared to the former integration of various hetero-

[26] Docker registry: http://goo.gl/2lgohL

[27] CoreOS: https://coreos.com

[28] Most downloaded cookbooks: http://goo.gl/8xZUCT

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 1: Measurements regarding generated API implementations for Chef cookbooks.

	mysql	apache2	java	nginx	zabbix	glassfish	postgresql	php	...
Avg. duration to scan and generate API impl.	13s	14s	7s	25s	90s	17s	16s	29s	
Add. size of generated API implementation	25M	25M	25M	25M	25M	25M	25M	25M	
*Avg. execution duration **with** API impl.*	54s	48s	84s	45s	47s	153s	60s	123s	
*Avg. execution duration **without** API impl.*	54s	39s	82s	39s	42s	140s	59s	110s	
*Max. memory usage **with** API impl.*	556M	471M	507M	461M	429M	674M	510M	614M	
*Max. memory usage **without** API impl.*	343M	258M	402M	270M	212M	456M	310M	426M	

geneous technologies. Combined with the generated APIs for Chef cookbooks as discussed in Section 5.2, the integration of both the ecosystem and our APIfication approach provides a powerful means to enable a fast development of management workflows for Cloud applications.

6 FURTHER USE CASES

Beside the deployment automation use case (Section 2) we were focusing so far, we identified further use cases to apply our APIfication approach presented in this paper. In the *cyberinfrastructure & e-science community* (Yang et al., 2011) scientific applications are utilized, orchestrated, and run in Grid and Cloud environments to perform complex and CPU-intensive calculations such as scientific simulations and other experiments. These applications are implemented in arbitrary programming or scripting languages; they are usually run as executables directly. Consequently, they cannot be directly utilized through APIs. Existing works focus on the usage of scientific applications through Web APIs (Afanasiev et al., 2013; Sukhoroslov and Afanasiev, 2014) to ease their integration and orchestration for more sophisticated experiments, where multiple scientific applications are involved. As an example, Opal (Krishnan et al., 2009) is a framework for wrapping scientific applications, so they can be used through a Web API, abstracting from the application-specific details and differences such as invocation mechanisms and parameter passing. We tackle these challenges with our work by generating API implementations and packaging them together with the actual scientific application, i.e., the executable. This eases the integration and orchestration of different scientific application through Web APIs, without having to create API wrappers manually from scratch. As a result, running complex experiments that involve several scientific applications becomes easier.

In the previously described use cases of deployment automation and e-science, we implicitly assumed an executable to be an individual file or a collection of files (scripts, compiled executables, scientific applications, etc.). However, existing API endpoints as they are, e.g., exposed by provider-hosted Cloud APIs and social media APIs (Facebook[29], Twitter[30], etc.) can be considered as executables, too. This is motivated by the need for wrapping existing API endpoints to make them available through different communication protocols (e.g., wrap WebSocket by HTTP) or communication paradigms (e.g., wrap RPC by REST). As an example, Twitter provides the *users/show* endpoint[31] to retrieve a variety of information about a particular Twitter user. If this API endpoint needs to be utilized in a deployment workflow implemented in BPEL, a wrapper has to be implemented to make the endpoint accessible through a WSDL/SOAP-based interface (Wettinger et al., 2014a). By treating API endpoints as executables, API implementations could potentially be generated for existing endpoints to make them accessible through different protocols and communication paradigms, without relying on central middleware components such as a service bus.

[29]Facebook Graph API: http://goo.gl/HKGpZG
[30]Twitter REST API: https://dev.twitter.com/rest
[31]Twitter *users/show* API endpoint: http://goo.gl/dmsJ22

7 RELATED WORK

As discussed in Section 1 and Section 2, using and creating APIs is of utmost importance today (Rudrakshi et al., 2014). Consequently, a huge variety of approaches is available to simplify the creation and development of APIs. Beside API development frameworks to create API implementations manually (e.g., Hapi[32] and LoopBack[33]), there are solutions to semi-automatically create Web APIs. As an example, API specifications defined using the *RESTful API Modeling Language (RAML)*[34] can be utilized to generate an API implementation skeleton based on Jersey[35], a Java framework to develop RESTful APIs (Masse, 2011; Richardson et al., 2013). These generated skeletons have to be refined by adding application-specific logic. Consequently, such approaches can be immediately used to develop *generator modules* for our APIfication framework: the generator produces a skeleton, which is then automatically refined by adding the logic to call a corresponding invoker to run the selected executable. Moreover, solutions such as Kimono[36] and Import.io[37] can be used to generate Web APIs for existing Web sites. These approaches provide interactive ways to extract content from HTML pages (e.g., using CSS selectors) to make them available in more machine-readable formats such as JSON. Thus, such Web page-centric approaches focus on extracting and re-formatting content, whereas our approach tackles the issue of managing the invocation of arbitrary executables. In contrast to service providers such as Kimono, our approach aims to generate self-contained, portable, and packaged API implementations that can be hosted anywhere, so they do not depend on specific provider offerings.

RPC frameworks such as Apache Thrift[38] and Google's Protocol Buffers[39] aim to ease the integration of application logic and executables that are implemented based on different technology stacks. For efficiency reasons, they typically do not rely on Web APIs but use lower-level TCP connection-based protocols. Such RPC frameworks can be perfectly combined with our APIfication approach by implementing *generator modules*. In this case, a module generates an API implementation, e.g., exposing a Thrift interface instead of an HTTP-based RESTful interface. Some of these frameworks offer support to generate code skeletons based on interface descriptions. This functionality can be reused to ease the implementation of a corresponding *generator module*. However, by sticking to such non-standard communication protocols there are limitations on the orchestration level, meaning the same framework has to be used instead of interacting with a standards-based interface such as HTTP/REST. This is a trade-off between efficiency and interoperability that needs to be made individually based on concrete use cases. Since our framework supports both approaches, different API implementations (e.g., Thrift-based and HTTP/REST-based) can be generated and exchanged for a particular executable as needed. In the field of Web APIs, approaches such as websockify[40] and websocketd[41] can be used to expose the functionality of executables through the standards-based WebSocket protocol (IETF, 2011). Corresponding *generator modules* can be implemented to reuse these approaches in the context of our APIfication framework.

