SMPTE ST 2133:202x

SMPTE Public Committee Draft

Job Processing Architecture

Page 1 of 14 pages

Table of Contents Page

Foreword

SMPTE (the Society of Motion Picture and Television Engineers) is an internationally-recognized standards developing organization. Headquartered and incorporated in the United States of America, SMPTE has members in over 80 countries on six continents. SMPTE's Engineering Documents, including Standards, Recommended Practices, and Engineering Guidelines, are prepared by SMPTE's Technology Committees. Participation in these Committees is open to all with a bona fide interest in their work. SMPTE cooperates closely with other standards-developing organizations, including ISO, IEC and ITU.

SMPTE Engineering Documents are drafted in accordance with the rules given in its Standards Operations Manual. This SMPTE Engineering Document was prepared by Technology Committee 34CS.

Normative text is text that describes elements of the design that are indispensable or contains the conformance language keywords: "shall", "should", or "may". Informative text is text that is potentially helpful to the user, but not indispensable, and can be removed, changed, or added editorially without affecting interoperability. Informative text does not contain any conformance keywords.

All text in this document is, by default, normative, except: the Introduction, any clause explicitly labeled as "Informative" or individual paragraphs that start with "Note:"

The keywords "shall" and "shall not" indicate requirements strictly to be followed in order to conform to the document and from which no deviation is permitted.

The keywords "should" and "should not" indicate that, among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited.

The keywords "may" and "need not" indicate courses of action permissible within the limits of the document.

The keyword "reserved" indicates a provision that is not defined at this time, shall not be used, and may be defined in the future. The keyword "forbidden" indicates "reserved" and in addition indicates that the provision will never be defined in the future.

A conformant implementation according to this document is one that includes all mandatory provisions ("shall") and, if implemented, all recommended provisions ("should") as described. A conformant implementation need not implement optional provisions ("may") and need not implement them as described.

Unless otherwise specified, the order of precedence of the types of normative information in this document shall be as follows: Normative prose shall be the authoritative definition; tables shall be next; then formal languages; then figures; and then any other language forms.

Introduction

This clause is entirely informative and does not form an integral part of this Engineering Document.

The reality of microservices in the media industry has often come up short of its promise. This document aims to remedy a large part of that. The promise of microservices, and the vision of this document, is an environment in which implementations can be completely agnostic when it comes to both vendors and products used and platforms.

Allowing implementations to have the flexibility and agility to select the most appropriate microservices to utilize at the time they are needed is key. For example, when the completion of a task is time-sensitive, the user can prioritize speed above all else, but that same task when executed in a non-urgent scenario could prioritize cost or carbon footprint over speed.

This can be accomplished by having a registry or repository (in effect, a private marketplace) for media microservices to which jobs can refer each time they are run, allowing them to choose the most appropriate microservice(s) for that particular instance of the job.

In summary, this document is envisioned as ultimately enabling the following capabilities for media microservices:

- A simple and standardized method for grouping microservices into jobs in such a way that those microservices work together to perform a useful overall task.
- Enable vendor and platform agnostic implementation and deployment of these jobs.
- Allow for agile and flexible selection of microservices at time of execution, based on a variety of parameters (e.g., price, speed, carbon footprint).
- Creation of a media microservice repository which jobs can use to select the best service for a required task.

The canonical definition of the API was developed using OpenAPI Specification v3.03. The standardized version of this API uses a YAML representation (see 5.7 Source Code in the OpenAPI Specification v3.03 document). The OpenAPI documentation provides additional clarification on the semantics and use of the fields of the API.

At the time of publication, no notice had been received by SMPTE claiming patent rights essential to the implementation of this Engineering Document. However, attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. SMPTE shall not be held responsible for identifying any or all such patent rights.

1 Scope

Defines an Application Programming Interface that provides a standardized framework, allowing the grouping of related services into reuseable units of work that can be instantiated as “jobs”. It builds upon the work of other initiatives, particularly the European Broadcasting Union’s Media Cloud and Microservices (EBU’s MCMA) group.

2 Normative References

There are no normative references in this document.

