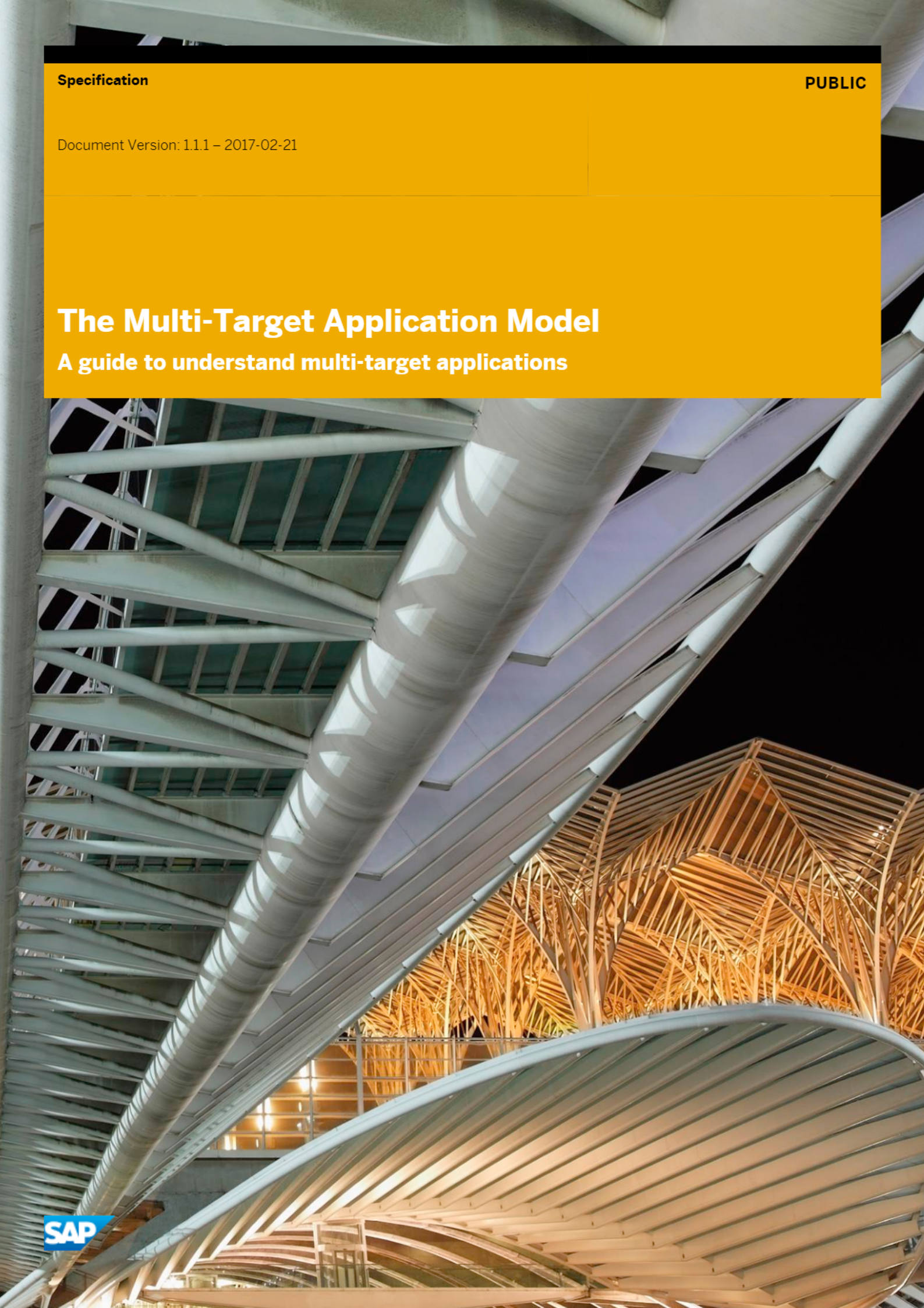
Specification

PUBLIC

Document Version: 1.1.1 – 2017-02-21

The Multi-Target Application Model

A guide to understand multi-target applications



Typographic Conventions

Type Style	Description
Example	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Textual cross-references to other documents.
Example	Emphasized words or expressions.
`Example`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE`	Keys on the keyboard, for example, `F2` or `ENTER`.

Document History

Version	Date	Change
1.0.0	May 31, 2016	Initial release
1.0.1	June 22, 2016	Corrections
1.0.2	June 30, 2016	Corrections
1.1.0	Sept. 21, 2016	Rework of some explanations, e.g., clarification between requires-provides relations and deployment order. Introduction of `build-parameters`.
1.1.1	Feb. 21, 2017	Corrections

Contents

1 Introduction

Recent trends and progress for programming languages, software design architectures such as micro-services, protocols like OData, and the diversity of multi-tiered and distributed deployment platforms have accelerated the trend towards applications constructed out of more, smaller, decoupled and diverse modules. Today, business applications are composed of multiple parts that are developed using different languages and technologies and deployed to a variety of target runtime environments.

