Institute of Architecture of Application Systems

Developing, Deploying, and Operating Twelve-Factor Applications with TOSCA

Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{wurster, breitenbuecher, falkenthal, leymann}@iaas.uni-stuttgart.de

Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann. 2017. Developing, Deploying, and Operating Twelve-Factor Applications with TOSCA. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services, Salzburg, Austria, December 4-6, 2017* (iiWAS'17), 519-525. https://doi.org/10.1145/3151759.3151830

BibTeX:

```
@inproceedings{Wurster2017_12FactorTOSCA,
  author    = {Michael Wurster and Uwe Breitenb{\"u}cher and
               Michael Falkenthal and Frank Leymann},
  title     = {Developing, Deploying, and Operating Twelve-Factor
               Applications with TOSCA},
  booktitle = {Proceedings of the 19\textsuperscript{th}
               International Conference on Information Integration
               and Web-based Applications \& Services},
  year      = {2017},
  pages     = {519--525},
  publisher = {ACM},
  doi       = {10.1145/3151759.3151830}
}
```


Universität Stuttgart
Germany

Developing, Deploying, and Operating Twelve-Factor Applications with TOSCA

Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann
Institute of Architecture of Application Systems, University of Stuttgart, Germany
[lastname]@iaas.uni-stuttgart.de

ABSTRACT

With Cloud Computing, offering and delivering services over the Internet became commonly feasible. This has impacts on application design, development as well as on the automation of application provisioning. The Twelve-Factor App is a methodology that documents best practices for building and operating scalable, maintainable, and portable web-based SaaS applications. However, a standards-based approach to build, release, and run Twelve-Factor Apps independently of individual cloud providers and specific deployment technologies is missing, which quickly leads to a vendor or technology lock-in. In this paper, we introduce a guideline to establish a development process using the Twelve-Factor App methodology together with the OASIS standard TOSCA to address this issue. We show how to realize the twelve factors with TOSCA and how the approach supports portability and automated deployment.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; • **Computer systems organization**;

KEYWORDS

Cloud Computing, Twelve-Factor App, TOSCA

ACM Reference Format:
Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann. 2017. Developing, Deploying, and Operating Twelve-Factor Applications with TOSCA. In *iiWAS '17: The 19th International Conference on Information Integration and Web-based Applications & Services, December 4–6, 2017, Salzburg, Austria.* ACM, New York, NY, USA, 7 pages.
https://doi.org/10.1145/3151759.3151830

1 INTRODUCTION

Cloud Computing is widely used in industry and academia [29]. As a service consumer, one can benefit from Cloud Computing properties such as pay-per-use pricing, scalability, and self-service capabilities [14]. These properties also influence modern software development processes: The shift from traditional, non-iterative software development processes, such as the waterfall model, to iterative processes and agile methodologies, such as Scrum, is one driver to let companies strive for a continuous software delivery model [24, 26]. Furthermore, due to software development methodologies from DevOps, where the barrier between development and operations people is eliminated [11], this shift facilitates companies to establish practices and automated processes to deploy applications rapidly and continuously into their production environments. These aspects, including the automated deployment and management of applications, are key enablers to reduce costs of their operation [14]. However, these properties also have a significant impact on how applications need to be built in order to utilize the advantages provided by Cloud Computing: Applications need to be designed and developed considering certain requirements, such as scalability, maintainability, and portability. Developing applications fulfilling these requirements and being able to be provisioned and managed automatically is a non-trivial process, especially when independent components are developed in different teams that are distributed among different countries. The *Twelve-Factor App* emerged as a methodology that documents best practices for efficiently building such applications [28]. This methodology gets more and more adopted in industry and provides well-described best practices to achieve the properties for cloud-based deployments named above.

However, a standardized technology- and cloud provider-agnostic Twelve-Factor approach is missing. Thus, establishing the Twelve-Factor App as a methodology in the development process quickly leads to a long-term upfront commitment to a deployment technology and cloud provider. This means in effect that we *lock-in* our development process to a certain vendor using its own mechanisms, technologies, and processes to deploy and manage applications. A change in the used provider is, therefore, associated with high costs to adapt the application deployment process together with high migration costs for moving the complete production deployment to the new vendor's environment. Moreover, in DevOps-oriented organizations, it is a frequent practice to use a separate cloud platform for development and integration testing than for production. As a result, such organizations need experts for all used platforms and may need operation engineers to manage the different environments, which results in an immense complexity.