3 Terms and Definitions

For the purposes of this document, the following terms and definitions apply:

3.1
job

stored set of processes that can be used to perform a specific piece of work or operation on media files repeatedly and as needed

4 Overview

To begin, let us define the problem this document attempts to solve.

Presently, there is no standardized method for media microservices to be abstracted into jobs in a way that would allow for easy deployment and management across infrastructures and platforms.

Serverless infrastructure is extremely desirable for its ability to effectively and inexpensively scale up and down with the needs of the customer. However, an abstraction of services is needed to make this work across vendors and platforms.

The ability to dynamically select the most appropriate microservices each time a job is initiated is seen as one of the most compelling reasons to embrace a microservice architecture by most media professionals, but does not currently exist in any standardized form. Today's marketplaces, managed by platform providers, provide neither a framework for the required flexible deployment of media jobs/workflows, nor a curated collection of enterprise-class services that are appropriate for use by professional media organizations.

5 RESTful design

The API uses a RESTful pattern where the HTTP verbs have the following meanings:

GET /services - retrieve a list of services.

POST /services - create a new service.

PUT /services/{id} - idempotent method to update a service.

DELETE /services/{id} - destroy a service.

GET /job-profiles - retrieve a list of job profiles.

POST /job-profiles - create a new job profile.

PUT /job-profiles/{id} - idempotent method to update a job profile.

DELETE /job-profiles/{id} - destroy a job profile.

6 OpenAPI definition

The API can be represented by the OpenAPI specification, and a full, testable representation is available in **element a** of this document (see Annex A). Note that **element a** is provided as a ZIP file that contains multiple YAML files.

7 Job and Services Architecture

7.1 Framework

This architecture provides a standardized framework for deployment and management of services across infrastructures and platforms.

7.2 Job

A job can be considered as a unit of work that can be performed by one or more microservices. An example of a high-level job could be the scope of ingesting of a piece of content. The high-level job can be assimilated into a media workflow and decomposed into three separate jobs:

- The upload of the media file.
- The extraction of technical metadata.
- The creation of a proxy copy of the content.

Each job may be performed by at least one microservice capable of handling the specific job type. A job can be described by its properties, aggregated together they form the job profile.

This architecture makes that unit of work reuseable by storing a job profile, meaning a set of processes that may be used to perform a specific piece of work or operation on media files repeatedly and as needed. Job profiles shall be stored in a library (the job profile repository, itself a component of a service registry) and contain the information necessary for them to be appropriately assigned to a service for execution.

7.3 Service Model

Figure 1 shows the minimum environment required to manage jobs, and consists of a job processor and a service registry.