This diversity introduces many life-cycle challenges. Developing, deploying and configuring all the separate parts of complex applications involves many steps, typically target-platform or application-server specific. Required services must be pre-configured and provisioned, the different modules must be "wired" together, configured, and deployed across multiple platforms in a strictly specific order, often using different tools, repeated for testing, staging and production environments. Zero-downtime upgrades is another complexity.

We coin the term **Multi Target Application** (**MTA**) to express this diversity of life cycle management requirements, and because other terms such as "distributed", "polyglot", "multi-module", "multi-tier", or "multi-headed" application, do not capture this diversity. But in essence, MTAs are just a natural evolution of existing multi-part applications.

For instance, **SAP HANA XS Advanced (XSA) applications**, consisting of UI and database modules, and even application code, are examples of MTAs. Developers and administrators want to manage develop, version, deploy and operate such a structured application as one logical unit.

Another example of an archetypical MTA is the Java EE application, composed of beans, web and application modules, resource adapters etc., all subject to the same development lifecycle and deployed across multiple computing tiers.

SAP HANA Cloud Platform introduces new distribution requirements for orchestrated cross-platform deployments. When acting as a SaaS extension platform and employing **Fiori as a Service (FaaS)** concepts, application developers need to distribute their applications across heterogeneous targets (Java VMs, Frontend Server, SaaS Backend), each with its own deployment APIs, while providing a, carefully managed, single application life cycle.

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 there is little uniformity. Each runtime, application server or cloud framework manages multi-target aspects in its own unique way, by introducing a variety of orchestration solutions (a multitude of manifest files and formats, project JSON files, app descriptors, repositories, SAP's **CTS+**, and more).

PaaS such as **Cloud Foundry** improve over traditional application servers in the flexible way they support diverse application runtime technologies through containerization. This introduces much more freedom to choose implementation technologies (Java, Node.js, Python, etc.). Applications can be decomposed into multiple modules which can be scaled independently and technologies can be chosen to best fit each module's concern. For instance, a scalable request pre-processing proxy implemented in Node.js might façade a Java module implementing business logic. While this is favorable from a runtime aspect, development and lifecycle management of such distributed applications gets harder.

The MTA specification addresses this lifecycle and orchestration complexity for cloud and on-premise platforms. We use the following definition:

> **Definition**
>
> A **multi-target application (MTA)** is comprised of multiple parts (**modules**), 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 (like Cloud Foundry '`cf push`'), via a formal **target- and technology-independent application model**. Developers describe the modules of the application, the dependencies to other modules, MTAs and (micro) services, and required and exposed interfaces. MTA-aware application lifecycle management frameworks validate, orchestrates and automate the MTA deployment on premise and on cloud platforms.

> **Note**
>
> SAP HANA XS advanced Model (XSA) 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.

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 SAP HANA Cloud Platform, XS Advanced and Cloud Foundry provide deployment services for these platforms. 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.

> **Note**
>
> At SAP, the individual MTA-aware tools involved in these different use cases will implement the full MTA specification over time. Today, each tool might reflect only a subset of the here described features, depending on priorities derived from the respective use case. You should consult the respective tool documentation to learn about the current implementation scopes.

2 MTA Concepts

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

> **Note**
>
> A module does not necessarily need to be code for execution in a runtime container. Instead, it could contain other artifacts required to make an application run. Consider, for instance, documents to be deployed to a documentation web server, or API metadata to be deployed to an API gateway, or configuration data to be deployed to a central registry.

This document defines the **MTA model**, a platform-independent description of the different application modules, their interdependencies and configuration data they expose, and the resources they require to run. This model is specified, using YAML (www.yaml.org), in descriptor files that accompany the development and deployment processes. An MTA model serves the following purposes:

1. Define an application composed of multiple (heterogeneous) sub-components (benefit: tools can establish a unique lifecycle of these sub-components)
2. Declare resources the application depends upon at runtime and/or deployment time (benefit: tools can allocate and bind such resources)
3. Define 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)

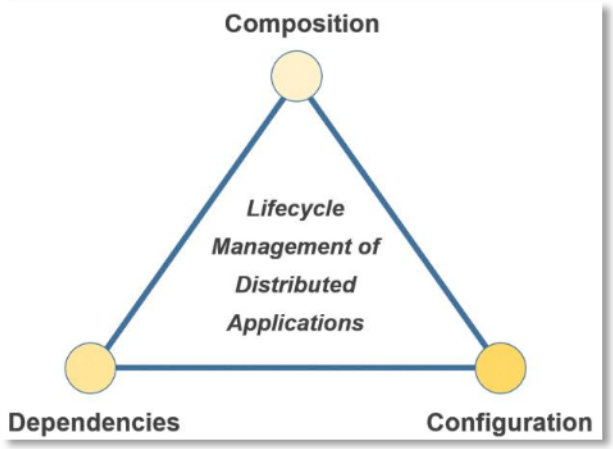


Figure 1: Three aspects to be considered when managing the lifecycle of a distributed application. These aspects are part of the MTA model.