As there are currently only partial solutions, we tackle these issues by applying the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [20] to this problem: In this paper, we present a *guideline* describing how to use the TOSCA standard for developing Twelve-Factor Apps independently of concrete cloud providers and deployment technologies. We elaborate how each step in the development process of a Twelve-Factor App can be realized with TOSCA. Moreover, our guideline does not only focus on development but also tackles the challenges of automating deployment and management for different environments and deployment

iiWAS '17, December 4–6, 2017, Salzburg, Austria

ACM ISBN 978-1-4503-5299-4/17/12…$15.00
https://doi.org/10.1145/3151759.3151830

stages. The approach significantly reduces the complexity of development and operations as only one single, generic technology has to be used, which supports integrating arbitrary cloud providers and deployment technologies. The proposed guideline intends to help novices and non-experts of TOSCA to realize Twelve-Factor Apps using this standard by explaining how TOSCA's concepts can be used in an efficient manner to realize the twelve factors.

The remainder is structured as follows: Section 2 motivates the paper and gives an overview of TOSCA. Section 3 presents our guideline for using TOSCA to develop and operate Twelve-Factor Apps. Section 5 concludes and discusses future work.

2 BACKGROUND, MOTIVATION, AND TOSCA

This sections presents background information about Twelve-Factor Apps and TOSCA. Moreover, we introduce a motivating scenario.

2.1 The Twelve-Factor App Methodology

The Twelve-Factor App methodology establishes a shared vocabulary and a set of best practices for building applications that are self-contained, stateless, with explicitly declared and isolated dependencies, where configuration can be supplied through the environment [28]. Thus, the methodology enables the development of scalable, maintainable, and portable cloud-native applications.

Having such a methodology to build cloud-native applications is indeed a key part. The Twelve-Factor methodology can be applied to applications written in any programming language targeting any kind of cloud provider, whereas a Platform as a Service (PaaS) offering suites best for such applications since the complete middleware stack and runtime environment is offered and managed by the cloud provider [14]. Hence, it is reasonable that the authors of the Twelve-Factor methodology formulated these best practices based on their experience in development, operation, and scaling applications during their work on the Heroku platform—a PaaS cloud platform based on a managed container system[1].

However, nowadays, a lot more different PaaS offerings and technologies are provided and are widely used in the industry. For example, besides Heroku and among others, there is Google's App Engine[2] and AWS Elastic Beanstalk by Amazon Web Services[3]. Furthermore, Cloud Foundry[4] has emerged as a cloud provider independent development and deployment platform. By using Cloud Foundry, applications can be deployed to cloud providers supporting this platform without changing the application's deployment mechanism nor the application's source code. Thus, Cloud Foundry provides a solid base for building portable applications.

As a result, there are many different deployment technologies and cloud providers available that offer capabilities required for building, deploying, and operating applications following the Twelve-Factor methodology. However, using one of these technologies directly leads to a lock-in: Development, deployment, and operation processes are tightly coupled to the capabilities, features, and APIs offered by these technologies [4]. For example, if automated deployment scripts are used to provision new instances of an application, these scripts are tightly coupled to the APIs, data formats, features, and invocation mechanisms of the employed technology [6]. Thus, if a company decides to use one of these technologies or providers, a later change is directly associated with a huge effort for adapting the corresponding deployment models and scripts, which results in very high costs and requires immense technical expertise. Moreover, a private or public cloud deployment often ends up in a hybrid cloud deployment due to new laws, new available services, or changing compliance requirements. Thus, often multiple providers and technologies have to be integrated for a single deployment, which is a complex, error-prone, and time-consuming challenge. By using the TOSCA standard as basis for the development and operation of a Twelve-Factor App, we can increase the level of portability and exchangeability of technologies [3]. With TOSCA we can use a complete vendor-neutral ecosystem enabling us to build portable and interoperable cloud applications, independent of any cloud provider-specific API, domain-specific language (DSL), or deployment technology. However, how to use the concepts of TOSCA to realize the twelve factors of the methodology is highly non-trivial. Therefore, we present a detailed guideline in this paper.

2.2 Motivating Scenario

A typical scenario in modern software development companies, influenced by DevOps [11] and the architectural style of microservices [13], could be formulated as follows: A software development team has the charter to develop and operate the front-end and back-end of a social blogging site. In the upcoming development iterations, the back-end application should be re-implemented and deployed to a public cloud provider. For the sake of brevity we simplify the functional and non-functional requirements for the back-end: (i) As a user, I want to use a RESTful HTTP API to list, create, and update articles, to comment on articles, and to follow other users, (ii) a follower request is published through a PubSub (Publish-Subscribe) message broker to the application, (iii) the application component must scale horizontally, and (iv) must be portable, so that it can be deployed to different execution environments. As a company guideline, Amazon AWS is chosen as cloud platform for newly developed applications. Moreover, since the team mainly consists of Java experts, the team decides to utilize the widely used Spring Framework and the framework additives of Spring Boot[5]. However, the team has to deal with a legacy front-end application which serves the user interface and is hosted on-premise with OpenStack. In such a scenario, the rapid provisioning and deployment of applications becomes a vital part [11]. On the one hand, the team has to be able to adapt computing resources quickly and in an automated manner. On the other hand, it has to be able to quickly and automatically deploy the application components to different execution environments, which implies a close collaboration between software developers and operation engineers [9].