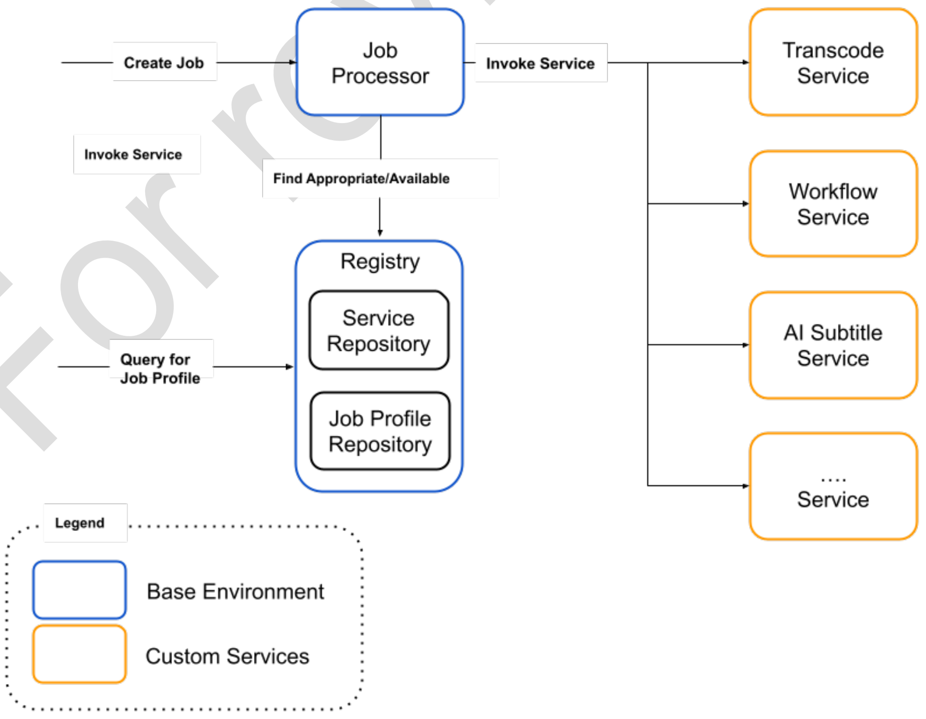


Figure 1 — Job Processing Architecture Service Model

7.4 Base Environment

In this architecture, a minimum required environment shall exist, comprising two services: a job processor and a registry, as described in Clauses 7.5 and 7.6, respectively, and any additional service capabilities required to perform work.

7.5 Job Processor

A job processor is the entry point for all requested processing once a job profile has been identified. It handles the routing of job requests to the services that can perform them.

The job processor also acts as a repository for jobs and is responsible for monitoring their execution and status, e.g., checking if a job has exceeded specified limits on processing time or cleaning up old job data when it is no longer needed.

7.6 Registry

A registry stores service capabilities (within a service repository) and job profiles (within a job profile repository) and makes them available for discovery. Each service stored within the service repository shall contain a reference ID to any job profile by which it may be called. The registry may be used both internally by the job processor to find services capable of processing jobs based on their supported job profiles and by external clients to find services that handle additional resources. The registry may also contain services that do not reference a job profile.

7.7 Custom Services

This simple model allows the creation of customized services and complex workflows, tailored to the particular requirements of each task or environment.

8 Data Model

8.1 Overview

Figure 2 shows high-level data structures for the various entities involved in this environment, including Jobs, Job Executions, Job Assignments, Job Profiles, and Services. YAML files illustrating each of these structures are available as an additional document element in st2133a.zip. It contains three YAML files, as shown below:

- job-processor.yaml – contains definitions for job, job execution, job profile, job assignment
- service-registry.yaml – contains definition for service
- mcma-service.yaml – contains definitions of REST API for an MCMA service

