The Multitarget Application Model

Version 3.3.0, August 2021

THE BEST RUN SAP®

Contents

Conventions: For all descriptor samples, **bold-blue font** is used to indicate keys defined by descriptor schemas. **Bold-black font** is used for other map and sequence keys that are specific to the MTA model at hand (for example, property names). Regular-black font is used for values assigned to keys.

Introduction

Motivation

Modern complex business applications are composed of multiple parts developed using various programming languages and technologies and deployed to a variety of target runtime environments. Recent trends and progress in programming language development, software design architectures - such as micro-services, protocols like OData, and the diversity of multi-tiered and distributed deployment platforms - further accelerate the trend towards applications constructed out of more, smaller, decoupled, and diverse modules.

This introduces many lifecycle challenges. Developing, deploying, and configuring all complex application parts involves many steps, typically target-platform or application-server specific. Required services must be pre-configure and provisioned; the various modules connected, configured, and deployed across multiple platforms in a strictly specific order - while often using various tools, repeated for testing, staging, and production environments. Zero-downtime upgrades introduce even more complexity.

We coin the term *Multitarget Application* (MTA) to express this diversity of lifecycle management requirements, and because other terms such as “plain” application, or “distributed”, “polyglot”, “multimodule”, “multitier”, or “multiheaded” application do not capture this diversity. But in essence, MTAs are just a natural *evolution* of existing multi-part applications.

For example, the SAP HANA “native application”, which consists of UI and database modules, and even application code, can be considered a HANA-based MTA. The now deprecated HANA repository resolved the deployment dependencies between these modules. The deprecated HANA Delivery Unit (DU) was a proprietary HANA XS1 archive. This solution had to evolve to support standard application languages and decoupled platforms, while preserving the smooth developer experience.

Another example of a typical MTA is the Java EE application, composed of beans, web and application modules, resource adapters, among others - all of which are subject to the same development lifecycle and deployed across multiple computing tiers.

SAP’s SaaS “extension platform” (Neo) and its Fiori as a Service (FaaS) concept introduce new distribution requirements for orchestrated cross-platform deployments. Application developers need to distribute their applications across heterogeneous target platforms (cloud platforms, on-premise, desktops, mobile devices), each with their own operations, infrastructure, and principles, while providing a carefully managed single application lifecycle.

An increased focus on micro-service design principles, API management and the emergence of the OData protocol as a rich service-UI edge are further encouraging the proliferation of application modules, developed with different languages, IDEs, and build methodologies.

But all these parts, UIs, services, and data models, must still run as a *coherent* application. When it comes to *deployment*, the uniformity is limited. Each runtime, application server, or cloud framework manages multitarget aspects in its own unique approach, by introducing a variety of orchestration solutions (a multitude of manifest files and formats, project JSON files, app descriptors, repositories, SAP's CTS+, among others).

Cloud platforms, such as Cloud Foundry and the Google Cloud Platform, improve over traditional application servers by providing flexible support for diverse application runtime technologies, but solutions are still limited to a single platform technology.

When considering the disparity of metadata descriptors, runtimes, platforms, and "native" tools, the task of ensuring that all the parts of an application are properly provisioned, configured, connected, and are of consistent versions, remains complex - and also resists automation due to said disparity. The diversity and combination of landscapes, from traditional products to cloud-based services, adds even more complexity.

The MTA specification addresses this lifecycle and orchestration complexity of continuous deployment for Cloud and On-premise platforms. We use the following definition:

> ***A multitarget application (MTA) is comprised of multiple parts ("modules")[1], created with different technologies and deployed to different targets, but with a single, common lifecycle.***

The MTA addresses the deployment challenges by isolating the developer from target specific native tools (for example, CF push), via a formal target-independent **application model.** Developers describe the modules of the application, the interdependencies to other modules, MTAs and (micro-) services, required interfaces, and exposed interfaces. An MTA-aware application lifecycle management framework validates, orchestrates, and automates the MTA deployment on premise and on cloud platforms[2]. At SAP, the SAP Web IDE provides MTA-aware development support both in the context of the new XS Advanced solution for SAP HANA and for Fiori-as-a-Service solutions. MTA aware deployers for HCP, XS Advanced and Cloud Foundry provide deployment services for these platforms.

---

[1] *A "module" does not necessarily need to be code to be executed in a runtime container. Consider, for instance, documents to be deployed to a documentation web server, or API metadata to be deployed to an API portal.*

[2] *SAP's "XS Advanced" effort is bringing cloud platform qualities also into an on-premise world. In such a context, this specification is relevant for on-premise scenarios as well.*

They integrate with SAP product assembly and production, are integrated into traditional SAP transport tools, and support delivery via SAP Service Market Place for on-premise deployment.

MTA concepts

An MTA is logically a single application, consisting of multiple related and interdependent parts (henceforth called ***modules***) developed using different technologies or programming paradigms, and designed to run on different target runtime environments, with a single, consistent lifecycle.

The MTA specification defines the ***MTA model***, a platform-independent description of the different application modules, their interdependencies, configuration data they expose, and the resources they require to run. This model is specified, using YAML [3], in design-time ***MTA descriptor*** files that accompany the development and deployment processes.

The MTA model is the formal contract between *developers* (using development tools) and the ***MTA deployer***. The deployer is a tool[3] that consumes a description of the MTA model and translates it into target platform specific "native" commands for provisioning runtime containers, creating and binding resources ("services" in Cloud Foundry terms), and installing, running, and updating the application modules.

These main concepts are illustrated in Figure 1. An application provider (a developer role) uses development tools to create the modules of an MTA and the corresponding MTA descriptor. The application is then typically delivered and distributed in the form of an ***MTA archive.*** As indicated by the direct arrow from development tools to deployer, MTA deployers may also accept the pure contents of this format, namely the directory structure of files with a deployment descriptor. An administrator optionally augments the MTA model in the deployment descriptor with an extension descriptor and uses the ***MTA deployer*** to orchestrate the actual deployment.

[3] *An "MTA deployer" may be more than a single tool and can include configuration tools and target platform specific deployers to deploy an MTA. Development environments contain such functionality as well, since deployment (for example, for testing) is an integral part of the development process.*

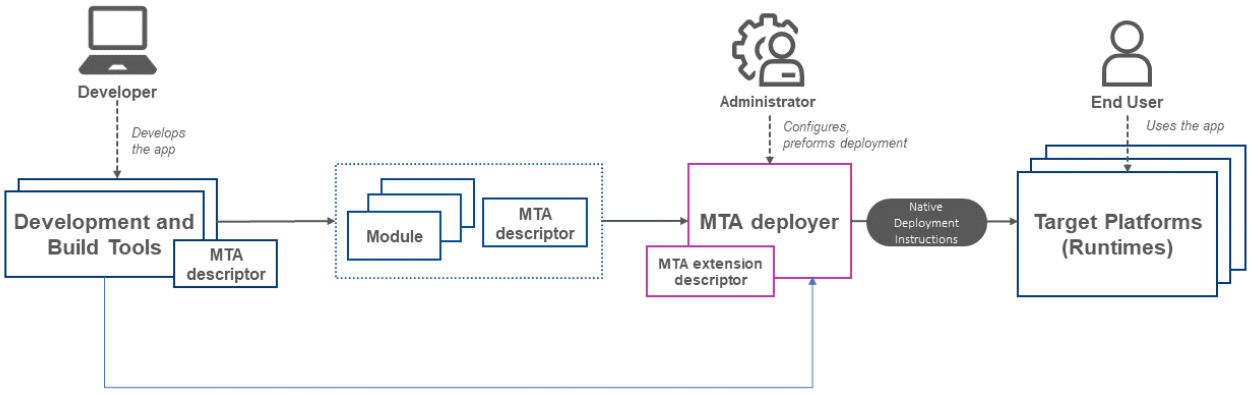

Figure 1.Basic development-to-deployment flow

This extension mechanism is required to keep delivered archives immutable with the additional option to sign them. Multiple extension descriptors can be maintained for the same MTA to support multiple deployment configuration variants. The administrator would also configure the deploy service itself, for example, to set default values used for all deployments.

Development

At design time the developer creates ***source modules***. A source module is typically a directory in the file system and can contain a variety of files. A source module has a *source module type*[4], for example Java or HTML5, which determines development tool choices, such as the tools to "build" it (for example, Maven to build a Java source module). Build tools transform source modules into ***modules***, for example a *.war* archive, which can be deployed to a target runtime[5]. All source modules of an MTA can be subject to the same versioning by a version control system (that is, one unique version can be attached to all source artifacts).

A development cycle may result in a software release, requiring a distribution package. For example, an application may be installed in a production landscape, or uploaded to a marketplace where it can be discovered and deployed by customers. To support this, we define the ***MTA archive*** (which follows the JAR specification) that includes all the modules of the MTA and the deployment descriptor. In the context of XSA, the MTA archive plays the role of the Delivery Unit (DU) of XS classic.

One way to author and build the different source modules of an MTA is to use MTA-aware tools (such as the SAP Web IDE), managing all MTA source modules as a single

---

[4] *Do not confuse types of source modules (directories) with file types based on file extension.*

[5] *This is not necessarily a 1:1 relationship. A build-tool may for instance create both an executable and a documentation module from a single source module.*

MTA project, and if possible, within one Git repository. MTA-aware tools use a ***development descriptor***[6] to manage the design-time perspective of the MTA model, and provide the feature of an "MTA build" to orchestrate the individual source builds, unit, and integration tests, and automatically create the MTA archive with its deployment descriptor.

Alternatively, developers can use any toolset to create and build the modules. In such cases, they are also responsible for creating the deployment descriptor (and optionally the MTA archive) that the MTA deployer will use.

Using MTA-aware development tools improves developer productivity. Dependency provisioning, setup, and configuration can be automatically handled by the tools. An MTA project can conveniently be shared between collaborating teams as a whole. Automatic integration tests between the modules are part of the MTA build, or part of continuous integration and deployment processes.

Deployment

While the MTA model is not target-specific, the MTA deployer is (multi-) target aware. Even when targeting only one platform (for example, Cloud Foundry), the MTA deployer is still needed to interlink the different application parts.

Using MTA deployers instead of the underlying "native" deployment APIs results in a single orchestrated deployment approach for applications that may span multiple runtime tiers on more than one target platform, automating many of the enterprise lifecycle management requirements[7]:

- Auto-provisioning of required services
- Deterministic deployment order and timing. This means that a service must be up before a dependent MTA module can be deployed; one module must be running before another can be deployed.
- Automatic creation and selection of technical artifacts, such as routes, domains, technical names, technical users, and so on, with centrally defined defaults.
- Recoverability: resume deployment from the point of failure of a multi-step operation
- Transactional consistency: a deployment either finishes successfully or fails completely (roll-back), leaving no inconsistent intermediate states.

---

[6] *The development descriptor may be generated from templates, wizards, as well as higher level abstractions.*

[7] *This is essential for companies like SAP which provide mission critical software to large enterprises.*

• Zero downtime update (for example, blue-green strategy), or update with maintenance page

• Support for canary release strategies

• Logging, tracing, and coherent auditability of the entire application lifecycle

• Support for version evolution policies, that is, to protect against downgrades

• Deletion of non-relevant modules and services (caused by application structure changes during update) and release of related infrastructure resources

• Software source policies: deploy only signed content from certain vendors

• Query interface for application monitoring and support tools (application state, dependencies, version history, and so on.

The target-independent MTA model increases portability and durability. Runtime infrastructure changes are centrally handled by the MTA deployer without affecting application developers. As the MTA model is not specific, it lays the ground for providing a uniform development and deployment experience. See section 15.3 for further advantages of the MTA deployment descriptor compared to Cloud Foundry's manifest.yml.