[32]Hapi: http://hapijs.com
[33]LoopBack: http://loopback.io
[34]RAML: http://raml.org
[35]RAML to JAX-RS (Jersey): http://goo.gl/E39jun
[36]Kimono: https://www.kimonolabs.com
[37]Import.io: https://import.io
[38]Apache Thrift: http://thrift.apache.org
[39]Protocol Buffers: http://goo.gl/uq69p

8 CONCLUSION

In this paper we introduced an automated APIfication approach to ease the integration and orchestration of arbitrary executables. To fulfill the requirements derived from the deployment automation use case and the motivating scenario, we presented a generic APIfication method and a corresponding framework to automatically generate API implementations. In order to confirm the practical feasibility of the presented method and framework, we published ANY2API as a modular and extensible implementation. To analyze the efficiency of our approach, we conducted an evaluation with comprehensive measurements. The measured results show a small overhead when following the APIfication approach, which is acceptable for most use cases, considering the significant simplification and convenience that our approach provides. In addition, we did a case study in the field of deployment automation to confirm the actual applicability of our approach in practice. Finally, we outlined additional use cases in different fields to apply the proposed APIfication approach.

In terms of future work, we are going to extend the APIfication framework to support an additional but optional step to refine the parameter mapping (e.g., aggregating, splitting, or transforming parameter values). For this reason we intend to enable the definition of JavaScript functions that are executed in a sandboxed

[40]websockify: https://github.com/kanaka/websockify
[41]websocketd: https://github.com/joewalnes/websocketd

environment at runtime. Moreover, we plan to extend and refine the ANY2API implementation. Existing scanners, generators, and invokers will be refined, and additional ones will be implemented. As an example, refinement may include authentication and authorization mechanisms for generated API implementations. The currently implemented generators can be used to create API implementations that expose Web APIs such as HTTP/REST. In future, we plan to implement generators in conjunction with alternative packaging formats to generate API libraries that can be directly used in certain programming and scripting languages such as Java and Python.

ACKNOWLEDGEMENTS

This work was partially funded by the BMWi project *CloudCycle* (01MD11023) and the DFG project *SitOPT* (610872).

SCIENCE AND TECHNOLOGY PUBLICATIONS

REFERENCES

Afanasiev, A., Sukhoroslov, O., and Voloshinov, V. (2013). MathCloud: Publication and Reuse of Scientific Applications as RESTful Web Services. In *Parallel Computing Technologies*. Springer.

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., and Wagner, S. (2013). OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing*, LNCS. Springer.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, pages 527–549. Advanced Web Services. Springer.

Guinard, D., Trifa, V., and Wilde, E. (2010). A Resource Oriented Architecture for the Web of Things. In *Internet of Things (IOT), 2010*. IEEE.

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

Hüttermann, M. (2012). *DevOps for Developers*. Apress.

IETF (2011). The WebSocket Protocol.

Internet Engineering Task Force (2013). JSON Schema.

Kopp, O., Binz, T., Breitenbücher, U., and Leymann, F. (2013). Winery - A Modeling Tool for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing*, volume 8274 of *LNCS*. Springer Berlin Heidelberg.

Krishnan, S., Clementi, L., Ren, J., Papadopoulos, P., and Li, W. (2009). Design and Evaluation of Opal2: A Toolkit for Scientific Software as a Service. In *World Conference on Services I*. IEEE.

Masse, M. (2011). *REST API Design Rulebook*. O'Reilly Media, Inc.

Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*.

Nelson-Smith, S. (2013). *Test-Driven Infrastructure with Chef*. O'Reilly Media, Inc.

OASIS (2007). Web Services Business Process Execution Language (BPEL) Version 2.0.

OMG (2011). Business Process Model and Notation (BPMN) Version 2.0.

Pepple, K. (2011). *Deploying OpenStack*. O'Reilly Media.

Richardson, L., Amundsen, M., and Ruby, S. (2013). *RESTful Web APIs*. O'Reilly Media, Inc.

Rudrakshi, C., Varshney, A., Yadla, B., Kanneganti, R., and Somalwar, K. (2014). API-fication - Core Building Block of the Digital Enterprise. Technical report, HCL Technologies.

Sabharwal, N. and Wadhwa, M. (2014). *Automation through Chef Opscode: A Hands-on Approach to Chef*. Apress.

Scheepers, M. J. (2014). Virtualization and Containerization of Application Infrastructure: A Comparison.

Sukhoroslov, O. and Afanasiev, A. (2014). Everest: A Cloud Platform for Computational Web Services. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science*. SciTePress.

Turnbull, J. (2014). *The Docker Book*. James Turnbull.

W3C (2007). SOAP Specification, Version 1.2.

Wettinger, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., and Zimmermann, M. (2014a). Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science*. SciTePress.

Wettinger, J., Breitenbücher, U., and Leymann, F. (2014b). Standards-based DevOps Automation and Integration Using TOSCA. In *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC)*.

World Wide Web Consortium (W3C) (2012). XML Schema.

Yang, X., Wang, L., and Jie, W. (2011). *Guide to e-Science*. Springer.