For deploying the back-end application into production, AWS CloudFormation[6] can be used, which is a DSL supported by AWS for deploying applications on different AWS services. To automatically deploy the front-end application, expertise on OpenStack is required. On top of that, a third technology is required as the

[1] Heroku: https://www.heroku.com
[2] Google App Engine: https://cloud.google.com/appengine
[3] AWS Elastic Beanstalk: https://aws.amazon.com/de/elasticbeanstalk
[4] Cloud Foundry: https://www.cloudfoundry.org
[5] Spring Boot: https://projects.spring.io/spring-boot
[6] AWS CloudFormation: https://aws.amazon.com/cloudformation

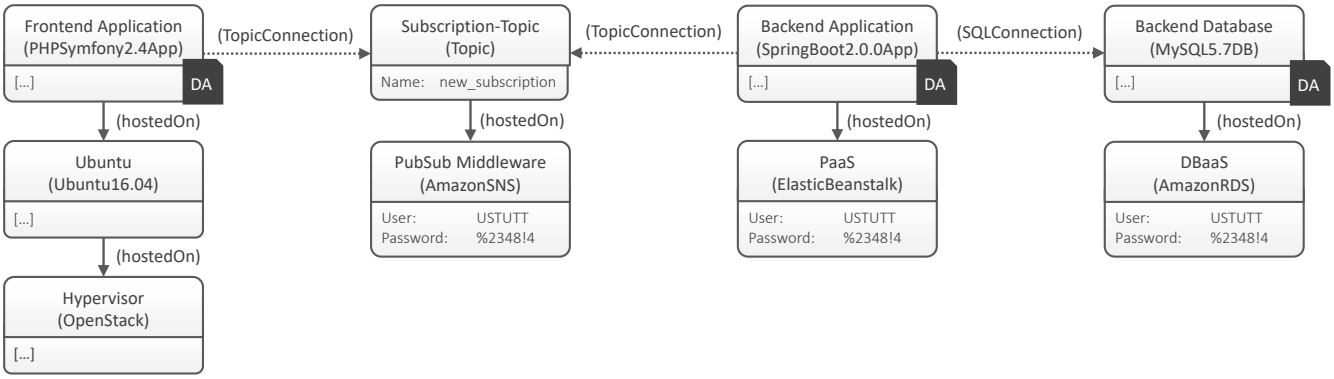


Figure 1: Motivating scenario modeled as simplified TOSCA Topology Template.

overall orchestration layer to combine and control the deployment of these two applications. Furthermore, as the back-end application needs to be properly tested, there must be the possibility to deploy the whole application landscape into the company's local test infrastructure. In order to achieve this, we have to employ different kinds of deployment technologies that require different kinds of expertise. Moreover, if the company decides to change the cloud provider or to combine services of different providers, then additional deployment technologies have to be used and integrated [6]. Thus, we end up in the issues discussed previously.

2.3 The TOSCA Standard

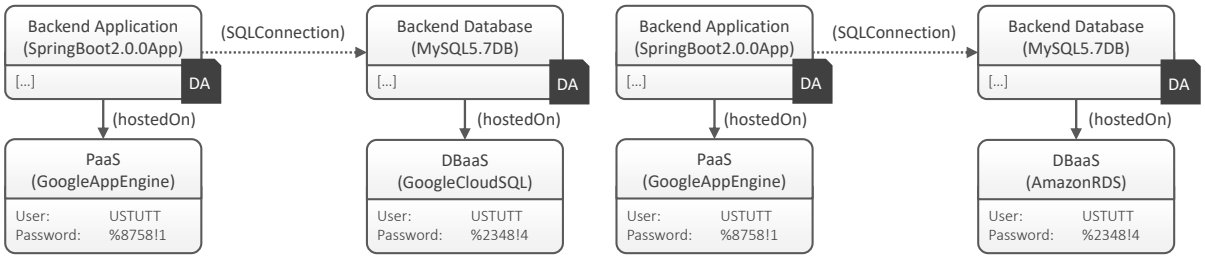
The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an OASIS standard that enables modeling, provisioning, and management of cloud applications [2, 20]. TOSCA enables modeling the structure of an application to be deployed as a directed graph called *Topology Template*, which consists of Node Templates (vertices) and Relationship Templates (edges). *Node Templates* represent components of an application such as virtual machines, web servers, or software components. *Relationship Templates* represent the relations between nodes, e. g., that a node is hosted on or connects to another node. Node and Relationship Templates are typed: *Node Types* and *Relationship Types* are reusable classes that define the semantics of the corresponding template. For example, a Node Template can be of type "ApacheTomcat" while a Relationship Template can be of type "hostedOn", which specifies that the source node shall be hosted on the target node. Node Types define properties that can be used to configure the deployment. For example, the "ApacheTomcat" Node Type may specify the properties "HTTP Port" and "Username", which are filled with concrete values by the Node Template. Node and Relationship Types may also specify *Management Operations*, e. g., a "create" operation for installing the Tomcat web server. *Implementation Artifacts* (IAs) provide the implementation for these operations, for example, the "create" operation could be implemented as a Shell script. TOSCA standardizes a *Lifecycle Interface* [21], which specifies that the operations *create*, *configure*, *start*, *stop*, and *delete* are executed in this order for each Node Template. Thus, based on this Lifecycle Interface, arbitrary installation and configuration logic can be specified.