1. MTA Model

1.1 Overview

An MTA model serves the following purposes:

1. **Defines an application composed of multiple (heterogeneous) sub-components** (benefit: tools can establish a unique lifecycle of these sub-components).

2. **Declares resources the application depends upon at runtime and/or deployment time** (benefit: tools can allocate and bind such resources).

3. **Defines configuration variables (and their relation), whose values distinguish different deployments of the application** (benefit: tools can bind sub-components, can automate deployment based on default settings, or request missing mandatory values interactively).

Figure 2 shows the main entities of the MTA model. Also refer to section 15, where these concepts are illustrated by means of a hypothetical sample application.

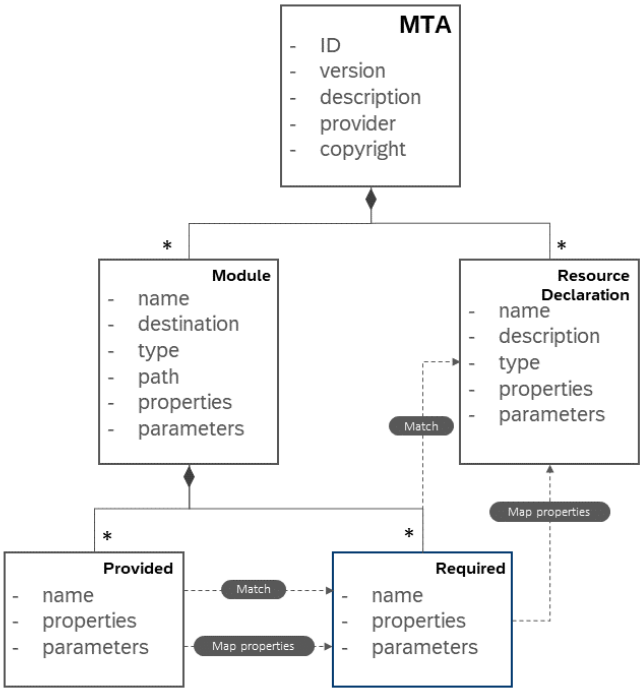

Figure 2: An MTA Model

NOTE: An MTA must have either at least one module or at least one resource.

The MTA model is persisted in descriptor files, using YAML [3]. The following shows a very simple ***deployment descriptor***:

```
_schema-version: "3.1"

ID: com.acme.mta.sample
version: 1.0.1-beta


modules:

 - name: pricing-ui

   type: javascript.nodejs
   requires:

    - name: thedatabase


resources:

 - name: thedatabase

   type: com.sap.xs.hdi-container
```

Table 1: Example deployment descriptor. Keywords which are defined by the corresponding descriptor schema are depicted in bold-blue font.

1.2 Top level descriptor elements

An MTA descriptor has top level descriptor elements, like an identifier and version that uniquely identify it. MTAs contain **modules**. Modules expose (*provide*) elements and depend on (*require*) elements that are exposed by other modules or by declarations of **resources**. Resources are anything required to run the application, but which is not contained inside the MTA. The following top-level descriptor elements are supported:

- The mandatory **_schema-version** is used to indicate to an MTA processing tool (for example, a deployer), which specification version was taken as the base when authoring a descriptor. Usually, it will be enough to indicate a major version here, as then tools can apply the appropriate processing, or they can deny processing if they cannot provide a processing routine which is compatible due to a different major version. Alternatively, the MTA-aware tool might include multiple processing routine versions and select the one matching the version specified in the descriptor. Schema versions have to follow the semantic versioning standard (http://semver.org/) with the exception that trailing numbers (<minor>.<patch> or <patch>) can be omitted. A tool (for example, a deployer) shall then insert the highest numbers it supports. It's a recommended practice to enclose the version string by quotes to make sure that, e.g., "3.0" is really interpreted as a string and not as a number. This number might be converted by a parser into 3, which has a different semantics than the "3.0" string.
- The **ID** value is a mandatory string, not limited in length, uniquely representing the application. It must adhere to the following regular expression:

/^[A-Za-z0-9_\-\.]+$/

The ID should be, by convention, a reverse-URL dot-notation.

- **description:** an optional text, describing the application. This text is not meant to be visible on application user UIs.
- **version**: The mandatory version follows the semantic versioning standard (http://semver.org/). Therefore, it must have at least the structure

  <major version number>.<minor version number>.<patch version number>

  The ID and version of an MTA defines a unique application version. An MTA deployer can compare the version of a newly submitted MTA to one already deployed, and for example reject the new MTA if its version is older than the already deployed software.
- **provider**: an optional name of the organization providing the MTA
- **copyright**: an optional copyright notice of the author
- **parameters**: an optional map of parameters that affect the processing by MTA-aware tools, such as the deployer. See section 1.7 for details.
- **modules**: see below.
- **resources**: see below.

1.3 Modules

Modules are deployable content units that need to be deployed to appropriate targets. Within the MTA development descriptor, the **modules** element lists the sources of the MTA project. Within the MTA deployment descriptor, it lists the deployable modules.

Modules have two mandatory attributes:

- The module **name** (must be unique within the MTA)
- The module **type**

The **name** of a module is a purely MTA internal identifier, not limited in length. It must adhere to the following regular expression:

/^[A-Za-z0-9_\-\.]+$/

A module name must be different from any resource name and any name of a provides-section within the same descriptor. This assures that any "requires:" statement has a unique meaning.

A deployer must carefully decide if and how this name will be reflected on the target platform after deployment. There is always a likelihood that two MTAs use the same internal **name**. This must not lead to the situation that deploying an MTA leads to overwriting runtime artifacts of a previously deployed MTA with a different ID. One mitigation strategy is for deployers to use the MTA ID as a namespace by prefixing generated runtime artifacts such as <**ID**>.<**name**>. Assuming uniqueness of the MTA ID, this prevents naming conflicts. Still, deployers must implement mechanisms to detect naming conflicts and reject conflicting deployment requests.

The **type** of a source module is one out of a reserved list of source module types supported by the MTA-aware development tools[8].

It uniquely identifies the builders and design time tools for processing these source modules. Similarly, the **type** of a deployable module is one out of a reserved list of target runtime types supported by MTA deployers[9].

A source module must have another mandatory **path** attribute. This can be, for example, the file-system path from the root of the MTA project. However, **path** can contain any other type of reference to a (possibly remote) location. Specifying the **path** of a deployable module is optional if a deployment is based on an archive, where the additional MANIFEST.MF-file contains the required path information (see section 4). In case of deploying an archive, the information in MANIFEST.MF has precedence over any **path** specification in the deployment descriptor.

Other optional attributes of modules are:

- **description** (non-translatable text string, not meant to be presented on application UIs)
- **requires**
- **provides**
- **properties**
- **parameters**
- **build-parameters**

1.4 Resources

A resource is an entity required by a module of the MTA at runtime or at deployment time but is not provided inside the MTA. More precisely, an MTA descriptor declares a resource *dependency*, not the resource itself. In this document, if the context is clear, we loosely use the term “resource” and mean the metadata declaring a dependency to a real resource.

---

[8] *The SAP Web IDE currently supports HTML5, Node.js and HDB source module types.*

[9] *The XSA deployer supports modules of type javascript.nodejs, java.tomcat, java.tomee, com.sap.xs.hdi, and among others.*

Resource declarations use the **resources** element with the following important attributes:

- The resource **name** (mandatory, unique within the MTA)
- The resource **type** (optional)

The **name** of a resource is a purely MTA internal identifier, not limited in length. It must adhere to the following regular expression:

```
/^[A-Za-z0-9_\-\.]+$/
```

A resource name must be different from any module name and any name of a provides-section within the same descriptor. This assures that any "requires:" statement has a unique meaning. The **type** of a resource is part of a reserved list of resource types[10] supported by deployers. For some resources, the **type** indicates to the deployer how to discover, allocate, or provision the real resource, for example a managed service like a database, or a user provided service on Cloud Foundry. Other resources are not manageable by the deployer, for example an external weather forecast service, or an OData service exposed by a remote on-premise ERP system.

For such a resource, the deployer may deploy a proxy object (for instance, an HCP destination) that is then used by the requiring module to communicate with the resource.

Other optional resource attributes are:

- **description** (non-translatable free text string, not meant to be presented on application UIs)
- **properties**
- **parameters**

1.4.1 Optional resources

An application can declare resources as optional, if it can compensate for their absence. In this case, if a deployer cannot allocate the required resource, the deployer issues a warning and continues processing. In contrast, if a resource is not optional and fails to be allocated, the deployer throws an error and stops processing.

To declare optionality of resources, the new optional Boolean schema element **optional** is introduced with the possible values true or false. The default value is *false*.

---

10 The XSA deployer supports resource types like com.sap.xs.uaa, com.sap.xs.hdi-container, com.sap.xs.job-scheduler, org.cloudfoundry.user-provided-service, org.cloudfoundry.managed-service and org.cloudfoundry.existing-service

```
...
resources:
  - name: log
    type: application-logs
    optional: true          # true | false. Default value is 'false'.
```

Example 1: Optional Resource

NOTE: The schema element **optional** must not be used in extension descriptors.

1.4.2 (Non-) Active Resources

An application can declare resources to be active or non-active. If a resource is declared to be active, it is allocated and bound according to declared requirements. Usually, resources are set to be non-active by using extension descriptors, so that a resource is not allocated despite being available in a certain deployment target. This can occur, for example, if an application needs to be registered (using a registration resource) only for productive deployments, not for test deployments. The new optional Boolean schema element **active** is introduced with the possible values *true* or *false*. The default value is *true*.

```
...
resources:
 - name: log
   type: application-logs
   active: false          # true | false. Default value is 'true'.
```

Example 2: Non-active resource

NOTE: The usage of **active** is independent from **optional**. This means that both optional and non-optional resources can be set active or non-active.

1.5 Module properties

Module **properties** are (possibly structured) key-value pairs that must be “injected” (made available) to the respective module at runtime. No assumptions are made about

how the MTA deployer injects the values. Many operating systems and languages like C, Java, Go and Python support a concept of *environment variables*, but other methods work as well, for example JNDI in the context of Java. To read the injected values, a developer's code performs an object lookup, and references the module property names.

The used property *names* are determined by an application developer, as these have to be considered within the application code.

The *values* of properties can, of course, be specified at design time, but will more commonly be determined during deployment, either explicitly set by the administrator (via *extension descriptor files)* or inferred by the MTA deployer, for example derived from a target platform configuration (see section "placeholders" below). Once set, the deployer injects the property values into the module's runtime. A deployer should then report an error if it cannot determine a value for a property.

To structure this contract, we specify some constraints and illustrate them with the help of the exemplary assumption of using environment variables (which holds for SAP XSA or Cloud Foundry). A module's **properties** section has a map structure at its first level. Deeper levels can be structured in any way, as long as they can be transformed into a valid JSON object or array[11]. This constraint allows identifying a unique lookup key (the key names of the first level maps) with a well-defined value (the JSON object or array).

- If the value of a first level map key is a single value, then it would be an overhead to wrap it as a JSON object (by using curly brackets). In this case, the plain value string is used (see examples below).
- For module properties, keys of the first level map are used as lookup keys. This means that if environment variables are used, one variable is created per map key and the name of the variable equals the key value. Examples of valid and invalid

  properties sections are shown in Table 2. In the valid, left, example, **company**, **email**,

  and others, are the names of environment variables.

11 This is simply a special case of the requirement that every descriptor shall have an equivalent JSON representation. See https://www.json.org for a definition of "object" and "array".