The MTA model is the formal contract between developers (using development tools) and the **MTA deployer**. The deployer is a tool 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 (for instance, "service instances" on Cloud Foundry or SAP XS Advanced), and installing, running and updating the application modules. An MTA deployer may be more than a single tool, as it can include tools to maintain configuration and aggregate multiple target platform-specific deployers. Development environments contain such functionality as well, since deployment (e.g. for testing) is an integral part of the development process.

These main concepts are illustrated in Figure 2. An application developer uses development tools to create the

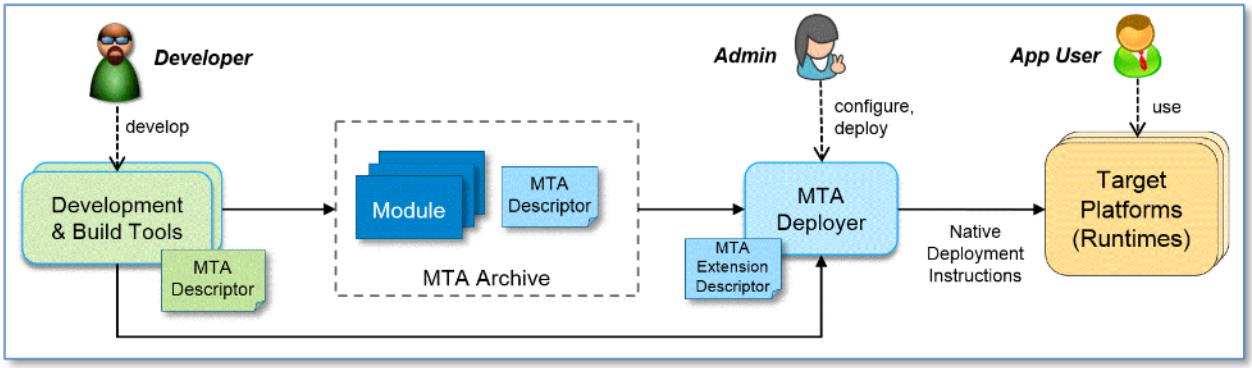


Figure 2: Basic development-to-deployment process flow.

modules of an MTA and the corresponding MTA descriptor. The application can then be distributed in the form of an **MTA archive** including the **MTA deployment descriptor**. This specification describes an archive format as a convenient distribution file. 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. 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, e.g., to set default values used for all deployments.

2.1 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, like Java or HTML5, which determines development tool choices, such as the tools to "build" it (e.g., Maven to build a Java source module). Build tools transform source modules into modules, e.g. a war-archive, which can be deployed to a target runtime. 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. All source modules of an MTA can be subject to the same versioning by a version control system (i.e., one unique version can be attached to all source artifacts).

A development cycle may result in a software release, requiring a distribution package. For instance, an application may be installed in a production landscape, or uploaded to a marketplace where it can be discovered by customers and deployed. 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 SAP XSA, the MTA archive plays the role of the Delivery Unit (DU) of SAP HANA 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 MTA project, possibly within one Git repository. MTA-aware tools use a **development descriptor** 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 & integration tests, and automatically create the MTA archive with its deployment descriptor. The development descriptor may even be generated from templates, wizards, as well as higher level abstractions.

Alternatively, developers can use any toolset to create and build the modules. In this case, they are also responsible for creating the deployment descriptor (and optionally the MTA archive) to be used by the MTA deployer.

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 is part of the MTA build, or part of continuous integration and deployment processes.

2.2 Deployment

While the MTA model is not target-specific, the MTA deployer is (multi) target aware. Even when targeting only one platform (e.g. Cloud Foundry or SAP XS Advanced), the MTA deployer is still needed to "wire" the different application parts.

Using MTA deployers rather than the underlying "native" installation tools, provides a single orchestrated deployment step for applications that may span multiple runtime tiers on more than one target platform, fulfilling many of the enterprise lifecycle management requirements, like:

- Auto-provisioning of required services
- Deterministic deployment order and timing (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, like routes, domains, technical names, technical users, etc. with centrally defined defaults and rules.
- 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.
- Zero downtime update (e.g. blue-green strategy), or update with maintenance page.
- Support of canary release strategies.
- Logging, tracing and coherent auditability of the entire application lifecycle.
- Implement version evolution policies, e.g., to protect against downgrades.
- Deletion of non-relevant modules and services (due to 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, etc.).

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

for any technology or target platform, it lays the ground for providing a uniform development and deployment experience.

3 The MTA Model

> **Note**
>
> This document reflects an MTA specification version 2.2. Future, versions might change and extend the MTA model elements presented here.

The MTA model is persisted in descriptor files using YAML (www.yaml.org). The following shows a very simple deployment descriptor:

```
_schema-version: 2
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
```

Example 1: Simple MTA descriptor

Throughout this document, within descriptor examples, we denote keys in descriptors defined by the MTA model specification by **`bold monospace font`**, whereas any other keys or values are indicated by `regular monospace font`.