Deployment Artifacts (DAs) implement the application's functionality and can be attached to Node Types and Node Templates, e. g., a DA of the "ApacheTomcat" Node Type could be the binary files of the web server. Based on these model elements, the entire structure of an application can be described. Moreover, Node Types can be defined arbitrarily, thus, any provider and platform technology can be specified as component within a topology in a generic manner.

Management Plans are executable process models implementing management functionality for the specified topology. For example, a Management Plan can be modeled for the initial provisioning of the application—these *provisioning plans* can be generated automatically based on the Topology Template [4]. Management Plans are recommended to be implemented as workflows [16, 20], for example, using BPEL [19]. Thus, based on this concept, arbitrary management functionality can be automated by such plans.

TOSCA specifies an exchange format called *Cloud Service Archive* (CSAR) to package Topology Templates, Node and Relationship Types, IAs and DAs, plans, and all required files into one self-contained archive. This package is portable across different standards-compliant *TOSCA Runtime Environments*, which are used to deploy and manage applications modeled in TOSCA.

Figure 1 shows a simplified Topology Template of the motivating scenario. It shows the production environment consisting of a Node Template for the legacy front-end application of Node Type "PHPSymfony2.4App" hosted on-premise on OpenStack and the back-end application of Node Type "SpringBoot2.0.0App", hosted on a Node Template of type "AWSElasticBeanstalk"—the PaaS offering of Amazon AWS. Both applications are connected through a topic which is in turn hosted on "Amazon SNS", Amazon's managed PubSub service. In addition, as persistence layer, the back-end application uses Amazon's database service "RDS" to host a MySQL 5.7 database, which is a fully managed SQL cloud data store. The application and database Node Templates also specify Deployment Artifacts: The DA of the application contains the respective implementation while the DA of the database specifies the database schema. Thus, this example shows that arbitrary types of software components and their deployment relationships can be described in a generic manner. Furthermore, Figure 2a depicts how the back-end application can be modeled and deployed to a different cloud provider by exchanging the respective Node Types, for example,



(a) Google Cloud Platform Deployment **(b) Multi-Cloud Deployment (GCP/AWS)**

Figure 2: Cloud Provider Deployment Alternatives of the Backend

Node Types supporting Google Cloud Platform. Even more, a developer can choose Node Types of different cloud providers in order to realize multi-cloud deployments as shown in Figure 2b. Therefore, TOSCA provides a suitable basis to automate deployments independently from concrete technologies.

3 A GUIDELINE FOR REALIZING THE TWELVE-FACTOR APPS WITH TOSCA

This section presents a guideline for realizing Twelve-Factor Apps using TOSCA as a vendor-neutral and technology-agnostic approach. Each section first describes one factor of the methodology and explains afterwards how the respective factor can be realized using the TOSCA standard following our guideline.

3.1 Codebase

A Twelve-Factor App must have one *codebase*, which is tracked in a version control system, such as Git or Subversion. This is a crucial factor because it must act as the single source of truth of your application's source code. The content of TOSCA's exchange format—Cloud Service Archive (CSAR)—shapes this codebase. A CSAR repository could contain multiple Topology Templates referencing application components (via Node Types) in a certain version as dependencies and targeting different deployment environments, such as development, testing, and production, as illustrated in Figure 3 (due to reasons of space, we only show the back-end part). By creating Topology Templates for certain target environments, we realize the "one codebase, many deploys" paradigm of the Twelve-Factor App. Doing so, we can maintain in one codebase the description how an application can be deployed to any number of environments. In fact, a Topology Template often contains multiple components (i. e., Node Types) and is the composition of a deployable application system. Node Types can be application-specific, but more often they characterize a common behavior that can be shared across multiple CSAR codebases. Such Node Types are practically dependencies for the CSAR codebase and are individually tracked in separate source code repositories.