... **modules:** **- name:** my_module # Valid structure. The first level # of this structure is a map **properties:** **company:** Sirius Cybernetics Corp. **email:** info@ssc.com **countries:** [DE, US, IL] **tax_attributes:** **attr1:** a value **attr2:** another value **employees:** **- code:** 101 **name:** foo **aliases:** [foo1, foo2, foo3] **attributes:** **entry_date:** "12.02.2001" **status:** active **- code:** 102 **name:** bar **aliases:** [bar1, bar2]	... **modules:** **- name:** my_module # Invalid structure. The first # level is a sequence (of maps) **properties:** **- code:** 101 **name:** foo **aliases:** [foo1, foo2, foo3] **attributes:** **validity_date:** "12.02.2001" **status:** active **- code:** 102 **name:** bar **aliases:** [bar1, bar2]

Table 2: Valid (left) and invalid (right) instances of property structures.

- All first level map values and substructures are content for the module runtimes. This content shall be transformed into one JSON object or JSON array per map key. For example, the valid descriptor from Table 2 yields the following environment variables:

Variable	Value
company	Sirius Cybernetics Corp.
email	info@ssc.com
countries	["DE", "US", "IL"]
tax_attributes	{ "attr1": "a value", "attr2": "another value" }

employees	[
	{ "code": 101,
	"name": "foo",
	"aliases": ["foo1", "foo2", "foo3"],
	"attributes": {
	"entry_date": "12.02.2001",
	"status": "active"
	}
	},
	{ "code": 102,
	"name": "bar",
	"aliases": ["bar1", "bar2"]
	}
	]

1.6 Required properties

For most use cases, a *binding* among modules and/or a *binding* of modules to resources needs to be performed. Binding in our case is a process that generates and shares configuration data required to make the set of modules operable.

For example, certain modules consume APIs that are exposed by existing applications, platform services, or by other modules in the same application. The challenge here is that most of these API URLs are determined and configured by the administrator deploying the applications. Another example is a Java application using a database; but the availability of a specific database instance and the associated credentials can only be ascertained at deployment time.

The MTA model offers a formal specification of such required bindings using ***requires-provides*** relations that can be automatically evaluated and enforced by MTA deployers. Binding data is represented as a set of *required* properties.

Modules can *require* - by name – various sets of properties that are declared (that is, *provided)* within the specification of other modules of the same MTA or a resource, or even by other MTAs (see section 5). A module can provide multiple named property sets, a resource provides at most one property set identified by the resource name. A provided name must be different from any resource name and any module name within the same descriptor. This assures that every "requires:" statement has a unique meaning.

The MTA deployer determines all provided property values, then binds the evaluated properties from the provider to the requiring module, and “injects” them to the module runtime, for example by creating environment variables. This is illustrated below:

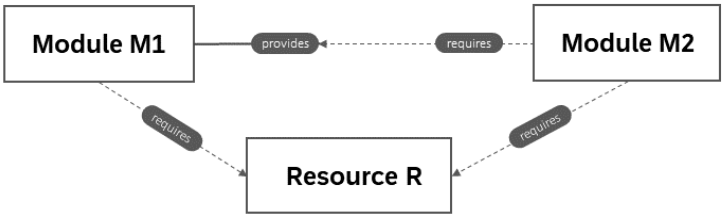

Figure 3: Requires-provides relations

In the example shown in Figure 3, the MTA is composed of two modules and one resource declaration. Module M1 requires configuration from R and provides a property set P. Resource R automatically provides one property set R (which can be empty). Module M2 requires property sets P and R.

1.6.1 Referencing required property values

Previous versions of the MTA specification introduced the **group** element as part of the MTA model. The purpose is to group multiple sets of provided properties into a JSON array (or an equivalent structure if the interpreting tool is not using JSON internally).

Here, an alternative approach is introduced for combining properties from multiple providers, which is not restricted to JSON arrays (or the respective equivalents).

- The provided property reference syntax ~{<provided-property-name>} is extended.
- The extended syntax can be used for constructing values of properties at the module level.
- The extended syntax is ~{<name-of-requires-section>/<provided-property-name>}.

This means that the following two descriptors are equivalent:

```
_schema-version: "3.1"
ID: com.sap.mta.sample
version: 1.0.0

modules:

 - name: pricing-backend

  type: java.tomcat

  properties:

    url: ~{competitor_data/url}
    api_keys: ~{competitor_data/keys}
  requires:

   - name: competitor_data


resources:

 - name: competitor_data
  properties:

    url: "https://examplesite.com/"
    keys:

    app_key: 25892e17-80f6
    secret_key: cd171f7c-560d
```

```
_schema-version: "2.1"
ID: com.sap.mta.sample
version: 1.0.0

modules:

 - name: pricing-backend
  type: java.tomcat
  requires:

   - name: competitor_data
    properties:
      url:  ~{url}
      api_keys: ~{keys}


resources:

 - name: competitor_data
  properties:

    url: "https://examplesite.com/"

    keys:

    app_key: 25892e17-80f6
    secret_key: cd171f7c-560d
```

Example 3: Both above descriptors are equivalent

Using this syntax, the usage of **group** can be avoided, as shown by the two following equivalent descriptors:

```
_schema-version: "3.1"
ID: com.sap.mta.sample
version: 1.0.0

modules:

 - name:  pricing-ui

   type:  javascript.nodejs

   properties:

      API:

      - key: internal1
        conn_string:
~{price_opt/protocol}://~{uri}/odata/

      - key: external

        url: ~{competitor_data/url}
        api_keys: ~{competitor_data/keys}
  requires:

    - name: price_opt

    - name: competitor_data



 - name: pricing-backend
   type: java.tomcat
   provides:

    - name: price_opt
      properties:
        protocol: http
        uri: myhost.mydomain


resources:

 - name: competitor_data

   properties:

      url: "https://marketwatch.com/"
      keys:

        app_key: 25892e17-80f6
        secret_key: cd171f7c-560d
```

```
_schema-version: "2.1"
ID:
com.sap.mta.sampleversion:
1.0.0

modules:

 - name:  pricing-ui

   type:  javascript.nodejs

   requires:

    - name: price_opt

      group: API

      properties:

            key: internal1

       conn_string:

~{protocol}://~{uri}/odata/



    - name: competitor_data
group: API       properties:
        key: external
        url:  ~{url}
        api_keys: ~{keys}



 - name: pricing-backend
type: java.tomcat     provides:

    - name: price_opt        properties:
        protocol: http
        uri: myhost.mydomain


resources:

 - name: competitor_data
properties:

     url: "https://marketwatch.com/"
     keys:

    app_key: 25892e17-80f6
    secret_key: cd171f7c-560d
```

*Example 4: The usage of **group** can be avoided. Both descriptors are eqiuvalent*

The usage of this new syntax allows for more flexibility when composing properties which goes beyond the group semantics as shown in the following snippet:

```
...
modules:
 - name:  pricing-ui
   type:  javascript.nodejs
   properties:
       API:
      name: internal1
      conn_string: ~{price_opt/protocol}://~{uri}/odata/
      externals:
       - key: external1
         url: ~{competitor_data_1/url}
         api_keys: ~{competitor_data_1/keys}
       - key: external2
    url: ~{competitor_data_2/url}
    requires:
     - name: price_opt
     - name: competitor_data_1
     - name: competitor_data_2
...
```

Example 5: The property API can have an arbitrary structure while using references ~{...} at the same time.

As a developer, you are now completely free to choose the structure of the property **API** and are not restricted to JSON arrays.

In addition, consider the following:

- The v2-style of referencing **provided** properties (within requires-sections) is still valid and can still

  be used. It is even possible to use a mixture of v2 and v3-style.

- As already valid for v2, **required** properties must only reference the first level keys of the provided property maps. Any deep structures on the provides-side are “flattened” by transforming them into a JSON string that becomes the **required** property value. It is not possible to use an extended path (for example, ~{<name-of-provides-section>/prop1/sub-prop1}) to address components of structured provided properties.

- If the **list** element is used within a **requires** section (see section 6.1), properties that are linked to provided values must still be listed in this **requires** section.
- The usage of **group,** as described in specification version 2, is deprecated. However, for descriptors tagged with **_schema-version:** "3.0", or higher, but still below 4, already existing MTA tools should keep on supporting **group**.

The same set of provided data can be required by more than one module within the MTA. A requiring module must declare a mapping from *provided* properties to *required* properties. Only the resulting required properties are relevant for the requiring module runtime. This mapping can consist of the following operations:

- selecting a subset of provided properties
- combining provided property values and string literals into required property values
- adding properties that are not provided

To reference a provided property value, a tilde-notation **~{<provided-property-name>}** is used.

The snippet shown in Table 3 illustrates these principles. Properties of the **requires** section must only reference the first level keys of the provided property maps. Any deep structure on the **provides** side will be "flattened" by transforming it into a JSON string, which becomes the required property value.

In the example in Table 3 the property **conn_string** required by module pricing-ui will get the string value *http://myhost.mydomain/odata/*, and the property **api_keys** (required by pricing-backend) will have the string value *{"app_key": "25892e17-80f6", "secret_key": "cd171f7c- 560d"}*.

```
...
modules:
 - name:  pricing-ui
   type:  javascript.nodejs
   requires:
    - name: price_opt
      properties:
        conn_string: "~{protocol}://~{uri}/odata/" # usage of reference notation ~{...}


 - name: pricing-backend
   type: java.tomcat
   provides:
```

```
  - name: price_opt
    properties:
      protocol: http
      uri: myhost.mydomain
  requires:

  - name: competitor_data
    properties:
      url: ~{url}
      api_keys: ~{keys}


resources:

- name: competitor_data

  properties:

    url: "https://marketwatch.com/"
    keys:

      app_key: 25892e17-80f6
      secret_key: cd171f7c-560d
```

Example 6: Example of provided and required properties.

Optionally, a requires section can declare a **group** assignment. Groups are used to combine properties from multiple providers into one lookup object (for example, an environment variable). This reduces the number of lookups done by the code of the requiring module. See the following example:

```
...
modules:
 - name:  pricing-ui
   type:  javascript.nodejs
   requires:
    - name: price_opt
      group: API        # group assignment
      properties:
   key: internal1
        conn_string: "~{protocol}//~{uri}/odata/"


    - name: competitor_data
```

```
      group: API        # group assignment
      properties:
        key: external
        url:  ~{url}
        api_keys: ~{keys}

  - name: pricing-backend
    type: java.tomcat
    provides:
     - name: price_opt
       properties:
         protocol: http
         uri: myhost.mydomain

resources:
 - name: competitor_data
   properties:
     url: "https://marketwatch.com/"
     keys:
       app_key: 25892e17-80f6
       secret_key: cd171f7c-560d
```

Table 3: Provides-requires relation using a group assignment.

Based on the assignment of the group **API**, the deployer has to create a lookup object (for example, an environment variable) with the name **API** with the following content:

```
[
{ "key": "internal1",
    "conn_string": "http://myhost.mydomain/odata/"
  },
  { "key": "external",
    "url": "https://marketwatch.com/",
    "api_keys": {
```

```
                "app_key": "25892e17-80f6",
                "secret_key": "cd171f7c-560d" }
            }
        ]
```

Thus, grouping multiple properties results in having a JSON array as value.

1.7 Parameters

The **parameters** are reserved *variables* that affect the processing by MTA-aware tools, such as the deployer. The set of allowed parameter names is determined by the tool which interprets an MTA descriptor. This is different to property names, which are determined by an application developer. In contrast to properties, parameters are not directly made available to the application runtime. However, this can be accomplished by constructing property values out of parameter values. Each tool (for example. a deployer) publishes a list of reserved parameters and their (default) values for its supported target environments. Thus, each tool owns its own parameter specification, which has a version evolution independent from the schema version of MTA descriptors. In this sense, this document represents a "core" specification on which other tool parameter specifications are based upon. It's up to the MTA-aware tool how to react if it meets unknown parameters. A deployer might issue a warning, but still go on with the deployment process.