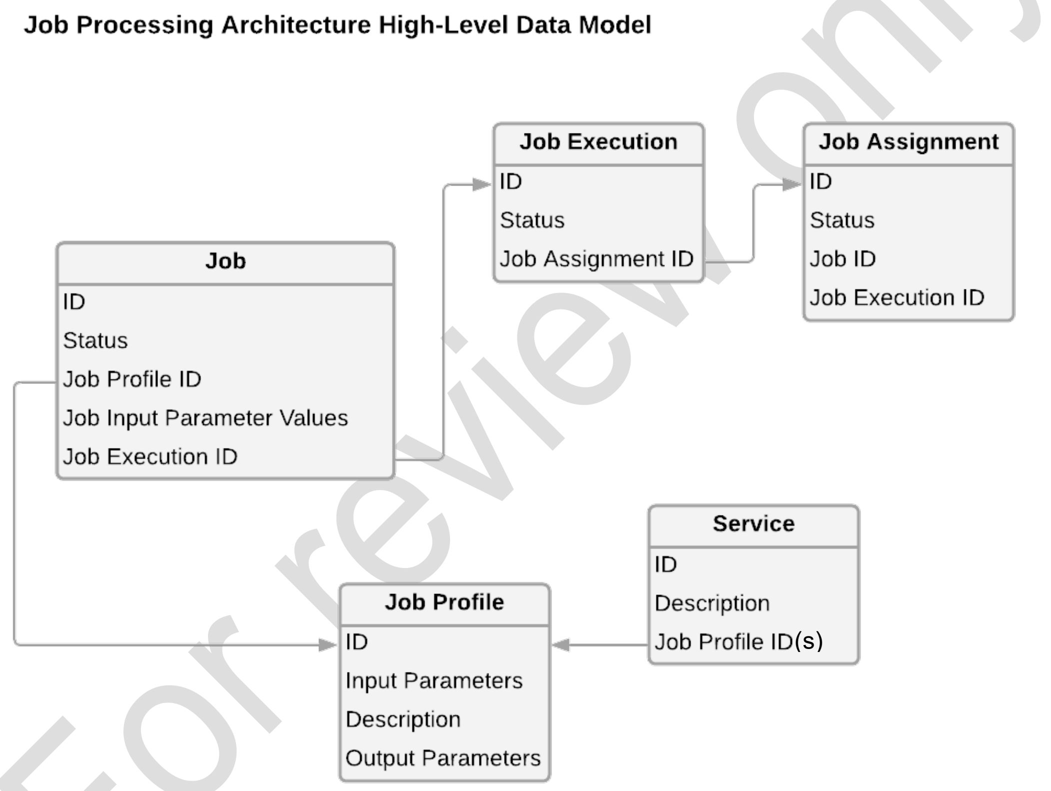


Figure 2 — Job Processing Architecture High-Level Data Model

8.2 Service and Data Model Overview

Figure 3 shows the Service Model, introduced in Figure 1, with the addition of the data payloads associated with the various Entities shown in Figure 2. Figure 3 provides a single view with all of the components of the Environment shown together.