3.1 Global Model Elements

An MTA descriptor has global 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 global MTA elements are supported:

`_schema-version`	This mandatory element is used to indicate to an MTA processing tool (e.g. 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. Schema versions have to follow the semantic versioning standard (semver.org) with the exception that trailing numbers (`<minor>.<patch>` or `<patch>`) can be omitted. A

	tool (e.g. 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., "2.0" is really interpreted as a string and not as a number. This number might be converted by a parser into 2, which has a different semantics than the "2.0" string.
ID	Mandatory string uniquely identifying the application. It has to match the regular expression `/^[A-Za-z0-9_\-\.]+$/`.
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 (semver.org). It has to be at least specified as `<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 instance 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.
modules	Begin of the section which declares all modules which constitute the MTA.
resources	Begin of the section which declares all external runtime or deploy-time dependencies of the MTA.

Table 1: Global MTA descriptor elements

3.2 Modules

Within the MTA development descriptor, the `modules` element declares the source modules of the MTA project. Within the MTA deployment descriptor, it declares the deployable modules.

The `modules` elements has the following sub-elements:

name	The mandatory name of a module. It must be unique within the MTA descriptor and is a purely MTA internal identifier. It has to match the regular expression `/^[A-Za-z0-9_\-\.]+$/`. A deployer must carefully decide if and how this name will be reflected on the target runtime 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 (having a different ID). One mitigation strategy is that deployers use the MTA ID as a namespace by prefixing generated runtime artifacts like as `<ID>.<name>`. Assuming uniqueness of the MTA ID, this prevents naming conflicts. Still, deployers shall implement mechanisms to detect naming conflicts and reject conflicting deployment requests. 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.
type	The type of a module indicates to tools how to interpret the module content. In a development descriptor, it can tell a builder how to generate a deployable module. In a deployment descriptor, it can tell a deployer which deployment mechanism to use for this module. The

	available module types are specified by the individual tools which are processing MTA descriptors. For instance, SAP Web IDE for HANA supports `HTML5`, `Node.js`, `HDB` and more source module types. The XS Advanced deployer supports modules of type `javascript.nodejs`, `java.tomcat`, `java.tomee`, `com.sap.xs.hdi`, and more. The deployer for SAP Hana Cloud Platform is supporting the module types `com.sap.hcp.html5`, `com.sap.fiori`, `com.sap.fiori.app`, `com.sap.java`, `java.tomcat`, and more.
`path`	A source module must have another mandatory `path` attribute. Usually, this is the file-system path from the root of the MTA project. Specifying the `path` of a deployable module is optional if deployment is based on an archive, where the additional MANIFEST.MF file contains the required path information (see below). In case of deploying an archive, the information in MANIFEST.MF has precedence over any path specification in the deployment descriptor.
`description`	An optional text, describing the application. This text is not meant to be visible on application user UIs.
`requires`	Starts an optional section which contains a list of required resources, or required parts of other modules which these provide (see below).
`provides`	Starts an optional section which contains configuration data which can be required by other modules within the same MTA (see below).
`properties`	A property is a named variable which can contain a string value. Properties are optional and are introduced to represent application specific configuration data (see below).
`parameters`	A parameter is a named variable which can contain a string value. Parameters are optional and are introduced by the tools which read the descriptor. They are used to steer the behavior of these tools. For instance, a parameter `memory` can instruct a deployer to allocate the appropriate amount of memory for a deployed module.
`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 an MTA build process, 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.

Table 2. Elements of a `modules` section within a descriptor

3.3 Resources

A **resource** is something which is required by a module of the MTA at runtime or at deployment time, but 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” to mean the metadata declaring a dependency to a real resource. A typical required resource can be for instance a relational database.

Resource declarations use the `resources` element with the following sub-elements:

`name`	The mandatory name of the resource. This name must be unique within the MTA descriptor and 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. It has to match the regular expression `/^[A-Za-z0-9_\-\.]+$/`.

type	The type of a resource is one of a reserved list of resource types supported by deployers. For some resources, the type indicates to the deployer how to discover, allocate or provision the real resource, e.g. a managed service like a database, or a user-provided service on XSA and Cloud Foundry. Other resources are not manageable by the deployer, e.g. 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, a "Destination" on SAP HANA Cloud Platform) that is then used by the requiring module to communicate with the resource. For instance, the XS Advanced 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`.
description	An optional text, describing the application. This text is not meant to be visible on application user UIs.
properties	A property is a named variable which can contain a string value. Properties are optional and are introduced to represent application specific configuration data (see below).
parameters	A parameter is a named variable which can contain a string value. Parameters are optional and are introduced by the tools which read the descriptor. They are used to steer the behavior of these tools. For instance, a parameter `service-plan` can instruct the XSA deployer to create a managed service instance based on the given service plan (see below).

Table 3: Sub-elements of a resource sections within a descriptor

3.4 Module Properties

Module `properties` are (possibly structured) key-value pairs that must be "injected" (made available) to the respective module at runtime. No assumption is made on how the MTA deployer injects the values. Many operating systems and languages like C, Java, node.js and Python support a concept of environment variables, but other methods will do as well (e.g. JNDI in the Java context).