Parameters can have read-only values, write-only, or read-write. In the latter case, they have a default value that can be overwritten. Parameter values can be used in descriptors by referencing the parameter name within a placeholder notation, *${<parameter-name>}* (see Table 5). Examples of common read-only parameters are *user*, *default-host*, *default-uri*.

Other parameters are writable, for example, their value can be specified within a descriptor. For example, a module might need to specify a non-default value for a target-specific parameter memory to configure the amount of memory for the module's runtime. Examples of using parameters and placeholders are shown in Table 5.

```
_schema_version: "3.1"

ID: com.acme.mta.sample
version: 1.0.1-beta


modules:
 - name: pricing-ui
   type: javascript.nodejs
   parameters:
     memory: 128M
   requires:
```

```
    - name: price_opt
      properties:
        conn_string: "~{protocol}://~{url_segment}/odata/"

  - name: pricing-backend
    type: java.tomcat
    parameters:
      domain: price.acme.com
    provides:
    - name: price_opt
      properties:
        protocol:
        url_segment: ${host}.${domain}  # usage of placeholders
```

Table 4: Usage of parameters.

In this example, the deployer will replace the reserved ${host} by a value chosen by the deployer for module pricing-backend at time of deployment. The descriptor author decided to choose a value for parameter domain on their own, so that in this case ${domain} resolves to price.acme.com. In turn, these values are considered when determining the value for property **conn_string**. Furthermore, the author sets an explicit value for parameter memory, which is then used when deploying module pricing-ui. This example descriptor is not complete in the sense that the property protocol for module pricing-backend has not been given a value yet. This can be done by using an extension descriptor (see 3.2).

By using the placeholder notation ${...}, a parameter value is retrieved within a certain context. For instance, in the example depicted in Table 5, the placeholder ${domain} represents the value of parameter **domain** as defined in the context of module pricing-backend (which is price.acme.com).

1.7.1 Parameter scopes

Parameters are bound to exactly one scope that defines its validity and visibility. A parameter scope can be either “global”, “module”, “requires” or “resource”. This means that it is possible to introduce two parameters with the same name if they are in different scopes. They can then potentially have different semantics and different values assigned.

It is allowed (but up to the MTA descriptor processing tool) to have parameters in a higher-level scope influencing the default value of a parameter in a lower-level scope (“global” is the highest-level scope, “module” scope is at a higher level than “requires” scope”).

It is not possible to retrieve a parameter value from a different scope by using a placeholder. For example, in Table 5, it is not possible to reference the value of parameter *domain* of the

module *pricing-backend* from within the module *pricing-ui*. Future versions of this specification may introduce a notation to enable this.

Parameter placeholders ${...} used in provides-sections are resolved within the module scope.

1.8 Build Parameters

A development descriptor can contain a **build-parameters** element within its module definition section. This is not allowed to be used in deployment descriptors. Build parameters are interpreted by the MTA build tool (described in [7]), which builds each of the modules in an MTA project, translates the development descriptor (mta.yaml) into the deployment descriptor (mtad.yaml) and packages the results into an MTA-archive. The set of build parameters is defined by the MTA build tool.

```
...
modules:
  - name: java-backend
    type: java
    path: java-sources
    build-parameters:
      maven-opts:
        defines:
          skipTest: true
...
```

*Table 5: Section of a development descriptor showing an exemplary use of **build-parameters**.*

2. MTA Descriptors

The MTA model is persisted in MTA descriptor files, using the YAML format (www.yaml.org) for its readability and support for comments. Due to the widespread tool/library support, the YAML 1.1 specification is taken as reference, although the MTA descriptor schema should be compatible with YAML 1.2 as well. Descriptor files contain the application-describing metadata for the various development and deployment tools. In general, there are three types of descriptor files: (1) development descriptor, (2) deployment descriptor, and (3) extension descriptors. A deployment descriptor is always required. It depends on the concrete development scenario and flexibility requirements which of the other two descriptors are involved in addition.

2.1 Development vs. deployment descriptor

A developer can use any development and build tool chain to create deployable modules together with the MTA deployment descriptor. Optionally, this developer should package all these artifacts into an MTA archive, as specified in section 3 for distribution purposes. This process only requires writing the **deployment descriptor** mtad.yaml. The deployment descriptor is then checked-in to a version control system.

Another option has been implemented by SAP Web IDE. There, the IDE is "MTA-aware"; for example, MTAs can be preferred in a multi-module development project. In this case, the deployment descriptor mtad.yaml is the immutable result of a build process owned by the IDE. Still, a developer needs a way to declare the MTA model, which is now done from a design-time and build perspective within the **development descriptor** called mta.yaml.

The mta.yaml is versioned in a version control system. This design-time perspective can be different from the deployment perspective. The latter is the result of a transformation (that is, the build). Therefore, mta.yaml declares modules of source code, whereas mtad.yaml declares deployable modules. This separation of concerns allows dealing with high level programming models where one source code module could correspond to multiple deployable modules. Further, the usage of certain module types can be related to the presence of related resources. As an example, when using SAP Web IDE for HANA to develop HANA database content artifacts (module type **hdb**), then SAP Web IDE can make sure that an appropriate resource of type **com.sap.xs.hdi-container** is declared in mta.yaml and bound to the module (see the "TinyWorld" tutorial on SCN, or the SAP Web IDE for HANA Documentation for more details). In corner cases, an IDE can even hide the file mta.yaml completely from the developer, making it an internal artifact as well, generated from the developer's project settings. The MTA-awareness of an IDE makes it possible to implement an overarching MTA build step. This MTA build orchestrates the individual module builds and can include integration tests covering the interaction among the modules.

2.2 Extension descriptors

Descriptor files can be extended via the concept of extension descriptors, whose contents are merged with the main descriptor to form a complete MTA model. This capability is usually used by administrators to complement deployment configuration. There are two instances where extension descriptors can be applied:

1. Design time: While a developer is the primary author of the development descriptor, there might be other roles who add properties. For better work organization, it is useful to author extensions within separate files. The MTA build process merges the development descriptor with any available extension before doing the actual build.
2. Deployment: In theory the developer-created deployment descriptor can contain all information required to deploy the MTA, but this is unlikely in practice. Usually,

deployment information is not stored in a version control system but is added before the actual deployment. An extension descriptor (provided by an administrator, perhaps using a wizard) is a way to declare such last-minute information. In this way, also alternative deployments can be configured by using multiple extension descriptors for the same application.

An extension descriptor must declare which other descriptor it is extending. This allows building an extension chain having a development or a deployment descriptor as a root. Other descriptors are referenced by their IDs, not by filename. In general, merging multiple extensions is coordinated in the following way (cf. Figure 4):

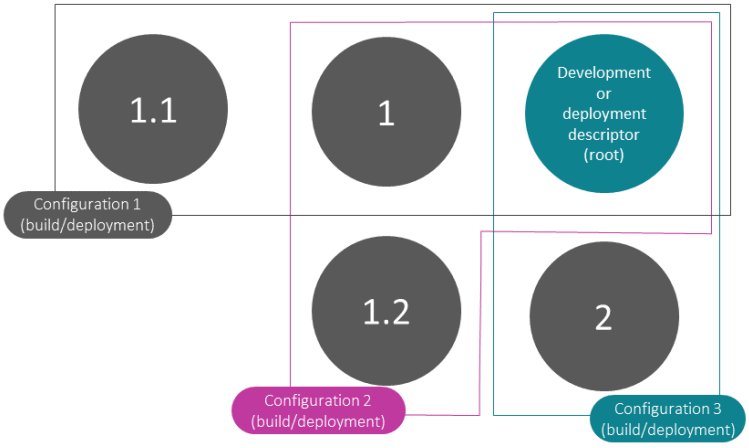

Figure 4: Multiple extension descriptors are grouped into extension configurations.

- An "extension configuration" is a linear chain of extension descriptors.
- Extension configurations can be used for building an MTA (build configuration), or for deploying/running an MTA (deployment/run configuration). Being part of a build or deployment configuration is not encoded into a single extension descriptor. Instead, this is decided by the development/deployment tools accessing this descriptor.
- It shall not be possible to build/deploy an MTA by using extension descriptors from multiple extension configurations.

If there is a need to recognize extension descriptors through the file extension, then the extension **mtaext** shall be used.

2.2.1 Extension rules (what does it mean to “merge” an extension)

You can only *add* information via an extension, not change (= overwrite), or delete. The following additions are allowed:

- o add property and/or parameter elements (= name of a property or parameter)
- o add a value to an existing property or parameter element without value
- o the following can be added via an extension to a deployment descriptor's root level (not to a development descriptor):

targets: <sequence of targets>

This instructs the deployer to use a specific set of deploy targets. For example, a deployer might support a DEV target (for example on Cloud Foundry, org=myorg, space=DEV) and a PROD target. Technically, a YAML sequence (corresponds to a JSON array) is specified because the MTA modules might be distributed across multiple targets. Any other add-operation is not allowed.

In particular, there shall be no way to add new **modules** or **resources**, or to add new **requires** or **provides** nodes.

- An extension descriptor should just include the pieces of information which must be added. If identical information is added again, it is ignored. Properties or parameters are added to a module or resource, by specifying them in context.

```
_schema_version: "3.1"

ID: com.acme.mta.sample.config1
extends: com.acme.mta.sample


modules:

 - name: pricing-ui
   parameters:
     instances: 2


- name: pricing-backend
  provides:

   - name: price_opt
     properties:
       protocol: https
```

Table 6: An extension descriptor extending the deployment descriptor shown in Table 5.

- **Parameters/properties without values**: If a parameter/property within a deployment descriptor is listed without any value, it must receive a value that is assigned by an extension descriptor. Otherwise, the deployment will fail. According to the YAML specification, the absence of a value is indicated either by specifying no value (or a set of whitespace characters), or by assigning the literal null.

```
A null: null
Also a null:    # empty
Not a null: "null"
Still not a null: ""
```

Table 7: null-values and non-null values according to the YAML specification.

NOTE: because the empty string "" is not a null value, it is valid to use it in an extension descriptor to assign a value to a parameter/property.

2.3 Validating descriptor files

As the MTA model is represented by a descriptor file (or a set of files), the problem of model validation must be solved by validating the syntactical correctness of the content of the descriptor file. There are three levels of actors (or authorities) that define what the validation rules are as shown below in Figure 5.

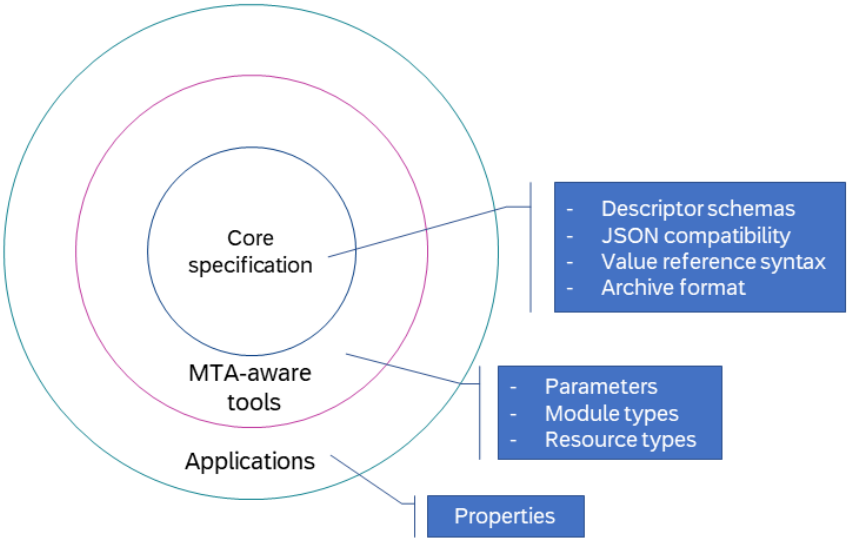

Figure 5: Three levels of authorities are responsible for validating an MTA model instance.