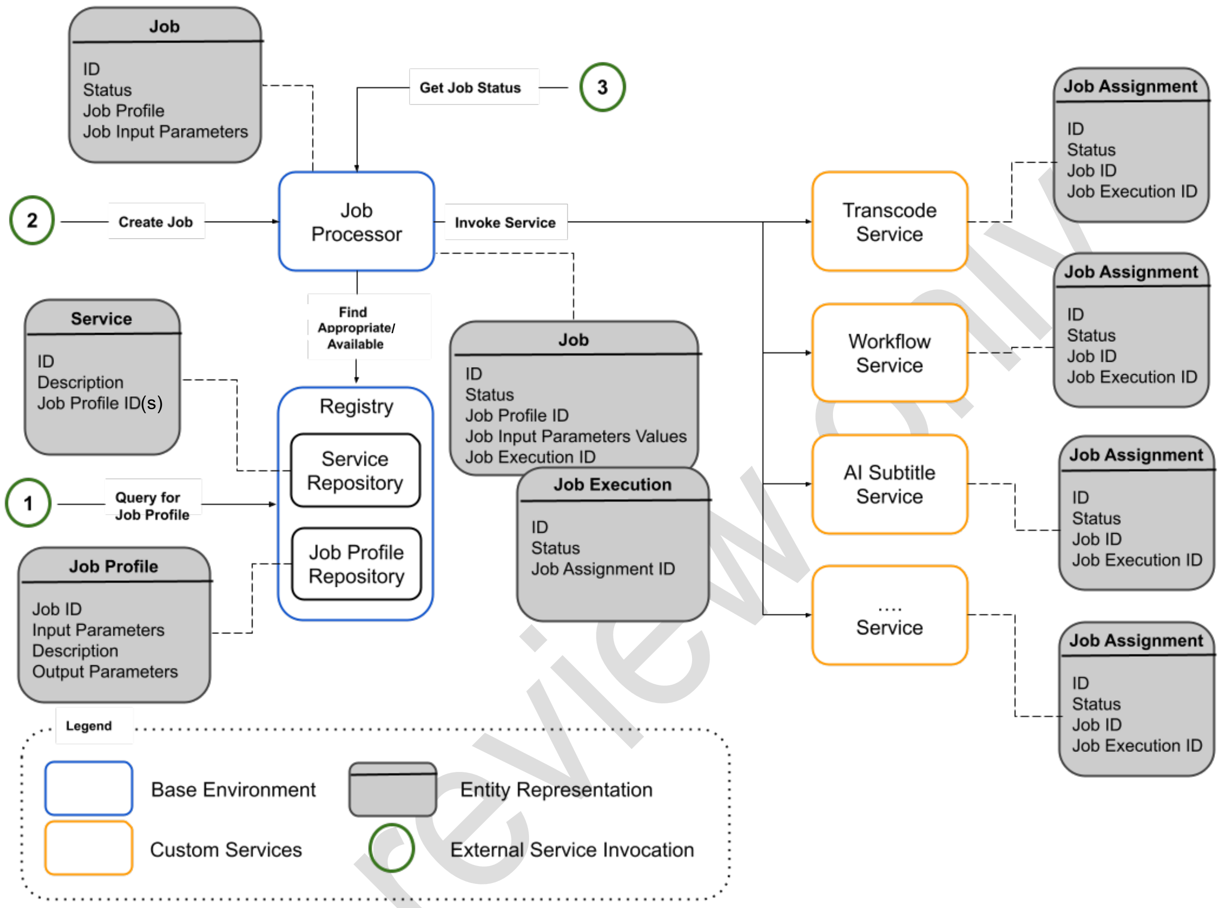


Figure 3 — Service and Data Model Overview

8.3 RESTful Implementation

Figure 4 shows the Data Model introduced in Figure 2 (Jobs, Job Executions, Job Assignments, Job Profiles, and Services), with example RESTful APIs associated with each data element. The URLs may reside on one or more hosts.