> **Note**
>
> The MTA deployer for XSA provides properties as or within environment variables of the runtime container operating system which is used to run the module code.

To read the injected values, a developer's code will do an object lookup, and reference the module property names. This means, the used property names are determined by an application developer as these will have to be considered within the application code as well. For instance, on XSA, developers will do a lookup on environments variables which have the same name as properties in a descriptor.

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, see below), or inferred by the MTA deployer, for instance derived from a target platform configuration (see "placeholders" below). A deployer will report an error if it cannot determine a value for a property.

To further constrain the contract between deployer and application developer, 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 `properties` section must have a map structure at its first level. Any deeper levels can be structured in any way, as long as they can be transformed into a valid JSON object or array (see www.json.org for a definition of "object" and "array"). This is simply a special case of the requirement that every descriptor shall have an equivalent JSON representation, as described in section 4.3. This constraint allows identifying unique lookup keys (the keys 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 shall be used (see examples below).
- If environment variables are used, this means that one variable is created per map key and the name of the variable equals the key value (but see the concept of **group** below).

Examples of valid and invalid **properties** sections are shown in the table below. In the valid example in the left column, `company`, `email`, etc. are the names of environment variables.

```
...
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]
```

Example 2: Valid (left column) and invalid (right column) structure of module properties.

The left, valid descriptor from the example above, 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" ]
   }
]
```

3.5 Required Properties

In a majority of cases, a binding among modules and/or a binding of modules to resources must be performed. By binding, we refer to a process which generates and shares configuration data required to make the set of modules runnable.

For instance, certain modules consume APIs that are exposed by existing applications, by platform services, or by other modules in the same application. The challenge 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 via 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, sets of properties which are declared (provided) within the specification of other modules of the same MTA, or a resource (or even by other MTAs; this capability will be formalized in a next version of the specification). A module can provide multiple named property sets. A resource provides at most one property set.

The MTA deployer evaluates the dependency order, and transfers property values from the providing module to the requiring module and "injects" them to the module runtime, again, e.g. by creating environment variables. This is illustrated below:

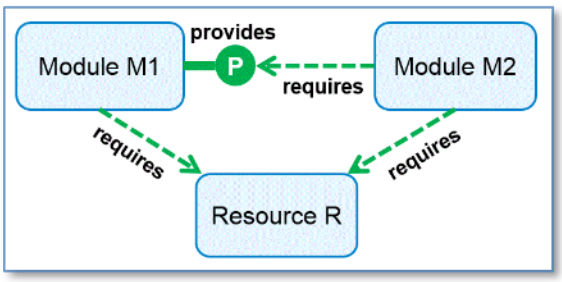


Figure 3: Requires-provides relations.

In this example, the MTA is composed of two modules and one resource declaration. Module **M1** depends on **R** ("requires 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**.

3.5.1 Requires-provides and deployment order

The MTA deployer will first determine all provided and required property values, then create/allocate required resources and then deploy all modules. Thus, requires-provides relations do not encode deployment order. Usually (and certainly for Cloud Foundry/XSA), the deployer cannot enforce deployment and startup order of modules under all circumstances. For instance, if a server running one or multiple modules of one MTA is restarted (due to a prior failure, for instance), any deployment order derived from an MTA descriptor cannot be reproduced.

Therefore, applications should not rely on any order in which a deployer is performing module deployments. This requires that they need to be built in a resilient way to survive temporal unavailability of services. Resilience is a quality which is very fundamental for the design of robust distributed systems.

Still, we specify a way to indicate to deployers to enforce a certain order of module deployment. This can be used, for instance, for those cases where building an appropriate resilient behavior is a high effort. To indicate a deployment order, one module declares a requires-relation to another module by requiring the module name.

3.5.2 Referencing required property values

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

- select a subset of provided properties
- combine provided property values and string literals into required property values
- add properties which are not provided somewhere else

To reference a provided property value, a tilde notation `~{<provided-property-name>}` is used. The snippet shown in *Example 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 *Example 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/" # using 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 3: 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 (e.g. an environment variable). This reduces the number of lookups done by the code of the requiring module (see *Example 4*).

```
…
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
```

Example 4: Provides-requires relation using a group assignment.

Based on the assignment of the group `API`, the deployer has to create a lookup object (e.g. 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.

3.6 Parameters

Parameters are reserved variables that affect the working of the 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 (e.g. 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 which is 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 (see section 4.3). 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 which case they have a default value that can be overwritten). Each tool (e.g. deployer) publishes a list of reserved parameters and their (default) values for its supported target environments.

Parameters values can be referenced by the **placeholder notation**, `${<parameter-name>}` (see *Example 5*). Examples of common read-only parameters are `user`, `default-host`, `default-url`.