The MTA core specification includes:

1. Descriptor schemas documenting exactly which key elements can be used within descriptor YAML files.

   For example, there is a **modules** key that must have **name** and **type** keys as sub-elements. In addition to the schema definition based on the Kwalify schema definition language (see below), there are some additional restrictions:

   a. All key names used in descriptors are case-sensitive, for example “**MODULES“or** “**Parameters“are** not valid. Only “**modules**” and “**parameters**” are accepted. This reflects a typical restriction of schema validators.

   b. The string values of **ID** and all **name** keys defined in the schema must obey the following regular expression:

      /^[A-Za-z0-9_\-\.]+$/

      This means that the complete string value must be composed of upper-case letters A-Z, and/or lower-case letters a-z, and/or digits 0-9, and/or underscore “_”, and/or dash “-“, and/or a period “.”. No length limit is specified, though for example, deployers might have to deal with length limits when generating runtime objects.

   c. The descriptor keys **type**, **description**, **provider**, and **copyright** are Unicode strings of arbitrary length.

   d. If the **path** descriptor key represents a file system path, it must be interpreted as a relative path and must be path-traversal safe. This means that **path** must not begin with a slash "/", backslash "\" or two periods "..".

   e. As **properties** and **groups** are often converted into environment variables, certain restrictions for allowed characters must be considered. Such restrictions are described in [2] as part of the POSIX.1-2008 standard. This means that names are composed of letters, digits and the underscore (“_”) sign, and must not begin with a digit. Some shells and utilities even restrict to capital letters only.

   f. Tools validating and parsing an MTA descriptor ensure that map keys have unique names. This is typically not validated by common YAML parser libraries. Uniqueness of key names prevents ambiguous descriptors, like in the following example:

      ID: com.acme.mta.sample

      version: 1.0.1-beta

      version: 2.0.1

      Note how the version of this descriptor is not well defined, for the sake of the example.

   g. All names of **requires**-sections must be provided within one MTA descriptor.

h. Module names, resource names, and names of *provides* sections must all be different from each other within the same descriptor. This assures that any "requires:" statement has a unique meaning. The core specification does not specify any restrictions on parameter names and values.

2. The requirement to be able to transform all MTA YAML descriptors into an equivalent, valid JSON format.

3. The syntax and rules how to reference property values (using ~{...}) and parameter values (using ${...}).

4. An archive format that can be used to package and distribute applications.

The next level of specification authorities is represented by MTA-aware tools, for example all tools that interpret MTA descriptors (design-time tools, build tools, and deploy tools). Each of these tools can specify its own set of module types, resource types, and parameters. It is useful to align these specifications to a reasonable extent, as this increases the portability of applications. Such alignments would be favorable for module and resource types, at the minimum, because parameters can be specified in extension descriptors - which come in addition to (and on top of) the deployment descriptor delivered with the application. This way, deployer or target platform-specific settings can be detached from the actual application model.

The third level of specification is provided by each application itself, by introducing the set of (potentially structured) properties that are used for application configuration. This set is highly application-specific, as property names are referenced in application coding as well, for example when accessing environment variables derived from property names.

With respect to validating against this core specification, we use Kwalify (http://www.kuwata-lab.com/kwalify/) for schema definition and validation. You can use any equivalent framework as well.

3. MTA Archives

MTA archives are designed to be compatible with the JAR file specification [1]. This allows reusing common tools for creating, manipulating, signing, and handling such archives.

The deployment descriptor always contains the description of the entire application (all modules and resource declarations), but there may be cases where an archive doesn't contain *all* MTA modules defined in the descriptor. An example, there could be a module with an unsupported module-type; another means could be used to deploy such a module, while the remaining elements of the MTA can be bundled into the MTA archive and handled in a straight-forward fashion.

To deal with such "partial" archives, the deployment descriptor does not necessarily contain information concerning the location of modules within the archive (cf. optional path attribute described in section 2.3). This location aspect amongst others, such as hash values and signatures, is added by the archive manifest **MANIFEST.MF**.

The deployment descriptor is located within the **META-INF** folder of the JAR. The file **MANIFEST.MF** contains at least a **name** section for each MTA module contained in the archive. Following the JAR specification, the value of a name must be a relative path to a file or directory, or an absolute URL referencing data outside the archive. It is explicitly required to add a row

MTA-module: <module name>

to each **name** section that corresponds to an MTA module, to bind archive file locations to module names as used in the deployment descriptor. The **name** sections with the **MTA-module** attribute indicates the path to the file or directory that represents a module within the archive.

In case of deploying an archive, the information in **MANIFEST.MF** has precedence over any **path** specification in the deployment descriptor. If there is an MTA-module entry in the manifest that points to a non-existing file or directory in the archive, an error is thrown by the deployer.

If there is an MTA-module entry in the manifest that refers to a module name not listed in the deployment descriptor (mtad.yaml), a warning should be thrown by the deployer.

In addition to declaring MTA modules, **MANIFEST.MF** can encode the binding of additional configuration files to resources, or to a **requires**-relation between a module and a resource. Such binding is useful when, for example, a deployer can use this additional configuration file when creating the required resource. Such configuration files must be included within the archive. They must not be part of module content. For a deployer, modules are opaque objects (for example, a war-file). A file within a module must be considered as inaccessible

by the deployer. To establish such bindings, the keys **MTA-Resource** and **MTA-Requires** are introduced as shown in Table 8.

Manifest-Version: 1.0

...

Name: src/backend.zip
MTA-Module: backend
Content-Type: application/zip

Name: cfg/backend-db-params.json

MTA-Requires: backend/db
Content-Type: application/json

Name: cfg/security.json
MTA-Resource: uaa

Content-Type: application/json
...

Table 8: Example for using MTA-Resource and MTA-Requires for encoding a binding of configuration files to resources and requires-relations.

Generally, it is required that the rows are built as:

MTA-Resource: *<name of resource>*

and

MTA-Requires: *<module name>/<name of requires section>*

The referenced names must be defined in the deployment descriptor mtad.yaml contained in the archive. If not, the deployer must throw an error.

An archive can contain any other artifacts that are not related to any deployment descriptor entries. It is not defined by this specification what should happen with such artifacts.

The extension **mtar** is used as the file extension of an MTA archive.

4. Requires-Section for Resources

In version 2 of this specification, requires-provides dependencies enable the possibility of exchanging configuration data between modules, as well as between resources and modules. Note that requires-provides does not necessarily enforce deployment order or resource allocation order.

Here, we introduce requires-sections for resources, to cover use cases where configuration data needs to be transferred from a module to a resource, or between two resources. An example is depicted in Example 7, where a resource is requiring configuration data from a module to set up a whitelist of redirect-URLs.

```
_schema-version: "3.1"
...
modules:
 - name: approuter
   requires:
    - name: uaa
   provides:
    - name: backend
      properties:
        url: ${default-url}/foo
resources:
 - name: uaa
   type: uaa-type
   requires:
    - name: backend
   parameters:
     config:
     oauth2-configuration:
       redirect-uris: [~{backend/url}]
```

Example 7: A resource which requires configuration data from a module.

5. Provisioning MTA-external Configuration Data

Multitarget application modules can provide configuration data to other modules within the same MTA. This concept is extended here by being able to tag provides-sections as **public**. The semantics of **public** is to indicate to a deployer that the provided properties are potentially required by other components "outside" the MTA. No assumption is made about what these components are and how a deployer is providing the public properties to these components.

A new optional schema element **public** is introduced for use in the context of modules' provides- sections. It can take values true or false. If **public** is not specified, then "**public**: false" is used by default. MTA-internally, public provides-sections can be required by other modules within the same MTA the same way as non-public sections.

```
_schema-version: "3.1"
ID: com.sap.ui5-v1.28
version: 1.28.17
modules:
 - name: ui5
   type: javascript.nodejs
   provides:
    - name: ui5-lib
      public: true   # public value = true | false. Default = false
      properties:
        type: com.sap.ui5-lib
        url: ${default-url}
        lib-version: "1.28.17"
...
```

Example 8: Declaring a public provides-section.

A developer of a module must be aware that declaring a provides-section as **public** is equivalent to publishing a public contract. In Example 9, any consumer of the provided information must assume that this information is composed of three key-value pairs with the keys **type**, **url** and **lib-version**. Any change of this contract can cause various consumers to encounter issues.

NOTE: The new schema element **public** must not be used in extension descriptors.

6. Consumption of MTA-external Configuration Data

This scenario assumes that an appropriate configuration store or registry is available and accessible by the deployer[12]. This specification neither assumes the structure of the data persisted in such a registry, nor how such a registry has been implemented. Further, for the consumption scenario described here, we make no assumption about how the registry entries are created (for example, by the MTA deployer). Furthermore, we assume that the MTA deployer implements a resource type which can be read from this configuration registry.

We specify a way to model the consumption of such externally stored configuration during deployment. As this modeling is partly based on **parameters**, it reflects the above assumptions we made about the deployer.

The following examples show an application consisting of a "framework" module that has a dependency to a resource, which can be read from the configuration registry during deployment. Registry entries have been populated by “plugin”-type applications to share a name and a URL endpoint for embedding "plugins" into the "framework".

The case for consuming exactly one set of external configuration data is illustrated by the following descriptor:

```
_schema-version: “3.1”
ID: com.acme.framework
version: 1.0.0

modules:

 - name: framework

   type: javascript.nodejs

   requires:

    - name: plugin

      properties:

         plugin_name: ~{name}     # reference to the key ’name’ in the config store
         plugin_url: ~{url}/sources

resources:

 - name: plugin

   type: configuration

   parameters:
```

[12] *For XSA and Cloud Foundry, such a registry has been introduced.*

```
        filter:
          type: com.acme.plugin  # query config store with filter 'type'
```

Example 9: Consuming one set of external configuration data.

The module framework declares a dependency to the resource plugin. The resource type name configuration used here is just for the sake of the example, as a specific deployer might introduce a different name. It is assumed that the deployer is instructed by this resource type to read from the configuration registry using a query defined by the **parameters** section of the resource. The parameters used here just serve as an example, as these reflect the structure of the configuration registry and the capabilities of the resource to query from it. This example assumes that "plugins" have been registered by creating entries in the registry with the fields type, name, and url. The type field is used as a selection criterion, name and url represent the result set.

It is specified here that all values of the fields in the configuration store can be accessed by a tilde reference in the corresponding requires sections.

In Example 9, ~{name} and ~{url} and are assigned to the properties plugin_name and plugin_url in the section that requires the configuration resource.

NOTE: Consider as a recommendation the way parameters are used in Example 8 to model a query to the configuration store.

The way the requirement plugin is declared implies that exactly one configuration data set must be returned by the resource plugin. If this is not the case, the deployer throws an error.