3.2 Dependencies

Twelve-Factor applications must declare all *dependencies* explicitly and completely. All dependencies must be scoped only for a single application by bundling them together to an autonomous artifact before deploying to an execution environment. Such applications must never assume that any dependency is provided implicitly or system-wide in the deployment environment. Furthermore, dependencies of one component must be isolated from other component dependencies on the same computing resource. This factor is important to reduce the possibility of failed deployments where assumptions to a computing environment cannot be fulfilled. With the CSAR format, a self-contained exchange and packaging format is intended by the TOSCA standard itself. Thus, by creating Topology Templates, we can explicitly declare all dependencies (Relationship Templates) between all modeled components (Node Templates) in the topology model. With TOSCA we can, for example, represent which component is hosted on what computing resource and which runtimes must be installed on this resource through modeling Relationship Templates of type *requires*. Furthermore, we can model which component needs to connect to another component. Thereby, a TOSCA runtime is able to provision the application as modeled including all required dependencies. Moreover, TOSCA also provides the concept of *Requirements* and *Capabilities*, which can be used to model what a certain Node Template requires or provides, respectively. On top of that, TOSCA also enables specifying incomplete Topology Templates, in which Node Templates specify Requirements that have to be fulfilled during provisioning. TOSCA Runtimes are capable of resolving these *dependencies* by injecting appropriate Node Templates into the Topology Template during deployment that fulfill the specified Requirements by matching Capabilities, which ensures that a complete Topology Template, i. e., an autonomous artifact is processed and deployed. Thus, TOSCA provides extensive support for handling dependencies by the two possibilities of (i) packaging all required dependencies into the CSAR to get a self-contained exchangeable archive and (ii) explicitly specifying required dependencies that are not contained in the CSAR but have to be fulfilled during deployment.

3.3 Config

An application's *config*uration differs between different deployment environments, such as development, staging, or production. Therefore, configuration parameters must neither be checked into the application's source code repository nor be hardcoded as constants on programming language level. A naive approach with TOSCA is to maintain configuration files that are attached as DAs to the respective Node Templates. But this violates this factor since all configuration data should be stored in a separate place than the actual codebase, which is especially critical if configuration values

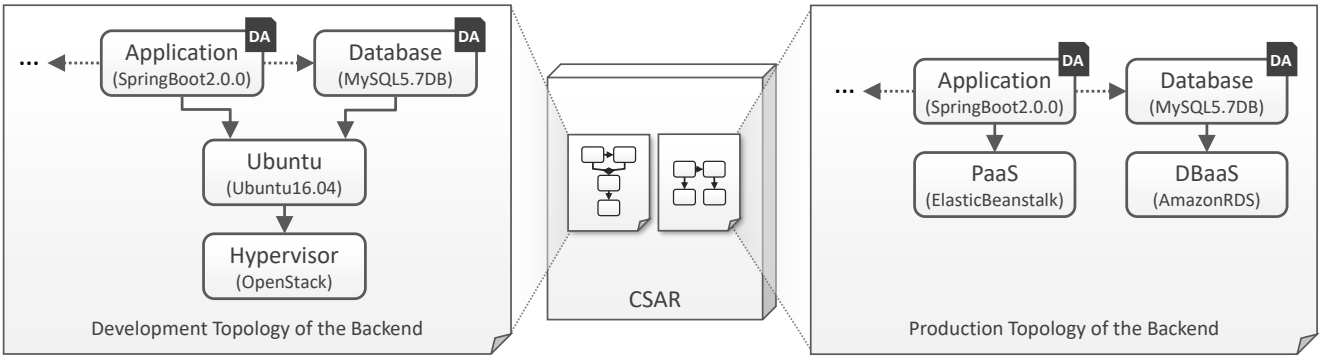


Figure 3: The "one codebase, many deploys" paradigm with TOSCA.

contain passwords. A better approach, and the recommended best practice of the Twelve-Factor App methodology, is to store configuration values in environment variables that are populated during deploy. With TOSCA, properties can be specified and attached to Node Templates to specify such environment variables. Thus, we can create arbitrary Node Types that specify an *install* operation, which is implemented by an IA that populates any property as a separate environment variable when deploying the respective component. Moreover, TOSCA provides support to specify that a property's value shall be requested from the user during deployment (`get_input`, see TOSCA's Simple Profile [21]). Thus, the system engineer can specify properties that shall be used as environment variables in two ways: (i) by providing the concrete value directly in the Node Template for information that is allowed to be contained in the codebase or (ii) by specifying that the property value shall be requested from the user when starting the deployment. Using this approach ensures that critical configuration information, such as login credentials, are never committed to the codebase as they are only provided when starting the deployment.