```
_schema_version: 2
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:
        # referencing values of provided properties using ~{…}
        conn_string: ~{protocol}://~{url_segment}/odata/

 - name: pricing-backend
   type: java.tomcat
   parameters:
      domain: price.acme.com
   provides:
    - name: price_opt
      properties:
        protocol:
        # using placeholders ${…} to read parameter values
        url_segment: ${host}.${domain}
```

Example 5: Using placeholders to read parameter values.

Other parameters are writable, i.e., their value can be specified within a descriptor. For instance, 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 are shown in *Example 5*. There, the deployer will replace the reserved `${host}` by a value chosen by the deployer for module `pricing-backend` at time of deployment. The descriptor author decided to choose a value for parameter `domain` by himself, so that in this case `${domain}` resolves to `price.acme.com`. In turn, these values are considered when determining the value for property `conn_string`. Further, 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 below).

4 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.

4.1 Development vs. Deployment Descriptor

A developer can use any development and build tool chain to eventually create deployable modules together with the MTA deployment descriptor. Optionally, this developer packages all these artifacts into an MTA archive, as specified in section 5, for distribution purposes. This process will only require writing the deployment descriptor `mtad.yaml`. The deployment descriptor will then be checked-in to a version control system.

Another option has been implemented by SAP Web IDE. There, the IDE is "MTA-aware", i.e., MTAs can be first class citizens of 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 perspective within the development descriptor called `mta.yaml`. Now, `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 (the build). Therefore, `mta.yaml` declares modules of source code, whereas `mtad.yaml` declares deployable modules. This separation of concerns allows, for instance, to deal 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 extreme, an IDE can even hide the file mta.yaml completely from the developer, making it an internal artifact too, 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.

4.2 Extension Descriptors

Descriptor files can be extended via a concept of extension descriptors, whose contents are merged with the main descriptor to derive 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:

- Development time: While a developer is the primary author of the development descriptor, there might be other roles that add properties. For better team work, it might be useful to author extensions within separate files. The MTA build process has to merge the development descriptor with any available extension before doing the actual build.
- Deployment time: In principle the developer-created deployment descriptor can contain all information required to deploy the MTA, but this is unlikely in reality. Usually, deployment specific configuration will not be 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 has to declare which other descriptor it is extending. This allows building extensions chains or trees 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 shall be coordinated in the following way:

- Create a graph where each node is an extension descriptor and each `extends` statement is a directed edge connecting each descriptor with the descriptor it extends.
- Verify that there are no circles and that the graph is connected, and that the development or deployment descriptor is the root of the graph.

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

4.2.1 Extension rules

What does it mean to "merge" an extensions?

- You can only add information via an extension, not change (= overwrite), or delete. The following additions are allowed:
  - Add property and/or parameter elements (= name of a property or parameter)
  - Add property or parameter values to an existing property element without value
  - The following can be added via an extension to a deployment descriptor only (not to a development descriptor):

    `target-platforms: <JSON array of target platforms>`

    This instructs the deployer to use a specific deploy target. For instance, a deployer might support a DEV target platform and a PROD target platform. Technically, a JSON array is specified because the MTA modules might spread across multiple target platforms.

  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: 2
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
```

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

- Parameters/properties without values: If a parameter/property within a deployment descriptor is listed without any value, then it must get a value assigned by an extension descriptor. Otherwise, deployment shall 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` (see *Example 7*).

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

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

In particular, 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.

4.3 Validating Descriptor Files

As the MTA model is represented by a descriptor file (or a set of files, see below), the problem of model validation must be solved by validating the syntactical correctness of the file content. The MTA concept prescribes that there are three levels of actors or authorities which define what the validation rules are (see Figure 4).

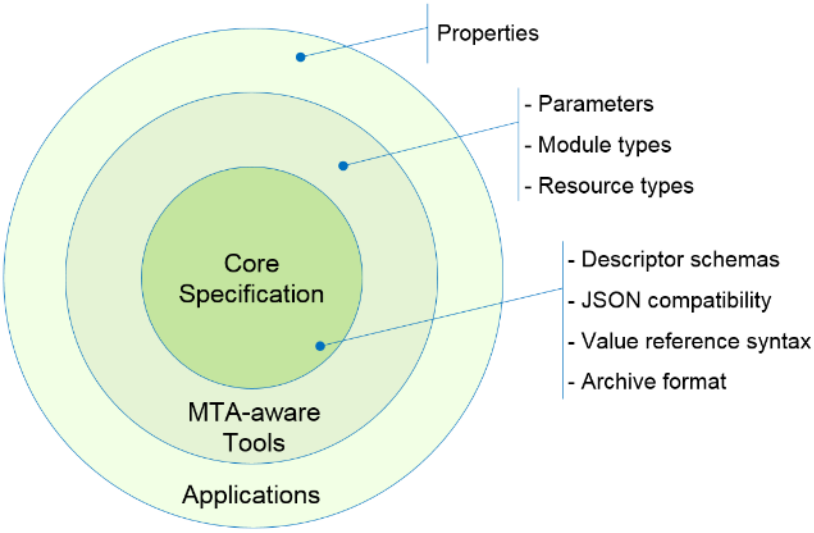


Figure 4: Responsibilities for specifying descriptor validation rules.