6.1 New schema element "list"

To model the consumption of potentially multiple configuration data sets, the new schema element **list** is introduced. It can be used at requires-sections entries. The module declaration of Example 9 can be extended as follows:

```
...
modules:
 - name: framework
   type: javascript.nodejs
   requires:
    - name: plugins
```

```
      list: plugin_configs  # 0 to n data sets shall be retrieved from the config store
      properties:

        plugin_name: ~{name}       # reference to the key 'name' in the config store
        plugin_url: ~{url}/sources

resources:

 - name: plugins

   type: configuration

   parameters:

     filter:

       type: com.acme.plugin  # query config store with filter 'type'
```

Example 10: Usage of list to process zero, one, or multiple sets of properties.

In this case, all found configuration data sets are listed within a JSON array within the environment variable plugin_configs. It is a valid outcome if no configuration data is found, resulting in an empty JSON array "[]" within the variable plugin_configs. If only some of the required properties are not found, then the JSON array will not contain these properties at all. The requiring application must deal with the fact that potentially no configuration data is found or is partially missing.

The behavior when no configuration data can be retrieved must be specified further to include the case of structured properties. This is done by the snippet shown in Example 11.

```
...

requires:

 - name: services

   list: service_configs
   properties:

     url: ~{url}

     keys:

       app_key: ~{key1}
       secret_key: ~{key2}

...
```

Example 11: Structured required properties.

If no matching properties are found and if we assume that environment variables are created from properties, then Example 10 generates the variable service_configs in the environment of the requiring module containing an empty JSON array "[]".

If a property can only be partially resolved, keys that cannot be resolved are not included into the resulting JSON array. For instance, assume the key key2 is not available in the configuration registry, then the result would be:

```
[ {"url": "https://mydomain.com/myservice/v1",
 "keys":
  {
    "app_key": "BDG75KBETSD9VZ"
  } } ]
```

NOTE: if **list** is used, then the required properties must be listed as part of the requires-section, and cannot be represented as module properties as described in section 2.
If one module declares multiple requires sections with the same **list** key, then all lists with the same key are appended (the JSON arrays are merged into one). The new schema element **list** must not be used in extension descriptors.

6.2 Providing and consuming external configuration data by the same MTA

It is possible for a module to declare public configuration data, which is consumed within the same MTA via a resource that reads the configuration registry. This enables you to combine configuration provided by the module with data provided from outside the MTA. For example, Example 8 and Example 9 can be extended to include an additional module default_plugin:

```
_schema-version: "3.1"
ID: com.acme.framework
version: 1.0.0

modules:

 - name: default_plugin

   type: javascript.nodejs
   provides:

    - name: default

      public: true

      properties:
```

```
      name: default

      url: ${default-url}
      type: com.acme.plugin

  - name: framework

    type: javascript.nodejs
    requires:

     - name: plugins

       list: plugin_configs
       properties:
         plugin_name: ~{name}
         plugin_url: ~{url}/sources
resources:

  - name: plugins

    type: configuration

    parameters:

      filter:

        type: com.acme.plugin
```

Example 12: Configuration that is published by an MTA module must be consumable within the same MTA.

The module default_plugin provides its properties as public. Thus, the resource *plugins* is expected to find this data in the configuration registry and can consider it when assembling the data list for the module framework.

6.3 Further modeling suggestions for consuming public configuration data

It is required that consumers are enabled to react to changes of configuration data they depend on. Either the consumer is actively notified about such change events (based on a prior subscription), or the consumer regularly polls (for example, using a message queue) for such changes.

There are no standard descriptor schema elements for modeling such processes. However, represented below is our recommendation on how to deal with the subscription use case based on deployer-specific parameters. As this modeling is based on **parameters** only, it reflects assumptions we made on capabilities of the deployer.

There is a combinatorial set of variants according to the answers to two questions: (1) who is creating the subscription; (2) who is handling change events. Both questions could be, in principle, answered by "deployer" or "application".

We restrict here to two variants only:

1. The deployer is querying the configuration registry, doing the subscription, and handling change events. The application is not involved in this. Any configuration changes would lead to a restart of the consuming applications after the deployer has set new values of environment variables. There should be the option to automatically restart or involve a human operator to trigger a restart.

   Example 12 shows a **modeling recommendation** to support this variant. Because parameters are used for modeling, this is a suggestion for deployers which parameters they might introduce.

```
_schema-version: "3.1"
ID: com.acme.framework
version: 1.0.0

modules:

 - name: framework

   type: javascript.nodejs

   requires:

    - name: plugins

      list: plugin_configs

      properties:

         plugin_name: ~{name}

         plugin_url: ~{url}/sources

      parameters:

         managed: true  # can be true | false. Default is false

resources:

- name: plugins

  type: configuration
  parameters:

     filter:

        type: com.acme.plugin
```

*Example 13: The parameter **managed** is determining whether the deployer is creating a subscription and is handling change events. Querying the configuration registry is modeled as a dependency to the resource of type **configuration**.*

2. The application is responsible for querying, subscribing and event handling. The deployer is not involved in this. We assume that there is a resource (for example, a managed service on Cloud Foundry) that wraps the configuration registry. The module is bound to this resource. Consequently, there would be configuration data

that is not stored within the environment of an application. Instead, this configuration data is read by the app from a configuration service during startup or when change events occur (if the app has subscribed). Example 14 shows a modeling recommendation to support this variant.

```
schema-version: "3.1"
ID: com.acme.framework
version: 1.0.0

modules:

 - name: framework

   type: javascript.nodejs
   requires:

    - name: plugins_configs


resources:

 - name: plugins_config

   type: managed_configuration_service
```

Example 14: The application takes full responsibility for communicating with a configuration service. To support this, an adequate resource type must be supported by the target platform.

7. Parameter Files

In descriptors, parameters are used to augment the behavior of the deployer when deploying modules or when allocating and binding to resources. Especially for resource configuration, such parameter sets can be structured and be very extensive. Furthermore, it might be more convenient for the developer to author certain parameter sets in terms of separate files because these are used by other tools as well.

We specify a way to include such files into the development and build processes for MTAs. Thus, **the following schema extension is valid only for MTA development descriptors that are the input of a build process**. This build process generates MTA deployment descriptors.

For MTA **development** descriptors, we introduce a new schema element **includes**. It can be used at any level where **parameters** can be placed. The values of **includes** point to a set of parameter files located on a file system.

It is required that, semantically, such a parameter file represents an externalized map of parameters within the descriptor. Thus, there is an alternative and equivalent representation in which all information contained in the file is represented as parameters in the descriptor. The build tools require that the parameter-based representations are generated during the creation of the deployment descriptor from the development descriptor.

As there should be support for multiple parameter files with different semantics, a set of **name** and **path** pairs can be included.

Development and build tools must at least support parameter files in YAML and JSON format. The following examples show the specific places where **includes** can be used in a development descriptor:

```
...
modules:
  - name: backend
    type: javascript.nodejs
    path: src/backend
    includes:
     - name: params-from-file        # this name is used when generating param entries
       path: cfg/backend-params.json  # a path to a file in the file system
    requires:
     - name: uaa
     - name: db
       includes:
        - name: config          # this name is used when generating param entries
```

```
        path: cfg/backend-db-params.json  # a path to a file in the file system
    resources:
     - name: db
       type: org.any-db
     - name: uaa
       type: org.cloud-foundry.uaa
       includes:
        - name: config # this name is used when generating param entries
          path: cfg/security.json  # a path to a file in the file system
    ...
```

Example 15: Development descriptor that includes parameter files.

It is required that a build tool resolves parameter files and convert them to an explicit list of parameters in the deployment descriptor. The resource section of Example 15 would be converted into:

```
    ...

    resources:
     - name: db
       type: org.any-db
     - name: uaa
       type: org.cloud-foundry.uaa
       parameters:
         config:
           <map of parameters which were contained in cfg/security.json>
    ...
```

Example 16: A deployment descriptor generated from the development descriptor shown in Example 15.

The new schema element **includes** is relevant for extension development descriptors because it represents a shorthand notation for a set of parameters. If the parameters inside the file include placeholders that are supported by the MTA deployer, then these would take effect as in regular usage.

8. Content Files

If no MTA-aware build tools are used, the deployment descriptor should be created manually (or by some other means, at least not based on an MTA development descriptor). Nevertheless, it is sometimes required to reference files from deployment descriptors to use during the deployment process. These files may be used as "parameter files", as mentioned above, for development descriptors or could contain any content that is to be considered during deployment. Therefore, we refer to them as "content files".

It is entirely up to the deployer how to process such files. A deployer can define resource type or module type specific parameters within the deployment descriptor to reference such files during the deploy process. An example is shown in the following deployment descriptor snippet:

```
...
resources:
  - name: uaa
    type: org.cloud-foundry.uaa
    parameters:
      config-path: cfg/security.json
...
```

Example 17: A possible reference to a content file in a deployment descriptor, when deploying without an archive.

While this specification relies on the deployer for referencing content files in deployment descriptors, MTA specification v2 provided information on how to reference content files in the manifest of MTA archives to bind these files to resources and/or requires sections of modules. Currently there is no known use case that requires binding of content files to modules, as the module artifacts themselves represent such content.

Content files referenced by deployment descriptors or within an MTA archive must not be part of module content. For a deployer, modules are non-transparent objects (for example, a war-file). A file within a module must be considered inaccessible by the deployer.

9. Metadata for Properties and Parameters

9.1 Metadata maps

Due to the requirement that properties and parameters should be able to carry more attributes than their pure value, we introduce the optional maps **properties-metadata** and **parameters-metadata** for development and deployment descriptors. They can be used on the same level as **properties** and **parameters** (see Example 18). Usually, such metadata is interpreted by UI-based tools for input validation when specifying the final deployment configurations.

```
...
modules:
  - name: frontend
    type: javascript.nodejs
    parameters:
      memory: 128M
      domain: ${default-domain}
    parameters-metadata:
      memory:
        optional: true
        overwritable: true
      domain:
        overwritable: true
    properties:
      backend_types:
        order_management: sap-erp
        data_warehouse: sap-bw
    properties-metadata:
      backend_types:
        overwritable: true
        optional: false
```

Example 18: Specifying metadata for properties and parameters.

NOTE: Do not use **properties-metadata** and **parameters-metadata** within extension descriptors. Names of parameters and properties for which metadata is declared must also be declared within the same descriptor.

If a parameter or property itself has a map structure, then metadata can only be defined for the root node of this map (see property **backend_types** in Example 17). Metadata is not inherited to sub-nodes.

NOTE: **properties-metadata** must not be used in the context of **list** and **group**.

The following metadata keys shall be supported by MTA processing tools:

Metadata Key	Allowed values (default is underlined)	Scope	Description
overwritable	true, false	parameters, properties	If set to true, the value can be overwritten by an extension descriptor.
optional	true, false	parameters, properties	If set to false, a value must be present in the final deployment configuration. A parameter or property value consisting of a set of "whitespaces" or "blanks" (ASCII code Dec 32) shall be interpreted as "no value given". This is equivalent to using the YAML literal null. In this way, values can be "deleted" by extension descriptors if overwrite and optional have been set to true.
datatype	str, int, float, bool	properties	Due to parameters being owned by the deploy service, it doesn't make sense to specify a datatype in the descriptor. Developers might specify a wrong datatype within a descriptor. Furthermore, a UI tool which wants to leverage type information will not find all possible parameters within a descriptor. Therefore, the deploy service should offer an interface, which a UI tool can query for possible parameter names together with the expected datatypes and default values. The datatypes listed here are specified at http://yaml.org/type/
sensitive	true,false	parameters, properties	To specify whether information is sensitive, so a UI tool can, for example, mask its value.

Table 9: Standard set of metadata.

In addition to the standard metadata set listed in Table 9, each MTA processing tool can support its own additional metadata fields. If a tool encounters an unknown metadata field, it should issue a warning (and continue processing), not an error (and interrupt processing).