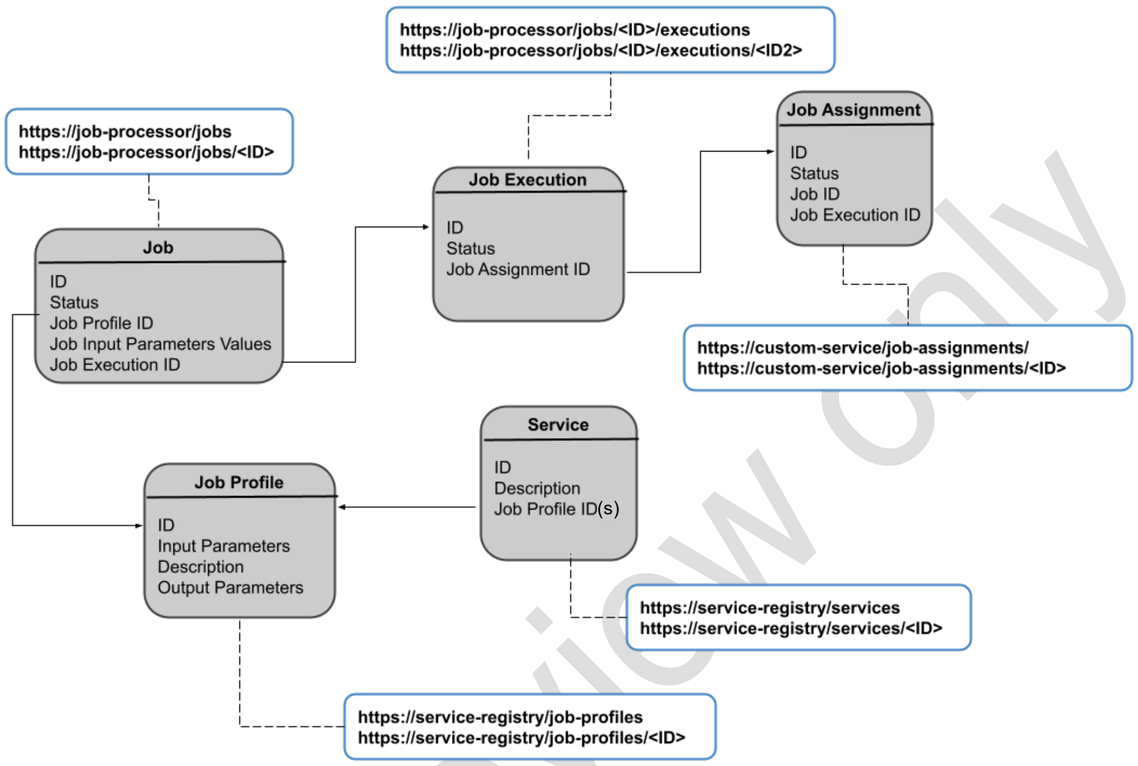
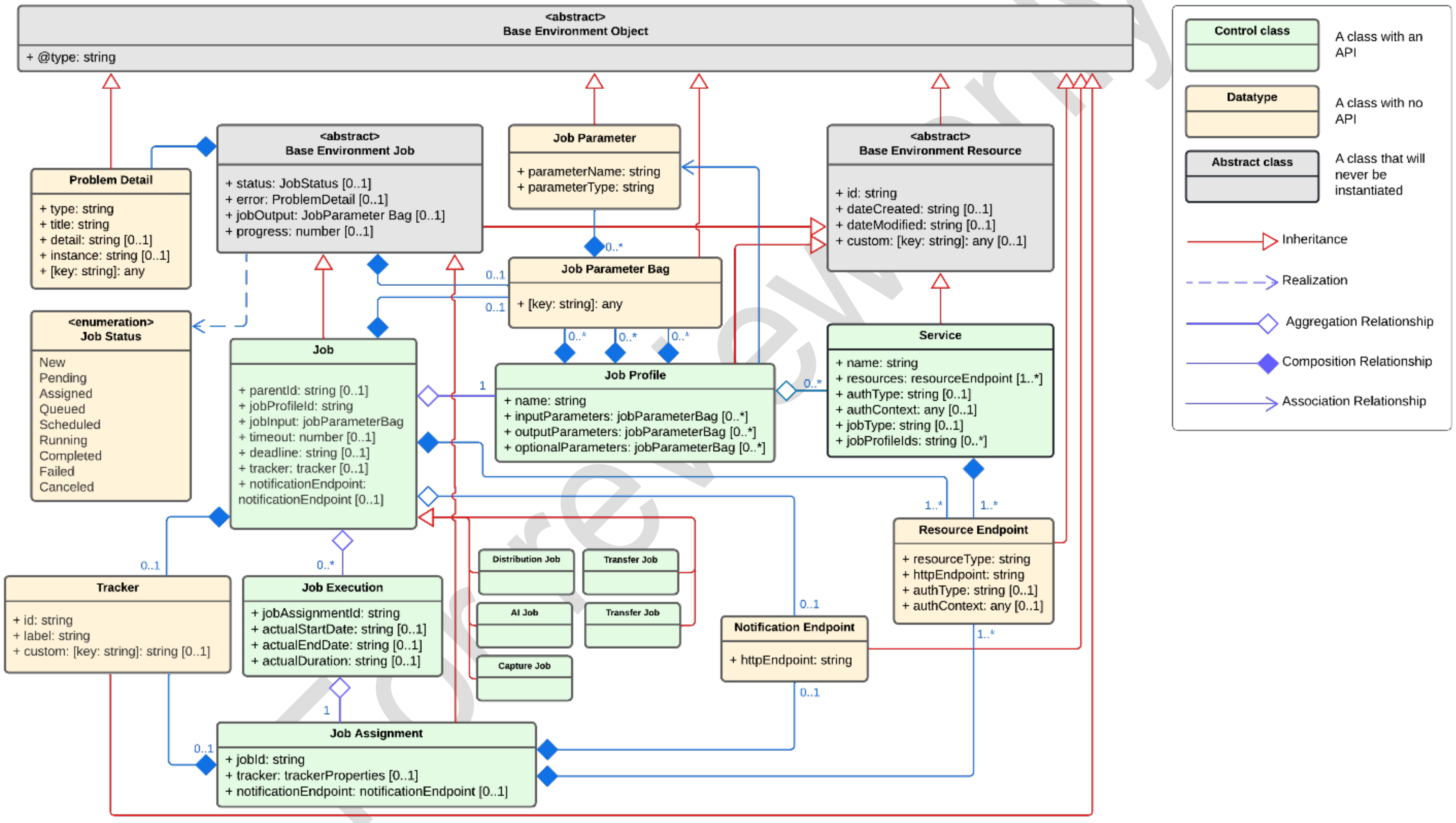

https://job-processor/jobs/<ID>/executions
https://job-processor/jobs/<ID>/executions/<ID2>
https://job-processor/jobs
https://job-processor/jobs/<ID>
Job Assignment
ID
Status
Job ID
Job Execution ID
Job Execution
ID
Status
Job Assignment ID
Job
ID
Status
Job Profile ID
Job Input Parameters Values
Job Execution ID
https://custom-service/job-assignments/
https://custom-service/job-assignments/<ID>
Service
ID
Description
Job Profile ID(s)
Job Profile
ID
Input Parameters
Description
Output Parameters
https://service-registry/services
https://service-registry/services/<ID>
https://service-registry/job-profiles
https://service-registry/job-profiles/<ID>

Figure 4 — RESTful Implementation

9 UML Class Diagram

To model this environment in a human-readable way, a set of UML diagrams are provided in this clause.