This document represents the **MTA core specification** which includes:

1. Descriptor schemas documenting which key elements can be used within descriptor YAML files. For instance, there is a `modules` key which must have `name` and `type` keys as sub-elements. There is a formal schema definition language for YAML files (an analogue to XSD for XML) which is used by SAP for MTA descriptor schema definition. In the future, such schemas might be published as well. In addition to this schema definition there are some additional restrictions:
   a. All key names used in descriptors are case sensitive, e.g. "**`MODULES`**" or "**`Parameters`**" are not valid. Only "**`modules`**" and "**`parameters`**" will be 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, the complete string value must be composed out of upper-case letters A-Z, and/or lower-case letters a-z, and/or digits 0-9, and/or the underscore "_", and/or the minus dash "-", and and/or the period "**.**".No length limit is specified, although for instance, 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 descriptor key **`path`** represents a file system path, then it must be interpreted as relative paths and must be path-traversal-safe. This means, **`path`** must then not begin with a slash "/", or backslash "\", or two periods "..".
   e. As **`properties`** and **`groups`** will often be converted into environment variables, certain restrictions for allowed characters have to be considered. Such restrictions are described in [3] as part of the POSIX.1-

2008 standard. Basically, names shall be composed of letters, digits and the underscore ("_") and must not begin with a digit. Some shells and utilities even restrict to capital letters.

f. Tools validating and parsing an MTA descriptor shall take care that map keys have unique names. This is typically not validated by common YAML parser libraries. Uniqueness of key names shall avoid to work with ambiguous descriptors, like this example:

```
ID: com.acme.mta.sample
version: 1.0.1-beta
version: 2.0.1
```

The version of this descriptor would not be well defined.

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.

This 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 which can be used to package and easily distribute applications.

The next level of specification authorities is represented by **MTA-aware tools**, i.e. all tools which will interpret MTA descriptors. For instance, design time tools, build tools and deploy tools. Each of these tools can specify its own set of module types, resource types and parameters. There is value in aligning these specifications as far as it makes sense, because this increases portability of applications. Such an alignment would be required at least for module and resource types. Parameters can be specified in additional files (called "extension descriptors" as described below), which are different to the deployment descriptor delivered with the application. In 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 which is used for application configuration. This set is highly application specific as property names will be referenced in application coding as well, for instance, when accessing environment variables matching property names.

5 MTA Archives

MTA archives are created in a way compatible with the JAR (Java Archive) specification. 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 in which an archive doesn't contains all MTA modules that are defined in the descriptor. An example may be a module with an unsupported module type. Another means will be used to deploy such a module, while the "rest" of the MTA can be bundled into the MTA archive, and can be handled in the regular way.

To deal with such "partial" archives, the deployment descriptor does not necessarily contains information concerning the location of modules within the archive (see also optional path attributes described in section 3.2). This location aspect (amongst others, like hash values and signatures) is added by the archive manifest `MANIFEST.MF`.

The deployment descriptor shall be located within the `META-INF` folder of the JAR.

The file `MANIFEST.MF` shall contain 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 required to add a row

`MTA-module:` `<module name>`

to each name section which 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 which 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 which points to a non-existing file or directory in the archive, an error shall be raised by the deployer.

If there is an MTA module entry in the manifest which refers to a module name which is not listed in the deployment descriptor (`mtad.yaml`), a warning should be raised 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 resources. Such a binding is useful, for instance, if 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 (e.g. a war-file). A file within a module must be considered to be inaccessible by the deployer.

To establish such bindings, the keys **`MTA-Resource`** and **`MTA-Requires`** are introduced as shown in *Example 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
…
```

Example 8: Example for using MTA-Resource and MTA-Requires to encode 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 which 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.

5.1 Sample Archive

SAP is publishing the "TinyWorld" tutorial "Developing with XS Advanced: A TinyWorld Tutorial" on SAP Community Network. Corresponding code and descriptors will be published on github.com in the future. The 3-tier application built in this tutorial consists of three modules `tinyui`, `tinyjs` and `tinydb`. Building and packaging the entire application results in an archive which contains the folder and file structure shown in *Example 9*.