3.4 Backing Services

Any service an application consumes over the network as part of its normal operation is called a *backing service* by the Twelve-Factor App methodology, such as database or messaging services. Whereby it must not make any difference whether the service is locally available or provided by a third party provider. They are just attached resources, which are made available at the deployment of the application. This factor provides the development team great flexibility in, e. g., using a local installation of a database during development and one offered by a cloud provider for production. In TOSCA, we can express such attached resources with Node and Relationship Templates. We can use the Relationship Type "connectsTo" to model that one Node Template needs to connect to another one. Using the properties of the target Node Template that represents the backing services enables specifying all information required for connecting to the service. During deployment, these properties are made available as environment variables to the source Node Template to enable connecting it to the backing service. For achieving this, the mechanisms described in Figure 1 can be used for setting this configuration.

3.5 Build, Release, Run

In order to deploy a *codebase* into an execution environment, three stages need to be passed: (i) build, (ii) release, and (iii) run. The *build* stage transforms the codebase into a self-contained, exchangeable, and executable package. In TOSCA, at this stage we build the CSAR archive containing all required resources for a deployment. The *release* stage takes the output of the build stage and combines it with a configuration for a certain execution environment. This means to supply all unspecified properties of Node Templates. Finally, the *run* stage, also called runtime, runs the components in a certain execution environment by starting the required computing resources and processes. A TOSCA runtime uses the CSAR and the supplied properties to allocate computing resources, to populate environment variables, and to start and wire application processes.

3.6 Processes

An application must be executed in one or more stateless *processes*. This means that all data types that relate to state of an application have to be stored in backing services, typically in databases or blob storage resources. This is a vital factor since stateless applications are more robust, easier to manage, and generally easier to scale. In TOSCA, we can define the minimum and maximum number of instances to be created when instantiating a certain Node Template. Whereby, "instance" could mean in effect that a Node Template could launch a number of isolated processes. This implies that all DAs attached to Node Templates must be build individually according to the Twelve-Factor App methodology.

3.7 Port Binding

Twelve-Factor applications must expose services to other parties by binding to a port, and listening to requests coming in on that port. The idea is, like any other backing service you are consuming, that your application also interfaces to other applications using a simple URL. For web-based applications, this means, for example, that static HTML files must be served with a web server, such as Apache HTTP Server[7], or Java-based applications be served with a Servlet container, such as Apache Tomcat[8]. This implies that, based on this concept, applications can become backing services

[7] Apache HTTP Server: https://httpd.apache.org
[8] Apache Tomcat Server: http://tomcat.apache.org

of other applications, whether HTTP or any other communication protocol is used. In TOSCA we could bundle the runtime environment directly into each DA if required, depending on your style of deployment. For PaaS deployments, this approach makes less sense since with such a deployment model one already gets an appropriate runtime provided. However, with TOSCA we model and deploy our application to all kinds of deployment models. Any port configuration can be supplied through properties of a Node Template. We model in the Topology Template what kind of service is exposed on a certain port, however, how the service is provided is hidden and an implementation detail of the Deployment Artifact.

3.8 Concurrency

In Twelve-Factor Apps, *concurrency* is realized through the process model, where processes become first class citizens in that context. This means that a service provided by an application is served by one or more processes. This factor generally eases the way to scale an application by just starting a new process on the same or on a different resource. As stated in Section 3.6, in TOSCA we can model the minimum and maximum number of instances, e. g., processes, in the Topology Template that is instantiated by a runtime.

3.9 Disposability

Disposability is achieved by designing an application elastic and ephemeral, meaning in effect that we can adjust the application's performance by dynamically adding or removing computing resources whenever the workload of components change. On the one hand, we have to consider to start application processes in seconds and, on the other hand, we have to make sure that application processes are gracefully shutdown if they are no longer required. In turn, applications also should be robust against crashing. Meaning, if components crash they should always be able to start up again cleanly. Such management activities are expressed in TOSCA using Management Plans. They represent activities that can be executed during runtime of an application. By defining Management Plans we can, e. g., specify how to start and stop instances of a certain part of the topology. Again, this implies that all attached DAs are designed and implemented according the Twelve-Factor App methodology. Moreover, with TOSCA we can model Management Plans that describe which resources should scale, what a scaling plan should actually do, and when a scaling plan should be triggered.