A complete UML Class Diagram, showing all of the various Entities involved in the full environment, is shown below in Figure 5.



<abstract>
Base Environment Object
+ @type: string
Problem Detail
+ type: string
+ title: string
+ detail: string [0..1]
+ instance: string [0..1]
+ [key: string]: any
<abstract>
Base Environment Job
+ status: JobStatus [0..1]
+ error: ProblemDetail [0..1]
+ jobOutput: JobParameter Bag [0..1]
+ progress: number [0..1]
Job Parameter
+ parameterName: string
+ parameterType: string
<abstract>
Base Environment Resource
+ id: string
+ dateCreated: string [0..1]
+ dateModified: string [0..1]
+ custom: [key: string]: any [0..1]
Job Parameter Bag
+ [key: string]: any
<enumeration>
Job Status
New
Pending
Assigned
Queued
Scheduled
Running
Completed
Failed
Canceled
Job
+ parentId: string [0..1]
+ jobProfileId: string
+ jobInput: jobParameterBag
+ timeout: number [0..1]
+ deadline: string [0..1]
+ tracker: tracker [0..1]
+ notificationEndpoint: notificationEndpoint [0..1]
Job Profile
+ name: string
+ inputParameters: jobParameterBag [0..*]
+ outputParameters: jobParameterBag [0..*]
+ optionalParameters: jobParameterBag [0..*]
Service
+ name: string
+ resources: resourceEndpoint [1..*]
+ authType: string [0..1]
+ authContext: any [0..1]
+ jobType: string [0..1]
+ jobProfileIds: string [0..*]
Resource Endpoint
+ resourceType: string
+ httpEndpoint: string
+ authType: string [0..1]
+ authContext: any [0..1]
Distribution Job
Transfer Job
AI Job
Transfer Job
Capture Job
Notification Endpoint
+ httpEndpoint: string
Tracker
+ id: string
+ label: string
+ custom: [key: string]: string [0..1]
Job Execution
+ jobAssignmentId: string
+ actualStartDate: string [0..1]
+ actualEndDate: string [0..1]
+ actualDuration: string [0..1]
Job Assignment
+ jobId: string
+ tracker: trackerProperties [0..1]
+ notificationEndpoint: notificationEndpoint [0..1]
0..*
0..1
0..1
0..*
0..*
0..*
1
0..*
1..*
1..*
0..1
0..*
0..1
0..1
1..*
1
0..1
0..1
Control class
A class with an API
Datatype
A class with no API
Abstract class
A class that will never be instantiated
Inheritance
Realization
Aggregation Relationship
Composition Relationship
Association Relationship

Figure 5 — UML Class Diagram

10 Source Code

Libraries containing the source code for implementing the features described in this document can be found here:

https://raw.githubusercontent.com/ebu/mcma-website/master/api/rest/0.13/job-processor.yaml

https://raw.githubusercontent.com/ebu/mcma-website/master/api/rest/0.13/service-registry.yaml

https://raw.githubusercontent.com/ebu/mcma-website/master/api/rest/0.13/mcma-service.yaml

NOTE For those who prefer a more graphical view of these, rather than the raw YAML, go to: https://editor.swagger.io/ and then copy and paste each of the above links into the **File > Import URL** dialog.

Annex A
(Informative)

Additional Elements

This annex lists non-prose elements, as shown in Table A.1 of this document.

Table A.1 — Additional Elements

Non-prose element	Description
a	**st2133a.zip – YAML files (normative)** This element contains YAML files that illustrate Jobs, Job Executions, Job Assignments, Job Profiles, and Service structures as follows: • job-processor.yaml – contains definitions for job, job execution, job profile, job assignment • service-registry.yaml – contains definition for service • mcma-service.yaml – contains definitions of REST API for an MCMA service

Bibliography (Informative)

Resources on MCMA are available in GitHub (https://github.com/ebu/mcma-libraries).

Example MCMA implementations using the Node.JS libraries can be found here:
https://github.com/ebu/mcma-projects.