```
tinyui/
   buildresults.zip
tinyjs/
   buildresults.zip
tinydb/
   buildresults.zip
META-INF/
   MANIFEST.MF
   mtad.yaml
xs-security.json
```

Example 9: File and folder structure within the MTA archive of the TinyWorld sample application.

The first three folders contain the build results for the individual modules which are the deployable artifacts. SAP Web IDE for HANA (which has generated this archive) is creating zip archives containing HMTL5/JavaScript artifacts and HANA database content. If the application would contain a Java module, a jar- or war-file would be included.

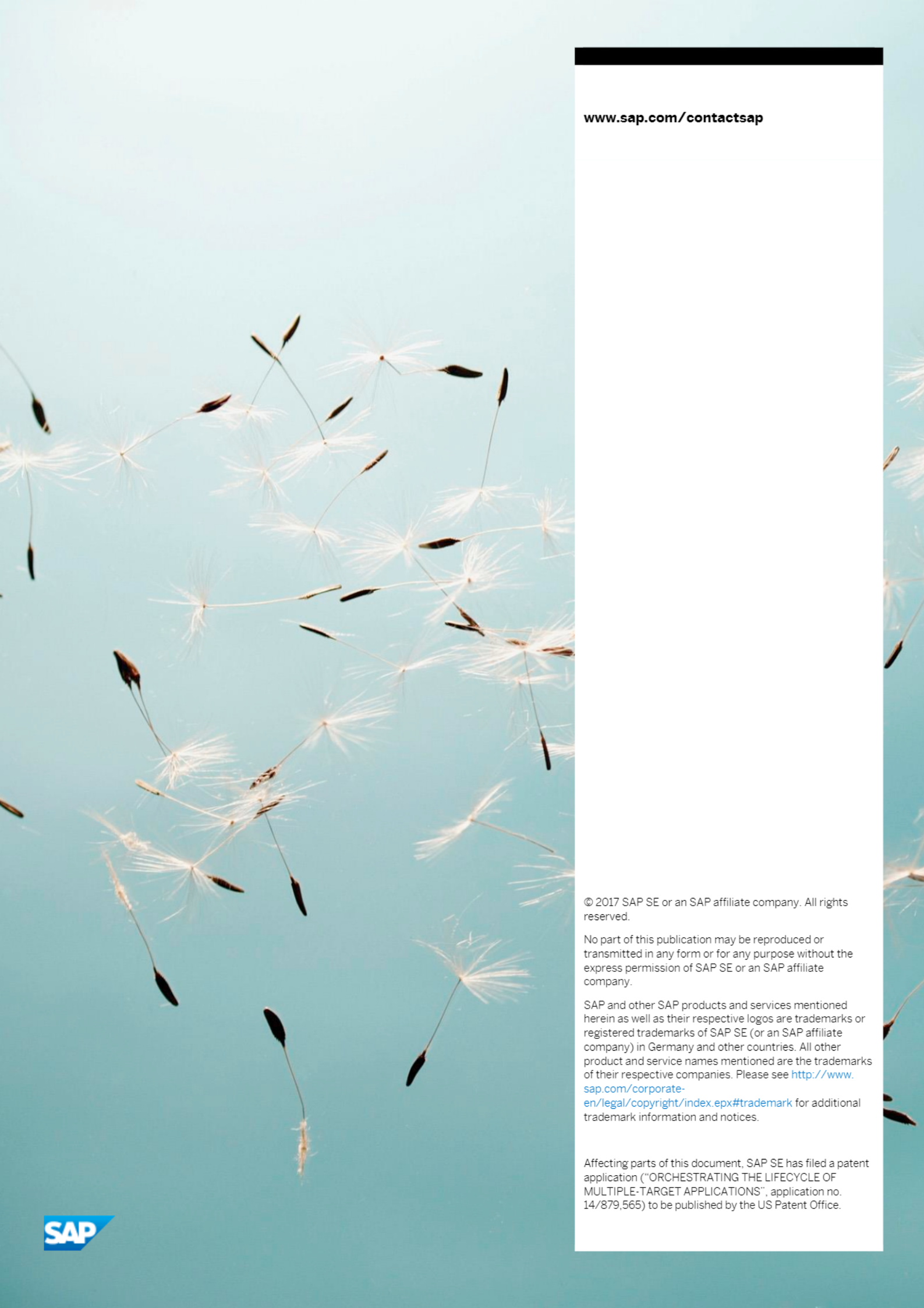
An additional file `xs-security.json` is part of the archive which contains configuration for authentication and authorization (see the TinyWorld tutorial for details).

As prescribed for jar-archives, additional metadata file `MANIFEST.MF` is contained within the `META-INF` folder. In our example, this file has the following content:

```
Manifest-Version: 1.0

Name: tinyui/buildresults.zip
MTA-Module: tinyui
Content-Type: application/zip

Name: tinydb/buildresults.zip
MTA-Module: tinydb
Content-Type: application/zip

Name: tinyjs/buildresults.zip
MTA-Module: tinyjs
Content-Type: application/zip

Name: xs-security.json
MTA-Resource: tiny_uaa
Content-Type: application/json
```

Example 10: Content of `MANIFEST.MF` of the TinyWorld application.

The three **`MTA-Module`** sections list the path to the deployable modules within the archive and associate them with the corresponding module name used in the deployment descriptor `mtad.yaml`. The **`MTA-Resource`** section lists the path to file `xs-security.json` within the archive and binds it to the resource `tinuy_uaa` which is declared in the deployment descriptor `mtad.yaml`. The file `xs-security.json` contains authentication and authorization relevant configuration. It is used by the XSA deployer when creating and configuring a service instance of the uaa service.



www.sap.com/contactsap

Affecting parts of this document, SAP SE has filed a patent application ("ORCHESTRATING THE LIFECYCLE OF MULTIPLE-TARGET APPLICATIONS", application no. 14/879,565) to be published by the US Patent Office.