3.10 Dev/Prod Parity

The idea of this practice is to close the gap between execution environments and keep them as similar as possible. This means, for example, to use the same backing services, the same versions of software components and libraries, and the same deployment techniques. However, there can be gaps in different areas: (i) the time gap, because it may take days or weeks until a source code change reaches the production stage, (ii) the personnel gap, due to a strict separation of developers and operations engineers, and finally, (iii) the tools gap, basing on the fact that developers may use totally different technology stacks in contrast to the running or targeted production environment. Whereas each area is equally important, we are focusing on the tooling area in this work. As the *codebase* practice states, the attached DAs in our Topology Template have to be the same for all deployment environments. With TOSCA we can create multiple Topology Templates representing different execution environments based on the same DAs. Then, we can package those Topology Templates into one ore more CSARs depending on our requirements. Furthermore, as depicted in Figure 2, with TOSCA we can easily target another cloud provider as execution environment by choosing different Node Types for the infrastructure layer. Finally, a TOSCA runtime then lets us launch a certain environment at any point in time once required.

3.11 Logs

Logs represent the behavior of a running application instance. As instances can be long-running, log entries must be treated as a continuous stream of time-ordered events. A Twelve-Factor App writes its logging information unbuffered to `stdout`. To implement this with TOSCA, each DA has to comply with this requirement. Moreover, an IA can be used that implements the TOSCA Lifecycle Interface to configure a log file adapter component, such as Logstash[9], to forward these information to a centralized log aggregation component. Moreover, lots of PaaS providers automatically enable the forwarding of log entries supplied through `stdout` into their log analytics service, such as Amazon CloudWatch[10]. As these PaaS offerings can be used as Node Types, this factor is nowadays often supported natively by the employed components.

3.12 Admin Processes

Administration tasks in Twelve-Factor Applications must be run in isolated one-off processes on identical environments as the production. This is an important factor so that management tasks are not executed manually, for example, directly against a database, or run from a developer's local terminal window. This is where the orchestration part of TOSCA comes into play and shows its strengths. Using TOSCA, we can specify Management Plans for such *admin processes* and activities. We can define multiple plans on Node Templates or Node Types to enable all kinds of management use cases—from simple ones, like start, stop, or restarting instances, to more complex ones, like executing migration or scaling workflows for certain parts of the application topology.

4 RELATED WORK

The Twelve-Factor App methodology was formulated based on experiences in building SaaS-based web applications and emerged to a widely accepted methodology in industry [28]. The principles are formulated as generic as possible so that they can be applied to any programming language and any type of service. With the advent of microservices [13, 18] and the driver to build cloud-native applications [15], the principles identified in the Twelve-Factor App methodology became the de-facto standard [1, 7].

With Docker and other container virtualization technologies we can achieve a certain level of portability and can manage applications in a loosely coupled and isolated manner with clear dependencies [10, 17, 22, 23, 30]. Cloud Foundry, as provider-independent development and deployment platform also provides a solid base for

[9] Logstash: https://www.elastic.co/products/logstash
[10] Amazon CloudWatch: https://aws.amazon.com/de/cloudwatch

building portable cloud applications. Such a PaaS platform supports the Twelve-Factor App methodology and provides the required functionality and features to implement the principles and best practices [8, 25]. However, to operate cloud-native applications and to eliminate vendor lock-in, a portable deployment and management approach is required [12]. Furthermore, there are a challenges with such approaches to implement multi-cloud or hybrid cloud deployments. The concepts of TOSCA tackle these issues by providing an extensible type system that allows the modeling of arbitrary types of application components and relationships. Using TOSCA, we can model such applications on top of multiple technology stacks, such as Cloud Foundry, Docker, or even on top of bare metal servers, possibly distributed over multiple cloud providers.

It is also possible to use other management tools, such as Ansible or Chef, instead of TOSCA for the automated deployment of computing resources. Furthermore, cloud platform specific tools such as AWS CloudFormation or OpenStack Heat allow as well to create, configure, modify, and terminate cloud computing resources. Moreover, the orchestration approach by Terraform[11] could be utilized which provides tooling to operate on a higher abstraction level. Having a provider-agnostic DSL, Terraform enables developers to deploy applications to multiple cloud providers, including multi-cloud provisioning scenarios. However, TOSCA is an official standard and tooling support is improving steadily. In contrast to these approaches, TOSCA enables a holistic, technology-agnostic approach based on topology models and a corresponding graphical notation [5]. These topology models are highly adaptable in a way that components can be easily interchanged, such as that we can combine many technology stacks with many cloud provider offerings. For example, by using an official standard, a company is able to exchange a TOSCA-compliant runtime with another one [27]. Thus, with TOSCA an orchestration and provisioning layer is provided which is completely vendor-independent and technology-agnostic in order to develop Twelve-Factor applications.