9.1.1 Overwriting structured maps

The metadata key **overwritable** carries special semantics when it comes to properties or parameters whose value is a map:

- If a parameter/property value is a scalar, then **overwritable** works as usual on scalar level.
- If a parameter/property value is a map, then **overwritable** has the semantics of "being extendable".
- A parameter/property with no value can be "overwritten" by a scalar value or a structured value.

The following examples illustrate the behavior:

Deployment descriptor	Extension descriptor	Merged descriptor
... **modules:** **- name:** java_app **type:** com.sap.java **properties:** **jvm_args:** **arg1:** value1 **arg2:** value2 **arg4:** **arg41**: value41 **properties-metadata:** **jvm_args:** **overwritable:** true ...	... **modules:** **- name:** java_app **properties:** **jvm_args:** **arg2:** **arg3:** value3 **arg4**: **arg42**: value 42 ...	... **modules:** **- name:** java_app **type:** com.sap.java **properties:** **jvm_args:** **arg1:** value1 **arg2:** **arg3**: value3 **arg4**: **arg41**: value41 **arg42**: value42 **properties-metadata:** **jvm_args:** **overwritable:** true ...

Example 19: Overwriting a property that has a map value provides the option to extend the map and/or change its values.

Deployment descriptor	Extension descriptor	Merged content
modules: - name: java_app type: com.sap.java properties: jvm_args: properties-metadata: jvm_args: overwritable: true ...	... modules: - name: java_app properties: jvm_args: arg2: arg3: value3 ...	... modules: - name: java_app type: com.sap.java properties: jvm_args: arg2: arg3: value3 properties-metadata: jvm_args: overwritable: true ...

Example 20: Overwriting a property with an initially empty value allows you to add any substructure (a map in this example, but a sequence is possible as well).

Deployment descriptor	Extension descriptor	Merged descriptor
modules: - name: java_app type: com.sap.java properties: jvm_args: properties-metadata: jvm_args: overwritable: true ...	... modules: - name: java_app properties: jvm_args: '{"arg1": "value1"}' ...	... modules: - name: java_app type: com.sap.java properties: jvm_args: '{"arg1": "value1"}' properties-metadata: jvm_args: overwritable: true ...

Example 21: Overwriting a property with an initially empty value can add a scalar.

Deployment descriptor	Extension descriptor	Merged descriptor
... modules: - name: java_app	... modules: - name: java_app properties:	

type: com.sap.java **properties:** **threshold:** 500 **properties-metadata:** **threshold:** **overwritable:** true ...	**threshold:** **t1:** 1234 **t2:** 900 ...	**Error Condition!**

Example 22: It shall not be possible to overwrite a given scalar with a structured value (and vice versa).

9.1.2 Overwriting with subordinate sequences

If a property or parameter has a sequence as a subordinate structure (resulting in a JSON array as value), the overwrite behavior is the same as for scalars. This means that overwriting always results in a complete replacement of the property or parameter value.

9.2 YAML tag !sensitive

YAML allows the use of "local tags" representing type information, specific for applications interpreting the YAML file (in our case, MTA processing tools).

The local yaml tag '!sensitive' is introduced for use with property and parameter values.

The intent is to tag values as sensitive information. It is up to the MTA processing tools to decide what this means in practice. For example, the deployer can mask a value when writing it to log files. As a sensitive metadata key has been introduced above, this tag is mostly for convenience when providing sensitive information which should not be logged.

```
...

modules:
 - name: frontend
   type: javascript.nodejs
   requires:
    - name: api
      properties:
         url: ~{url}
```

```
        pwd: ~{password}
    - name: backend
      type: java.tomcat
      provides:
       - name: api
         properties:
           url: !sensitive ${url}
           password: !sensitive
...
```

*Example 23: The **!sensitive** tag can precede property and parameter values. It can be used even without specifying a final value in the descriptor (like for the property **password**).*

An existing !sensitive tag must not be changed by merging with an extension descriptor. This prevents revealing sensitive information. However, it shall be possible to add the !sensitive tag using an extension descriptor.

10. Deploying Multiple Runtime Modules by Reusing Deployment Artifacts

In some use cases, multiple modules can reuse the same code. This results in multiple deployed runtime modules based on the same source module that are then usually started with different configuration parameters. A development project might want to have one source folder referenced by multiple module entries in the MTA development descriptor (see Example 24).

```
...

modules:
 - name: fileloader-master
   type: nodejs
   path: js
   properties:
     FL_ROLE: master

 - name: fileloader-worker
   type: nodejs
   path: js
   parameters:
     instances: 3
   properties:
     FL_ROLE: worker
...
```

*Example 24: Two modules referencing the same path **js**.*

If deployment is based on an MTA archive, there shall be no need to duplicate the code to have two different deployable modules. To achieve this, the specification for the MTA-module entry in MANIFEST.MF is extended.

It is possible to author a comma-separated list of module names, which are then used to associate one set of deployment artifacts with all listed modules. The jar manifest related to Example 24 would look as follows:

Manifest-Version: 1.0

...

Name: js/

MTA-Module: fileloader-master, fileloader-worker

11. Module and Resource Types

The specification defines the **type** attribute for modules and resources as a way for developers to identify how their entities should be handled during deployment. The supported module and resource types are dictated solely by the deployer and can vary from one to another.

We recognize that this leaves little control to the user and that a developer may want to create his own custom types by extending the default ones supported by the deployer. This will allow him to create aliases for types and further configure the entities using them.

For this reason, we are introducing two new top-level descriptor elements – **module-types** and **resource-types**. Both can contain a list of objects with the following structure:

- **name**: an MTA internal identifier that must adhere to the following regular expression:

  /^[A-Za-z0-9_\-\.] +$/

  Can be specified in the **type** attribute of modules/resources in the same MTA.

- **extends**: the name of the extended type, which can either be one of the default types supported by the deployer or a custom type defined in the descriptor. In the end, each custom type must have a valid chain of parents that ends with a type supported by the deployer. Multiple inheritance instances are not allowed. This attribute must not be specified in extension descriptors.

- **properties**: an optional map that is merged with the properties of all parents and is inherited by all modules/resources of this type. By default, child properties override parent properties unless **properties-metadata** explicitly forbids it.

- **properties-metadata**: See section 11.

- **parameters**: an optional map that is merged with the parameters of all parents and is inherited by all modules/resources of this type. By default, child parameters override parent parameters unless **parameters-metadata** explicitly forbids it.

- **parameters-metadata**: See section 11.

```
...

module-types:
 - name: java.tomcat
   extends: java
   parameters:
     memory: 256M
   properties:
     TARGET_RUNTIME: tomcat
```

```
...
resource-types:

 - name: postgresql
   extends: managed-service
   parameters:

     service: postgresql
     service-plan: v9.4-large
   properties:
     statistics-enabled: true
...
```

Example 25: Creating custom module and resource types.

12. Deployment Order

Until now, the MTA specification did not define any way for developers to influence the order in which their modules are deployed. As a result, deployers were free to choose any ordering algorithm, and an MTA that was deployed in one order by deployer X could have been deployed in an entirely different order by deployer Y.

We decided to standardize the ordering algorithm in the specification, so that modules can be deployed in a predictable and consistent order by every deployer. For this purpose, we introduce an optional module-level attribute in deployment descriptors called **deployed-after**, which must contain a list of module names. A module with this attribute is deployed only **after** the modules listed in the attribute have already been deployed. The relations expressed through this attribute are also transitive, so if module A is deployed after module B, and module B is deployed after module C, then it follows that module A should also be deployed after module C.

```
modules:
 - name: ui
   type: javascript
   deployed-after: [ backend, metrics ]

 - name: backend
   type: java
   deployed-after: [ hdi-content ]
   requires:
    - name: metrics
      properties:
        METRICS_URL: ~{url}
 - name: metrics
   type: javascript
   deployed-after: [ hdi-content ]
   provides:
    - name: metrics
      properties:
        url: ${default-url}
```

```
    - name: hdi-content
  type: hdi
```

Example 26: Using deployed-after to alter deployment order.

In the example above, the **deployed-after** attributes guarantee that the ui module is deployed after the backend and metrics modules, and that they are deployed after the hdi-content module. Note that the order in which the backend and metrics modules should be deployed is not specified in the attributes. The deployer is therefore free to choose **any** ordering – even random or parallel.

Also note that requires-provides relations do **not** affect deployment order **unless the deployer decides to use them for this purpose**. The specification does not restrict deployers in implementing additional ordering algorithms, as long as they also support the **deployed-after** based algorithm.

Similar to the **deployed-after** attribute, we offer the **processed-after** attribute to specify resource processing order. Aside from them being handled before the modules, there previously was not a way to specify ordering for resources. Ordering was decided by the deployer - sequential or in parallel.

Much like with modules ordering - **processed-after** is an optional resource-level attribute, which may contain a list of other resources. The deployer takes care of processing the resource after all the listed resources have already been processed themselves. Again, this relation is transitive.

```
resources:
  - name: my-first-service-instance
    type: managed-service

  - name: my-second-service-instance
    type: managed-service
    processed-after: [ my-first-service-instance ]

  - name: my-third-service-instance
    type: managed-service
    processed-after: [ my-second-service-instance ]
```

Example 27: Using processed-after to fully order the processing of three resources.

It's important to note that **processed-after** and **deployed-after** are not interchangeable and are specific for resources and modules respectively. You can only order resources with respect to other resources and modules with respect to other modules. Following this logic - a module name listed in the **processed-after** attribute will result in an error from the deployer or will be ignored. The same is true for resources listed in **deployed-after**.

13. Hooks

There are use cases for executing custom actions at specific points in an MTA process (deployment, undeployment, and so on). Examples include sending a notification after an application is updated, performing a graceful shutdown, initialization, or even database changes. For this purpose, the MTA specification defines the concept of “hooks”.

Hooks are predefined actions that can range from a simple network call to a complex script. These shall be defined via the top, module and resource-level attribute **hooks**. The type of the attribute is a list of objects with the following structure:

- **name**: an MTA internal identifier that must adhere to the following regular expression:

  /^[A-Za-z0-9_\-\.] +$/

  Can be used for documentation purposes and shown by MTA deployer tools.

- **type**: the type of action that should be executed. At this point, the MTA specification will not attempt to standardize the list of supported types, because different platforms can make the implementation of some types easy, while also making the implementation of others impossible. Due to this, MTA deployers have full control over what values they support for this attribute.

- **phases**: a list of strings that defines the points at which the hook must be executed. The list of supported phases is also defined by MTA deployers, because the platform-native entities they create from the MTA abstractions “modules” and “resources” can have different lifecycle phases.

  The list of phases supported by an MTA deployer may differ based on the location of the hook. For example, top-level hooks may support types A, B, and C, while module-level hooks may support A, B, and D.

- **parameters**: an optional map that can be used to configure/define the action that should be executed. The list of supported parameters is, as in all other MTA entities (modules, resources, etc.), deployer-specific.

- **parameters-metadata**: See section 11.

- **requires**: an optional list of required dependencies with the same structure as in other MTA entities. Can be used to obtain credentials or other information from a resource or a **provides** section.

```
modules:
 - name: backend
```

```
  type: application
  hooks:
   - name: cooldown
     type: task
     phases:
      - application.before-stop
     parameters:
      command: "sleep 30m"
   - name: send-notification
     type: http-request
     phases:
      - application.before-stop
      - application.before-stage
      - application.before-start
     parameters:
      method: POST
      host: ~{notification-service/url}
      body: >
       {
        "message": "..."
       }
      headers:
        Authorization: ~{notification-
service/token}        requires:
       - name: notification-service


resources:
  - name: notification-service
    type: service
```

Example 28: Using module-level hooks to define custom actions.