5 CONCLUSION AND OUTLOOK

In this paper, we described a guideline to develop, deploy, and operate Twelve-Factor application systems by means of the OASIS standard TOSCA. We can significantly increase the level of portability and exchangeability having positive impact on operation costs of an application system by combining the Twelve-Factor App methodology with TOSCA's interoperable metamodel. The proposed guideline supports developers and operation engineers in building portable and interoperable Twelve-Factor Apps. This paper showed that all of the twelve factors can be implemented with TOSCA and thus TOSCA is very well suited to develop and operate Twelve-Factor Apps. We showed that TOSCA is vendor-neutral and technology-agnostic, something that many other approaches, technologies, and platforms cannot provide. Therefore, this enables to develop, package, and deploy Twelve-Factor Apps as well as to automate operation and management processes.

As future work, we aim to incorporate concepts from DevOps to our approach, e. g., to establish a build and deployment pipeline to continuously integrate and deliver such applications by making use of TOSCA.

[11] Terraform: https://www.terraform.io

Acknowledgments. This work is partially funded by the BMWi project SePiA.Pro (01MD16013F) as part of the Smart Service World.

REFERENCES

[1] Kapil Bakshi. 2017. Microservices-based software architecture and approaches. In *2017 IEEE Aerospace Conference*. IEEE, 1–8.
[2] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2014. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*. Springer.
[3] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier. 2012. Portable Cloud Services Using TOSCA. *IEEE Internet Computing* 16, 03 (May 2012).
[4] Uwe Breitenbücher et al. 2014. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *International Conference on Cloud Engineering (IC2E 2014)*. IEEE.
[5] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and David Schumm. 2012. Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In *On the Move to Meaningful Internet Systems: OTM 2012 (CoopIS 2012)*. Springer.
[6] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and Johannes Wettinger. 2013. Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In *On the Move to Meaningful Internet Systems: OTM 2013 Conferences (CoopIS 2013)*. Springer.
[7] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. 2015. The making of cloud applications: An empirical study on software development for the cloud. In *10th Joint Meeting on Foundations of Software Engineering*. ACM.
[8] Tim Evko. 2015. The 12-Factor Apps Methodology: Implement It in Your Own Apps with AppFog. (2015). https://www.sitepoint.com/12-factor-apps-methodology-implement-apps-appfog
[9] Martin Fowler. 2014. Microservice Prerequisites. (2014). https://martinfowler.com/bliki/MicroservicePrerequisites.html
[10] Kelsey Hightower. 2015. 12 Fractured Apps. (2015). https://medium.com/@kelseyhightower/12-fractured-apps-1080c73d481c
[11] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education.
[12] Nane Kratzke and Rene Peinl. 2016. ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects. In *20th International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE.
[13] James Lewis and Martin Fowler. 2014. Microservices - A definition of this new architectural term. (2014). http://martinfowler.com/articles/microservices.html
[14] Frank Leymann. 2009. Cloud Computing: The Next Revolution in IT. In *52nd Photogrammetric Week*. Wichmann Verlag.
[15] Frank Leymann, Uwe Breitenbücher, Sebastian Wagner, and Johannes Wettinger. 2017. *Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation*. Springer.
[16] Frank Leymann and Dieter Roller. 2000. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR.
[17] Adrian Mouat. 2015. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Inc.
[18] Sam Newman. 2015. *Building Microservices*. O'Reilly Media, Inc.
[19] OASIS. 2007. *Web Services Business Process Execution Language Version 2.0*.
[20] OASIS. 2013. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*.
[21] OASIS. 2015. TOSCA Simple Profile in YAML. (2015).
[22] Cody A. Ray. 2015. How to Build 12 Factor Microservices on Docker. (2015). https://www.packtpub.com/books/content/how-to-build-12-factor-design-microservices-on-docker-part-1
[23] Ryan Schultz. 2015. Twelve-Factor Apps and Containers. (2015). http://blog.grio.com/2015/08/twelve-factor-apps-and-containers.html
[24] Ken Schwaber and Mike Beedle. 2002. *Agile Software Development with Scrum*. Prentice Hall Upper Saddle River.
[25] Stephen Spector. 2015. Using Twelve-Factor App Methodologies in Cloud Foundry. (2015). https://community.hpe.com/t5/Grounded-in-the-Cloud/Using-Twelve-Factor-App-Methodologies-in-Cloud-Foundry/ba-p/6710871
[26] Kalpana Sureshchandra and Jagadish Shrinivasavadhani. 2008. Moving from Waterfall to Agile. In *Agile 2008 Conference*. IEEE, 97–101.
[27] Johannes Wettinger, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2016. Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel. *Future Generation Computer Systems* 56 (2016).
[28] Adam Wiggins. 2012. The Twelve-Factor App. (2012). https://12factor.net
[29] Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications* 1, 1 (2010).
[30] Noah Zoschke. 2015. Modern Twelve-Factor Apps With Docker. (2015). https://medium.com/@nzoschke/modern-twelve-factor-apps-with-docker-55dd92c832b3