Hooks for the same module/resource **and** phase are executed in parallel, by default. Therefore, developers should not rely on any order of execution besides the one implied by the different hook phases. In the example above, “cooldown” and “send-notification” are executed in parallel before stopping the “backend” application (**if** it is stopped as part of the process). In addition, “send-notification” **may** be executed again for the other two phases it lists: “application.before-stage” and “application.before- start”. That would happen **if** the module’s corresponding application needs to be staged or started.

14. Escaping of References and Placeholders

The MTA specification introduces mechanisms for reducing duplication in MTA descriptors in the form of references (**~{reference}**) and placeholders (**${placeholder}**). For the sake of brevity and since the only difference between these constructs is the type of entities they can refer to (**properties** or **parameters**), we are going to refer to both as "MTA references".

Naturally, the very existence of MTA references prevents developers from using string literals identical to a reference, since the MTA deployer would not be able to distinguish between an actual MTA reference and the literal. Let's explore the following descriptor for an example:

```
modules:
 - name: backend
   type: application
   parameters:
     message: "Hello!"
     tasks:
       - name: echo_message
         command: "echo ${MESSAGE}"
   properties:
     MESSAGE: "Hello!"

resources:
 - name: db
   type: managed-service
   parameters:
     size: ~{default-size}
```

Example 29: Unescaped MTA references.

In the context of Cloud Foundry, this descriptor defines an application with an associated task, which is expressed as a Bash command containing a Bash placeholder: **${MESSAGE}**. Additionally, the descriptor also defines a database service with a creation parameter, whose value looks like an MTA reference (**~{default-size}**) but is intended to be resolved by a service broker.

Both MTA references should be escaped, so that the deployer knows to ignore them. For this purpose, the MTA specification defines the following syntax:

```
modules:
  - name: backend
    type: application
    parameters:
      message: "Hello!"
      tasks:
        - name: echo_message
          command: "echo \\${MESSAGE}"
    properties:
      MESSAGE: ${message}

resources:
  - name: db
    type: managed-service
    parameters:
      size: \~{default-size}
```

Example 30: Escaped MTA references.

NOTE: escaping an MTA reference in a quoted string requires two backslashes, whereas escaping a reference in a non-quoted string requires only one. This is a consequence of YAML's rules for escaping and not an MTA-specific detail.

15. Sample Application

Let us use a hypothetical sample application (the “app”) to illustrate the specified mechanisms.

The app provides product price optimization capabilities to retailers (see Figure 6). Price optimization is based on the retailer’s own pricing data held in an Amazon S3 storage, augmented with data from retail competitors. This competitor data is fetched from a web service of a market analysis company. The app is deployed to a Cloud Foundry-based platform, accessible at www.bestprice.acme.com. The app is composed of two modules and a service instance. The **pricing-ui** module provides static content. It also communicates with the external web service of the data provider company. The **pricing-backend** module runs on a Java EE server, validates data, performs optimization logic, accesses the storage service.

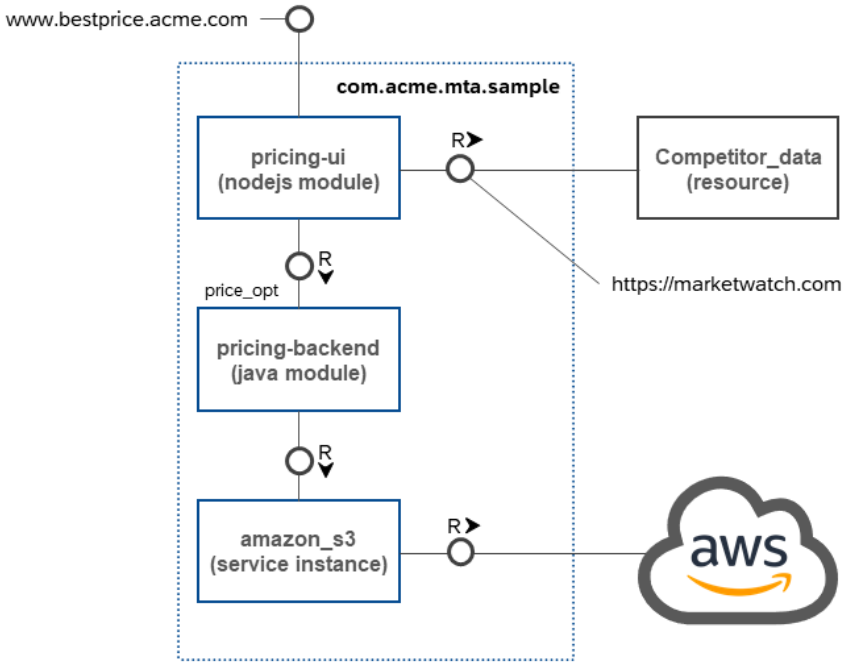

Figure 6: Logical view of the sample application.

15.1 Deployment view

The deployment descriptor on the left-hand side of Table 10 reflects the MTA structure of Figure 6. It is assumed that the deployer knows how to deal with the reserved module types and the resource type. Parameters (for example, a module’s host name) provide target specific configuration of modules and resources. Properties (for example, credentials to access the competitor price service) specify application runtime configuration.

The deployment descriptor does not specify all properties values because such values depend on the specific deployment scenario. For example, the properties **app_key** and **secret_key** have no value, because different credentials are used during development testing and for productive deployment. Leaving these properties without value allows the descriptor to be part of a software delivery (for example, when using an MTA archive). The missing values can be added during deployment, coded within MTA extension descriptors like the one shown in the right column of Table 10.

Deployment desctiptor	Extension descriptor
_schema-version: "3.1" ID: com.acme.mta.sample version: 1.2.1-beta modules: - name: pricing-ui type: javascript.nodejs parameters: host: www domain: bestprice.acme.com requires: - name: price_opt group: DESTINATIONS properties: name: internal url_path: ~{url_base}/odata/ - name: competitor_data group: DESTINATIONS properties: name: external url: ~{url} api_keys: ~{keys} - name: pricing-backend type: java.tomcat provides: - name: price_opt properties: url_base: ${host}.${domain} requires: - name: amazon_s3	parameters: memory: 128M instances: 2 buildpack: java-test resources: - name: amazon_s3 parameters: service-plan: basic - name: competitor_data properties: keys: app_key: 25892e17-80f6 secret_key: cd171f7c-560d

resources: **- name:** amazon_s3 **type:** org.cf.s3-cf-service **- name:** competitor_data **properties:** **url:** "https://marketwatch.com/" **keys:** **app_key:** **secret_key:**	

Table 10: Sample application deployment descriptor (left) and extension descriptor (right).

15.2 Equivalent cloud foundry API commands

We describe how a deployer for Cloud Foundry (CF) might transform the MTA descriptor into CF-relevant API commands. We represent most of the resulting commands by a CF *manifest.yml* file. We do not propose to physically create this manifest file. It would be better if a deployer uses the CF REST API. We use the manifest format just for logically representing the commands a deployer would send via the CF REST API. Additional commands (which cannot be coded into a CF manifest) are required. We only describe a "new deployment" without discussing the implications of re-deploying the app.

The transformation from the MTA deployment description into native CF API commands is possible if we assume that the deployer knows how to deal with the used module and resource types. For instance, using the resource type **org.cf.s3-cf-service** indicates to the deployer to generate a service instance using the s3 service broker (which is assumed to be running) and bind it to the requiring module. In detail, an MTA deployer for CF could proceed as follows:

1. To make CF applications names unique, the MTA ID is used to prefix all relevant names. It is assumed that MTA ID is unique.
2. The deployer scans the deployment descriptor (enriched with the extension depicted in the right column of Table 10).
3. The deployer detects the resource of type **org.cf.s3-cf-service**. Based on the service broker for Amazon S3, a service instance is created. Using a CF CLI, this would correspond to the following command:

   cf create-service org.cf.s3-cf-service basic amazon_s3

   The service plan has been configured via a parameter within the extension descriptor.

```
applications:


- name: com.acme.mta.sample.pricing-backend-blue
```

```
  path: ./backend
  memory: 128M
  instances: 2
  buildpack: java-test
  services:
   - amazon_s3


- name: com.acme.mta.sample.pricing-ui-blue
  path: ./web-server
  host: www
  domain: bestprice.acme.com
  env:
    DESTINATIONS: >
    [
     {
      "name": "internal",
      "url_path": "http://d0123-pricing-backend-blue.sofd605639a/odata/"          },
     {
      "name": "external",
      "url": "https://marketwatch.com/",
      "api_keys": {
        "app_key": "25892e17-80f6",
        "secret_key": "cd171f7c-560d" }
     }
    ]
```

Example 31: CF manifest that could be generated by an MTA deployer for CF.

4. The **pricing-ui** module requires information provided by the **pricing-backend** module and by the resource **amazon_s3**. By defining the group **DESTINATIONS**, the deployer is instructed to create the environment variable **DESTINATIONS** and pass the required property values as an array of JSON objects. To build this JSON string, the placeholders **${host}** and **${domain}** are replaced by default values known to the deployer.

5. The deployment descriptor declares a typed resource and an untyped one. A typed resource can be related to a service broker-managed service on CF and the developer can expect that the deployer allocates a service instance and passes the service configuration data via the generic VCAP environment variables. Untyped resources cannot be managed by the deployer. It is just used to attach configuration data as

property values, which then end up as JSON coded content within environment variables.

6. A manifest.yml might be created with the content shown in Table 11. For every module, a CF application section is created. We assume that the deployer offers a blue-green deployment strategy. Every redeployment of the MTA would therefore lead to an appropriate alternating renaming of CF applications. Initially, each CF application name gets the postfix **-blue** attached.

15.3 Benefits of using mtad.yaml instead of manifest.yml

As discussed in section 1.4, using a deployer for MTAs can provide advanced lifecycle management qualities. More benefits can be seen when comparing the role of the MTA deployment descriptor with the role of a CF manifest.

- **Team collaboration**: The MTA deployment descriptor provides a declarative description of an application and its required environment that can be shared within a development team. The CF manifest is specific for each developer. If one shared the manifest, each team member would need to repeatedly resolve merge conflicts to adapt to their specific parameters (for example, app names, domain names) each time the manifest get synchronized. In turn, every push into a version control system result in similar activities for all other team members. Even worse situations would be in cases where there are no merge conflicts and the manifest introduces new features, which are unwanted from the perspective of a specific developer. With MTAs, there can be a clear separation of sharable common parts and individual configurations captured by individual extension descriptors. In contrast, CF manifest is merely a convenience feature which saves typing by persisting repeatedly used command line interface instructions in a file.

- **Limited instruction set**: The CF manifest does not reflect the complete instruction set offered by the CF API. For instance, there is no way to reflect service instance creations or installations of a new service broker within the manifest. The deployment descriptor is more expressive in the sense that the contained declarative application description is transformed by the deployer into a sequence of any CF API instructions. This transformation is in principle not even limited to CF API instructions. If a CF instance provides services which offer their own API, the deployer might leverage these APIs as well. As an example, think of an MTA module containing process engine flow content which needs to be pushed to an appropriate process engine service running on a CF instance.

- **Sensible defaults**: The concept of parameters and placeholders within MTA descriptors can be used to shield developers from target specific details or defaults. Sensible defaults can be set by configuring the deployer. With that, they become effective for all developers simultaneously. At the same time, a developer or a person responsible for deployment can overrule such defaults if required.

16. References

[1] Jar File Specification: http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html

[2] The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html

[3] YAML Ain't Markup Language (YAML™) Version 1.2.

http://www.yaml.org/spec/1.2/spec.html