Cloud Service Innovation Platform

User Manual and Technical Documentation

Olaf David
Wesley Lloyd
Mazdak Arabi
Ken Rojas

Cloud Service Innovation Platform: User Manual and Technical Documentation

by Olaf David, Wesley Lloyd, Mazdak Arabi, and Ken Rojas

This is a draft document. Service signatures, API details, URL references may change because of design refinements.

Publication date Mar 23 2015 (rev 6)

Abstract

Cloud infrastructures for modelling activities such as data processing, performing environmental simulations, or conducting model calibrations/optimizations provide a cost effective alternative to traditional high performance computing approaches. Cloud-based modelling examples has emerged into the more formal notion: “Model as a Service” (MaaS). This manual presents the Cloud Services Innovation Platform (CSIP) as a software framework offering MaaS. It describes both the CSIP infrastructure and software architecture that manages computational resources for typical modelling tasks, and the use of CSIP's “ModelServices API” for a modelling application.

This document specifies the structure, programming interface, and usage of model and data services as provided by the Cloud Services Integration Platform (CSIP). A client can use those services to obtain data, such as soils, management, and climate, or to execute simulation models by providing a model parameterization and then retrieve the results. This service specification is fully based on RESTful principles using JSON as payload for transferred data. The services also contain support for service registration, lookup, and payload template specification.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

1. Naming

Throughout this manual the term 'CSIP' refers to the *Cloud Services Innovation Platform*. This software originated as the product of a collaboration project with the goal to explore the usability of cloud computing and related technologies for service-oriented modeling and simulation. This manual will not reflect on cloud management aspects of CSIP, it only refers to the model services software architecture.

2. License

CSIP is free under the LGPL 2.1 License. The license is bundled with the distribution but can also be obtained from https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html directly.

3. Resources

All source code, examples and this documentation is available at http://csip.javaforge.com. The reader can also find active services for general use. Trackers are available at this site to manage bugs and feature requests.

The CSIP source can be checked out using the mercurial versioning system.

```
> hg clone http://javaforge.com/hg/csip-core
```

The repository contains a file `readme.txt` describing the structure and the process of building your custom services based on examples.

This project site allows also the creation of CSIP repositories once you become a member as a developer.

4. Acknowledgements

CSIP research and development is being funded under a cooperative agreement 68-7482-13-518 between the US-DA-Natural Resources Conservation Service and the Department of Civil and Environmental Engineering at Colorado State University.

Chapter 1. Introduction

The use of cloud computing for scientific computing and environmental modeling has gained substantial traction in recent years [Jha2011]. A cloud is a large pool of available and easily accessible virtual resources such as hardware, platforms, and software services. Such resources can be allocated and disposed in an ad-hoc manner. The dynamic configurability of cloud resources to various requirements makes it attractive for scientists, research groups, organizations, and agencies to explore their potential for research projects and operational use.

Appealing cloud features include: (1) the absence of in-house maintenance and administration of such resources, (2) the availability of a range of competing vendors offering different pricing models, (3) the flexibility in adjusting applications or operating systems rapidly on a large scale, (4) secure access and data protection, (5) governing the physical location of cloud-managed resources, and (6) guaranteed availability had to be addressed by commercial cloud vendors to make it a viable option for operational use within the modeling community and not just academia.

The Cloud Services Innovation Platform (CSIP) project was established at Colorado State University (CSU) in collaboration with the USDA Natural Resources Conservation Service and Agricultural Research Service to explore the prospect of service oriented cloud computing for MaaS and related data management. As a result of this research, CSIP was developed as a scalable, modular, cost effective, and open deployment platform for simulation models while leveraging new and legacy research simulation models as cloud based web-services.

1.1. Model-as-a-Service (MaaS)

A *Model as a Service* (MaaS) [Zou2012] provides the capability to execute simulation models on demand as webservices. MaaS solely focuses on the application aspect of a model against data.

Model-as-a-Service has emerged as “a concept of being able to invoke re-usable, fine-grained software components across a network” [Roman2009], [Argent2004]. MaaS enables model service providers to lower the burden for model users by supporting autonomic model parameterization and in the service within a scalable execution environment. The MaaS concept has evolved as a merge of the Model Web and Software as a Service (SaaS), SaaS is defined as a model of software deployment whereby a provider licenses an application to customers for use as a service on demand. The Model Web is defined as an approach to manage models on the Web ([Roman2009]).

MaaS may harness the HTTP protocol as the interface to enable client/server communication. Data is exchanged as structured text, e.g. XML or JSON- JavaScript Object Notation, in a specified syntax. Data may also be passed to the service using file attachments in the model’s native format.

There are two main usage patterns: (i) The model is pre-deployed, has a well-known service endpoint, and may be supported by supplemental data services. This deployment is quite common for operational models used in a production environment; (ii) the model can be dynamically deployed from the client before execution. Model service development for research purposes requires this behavior. Both approaches address different workflows, need for availability and security. The model execution method may be specified in the service.

1.2. Cloud Services Integration Platform

The Cloud Services Innovation Platform (CSIP) provides a simple and open framework to implement MaaS services. CSIP provides a software infrastructure for the development and deployment of modeling and data services [David2014].

CSIP is based on RESTful web services. REST stands for Representational State Transfer. It is an architectural approach enabling software systems to be built in where clients send requests to service end points [Fielding2002]. REST is a widely used method to implement client/server webservices. REST allows building software applications in which clients can make requests of services that are simple to use, easy to scale, and highly inter operable. REST requires an HTTP library to be available for most operations. REST relates resources to uniform resource identifier (URIs) and the uniform interface. Different URIs support accessing different resources similar to typing URLs in browser to access different website.

The CSIP REST-based Modeling and Data Services framework is an open source, production quality framework for developing RESTful Services in Java that is built on top of JAX-RS APIs (JSR 311 & JSR 339 Reference Implementation) as provided by J2EE 6 and the OMS3 modeling framework API [David2002]. CSIP provides it’s own API that extends the JAX-RS toolkit with additional features and utilities tailored for simulation models and data sources. CSIP also exposes various methods so that model developers may extend it to best suit their needs. Goals of the CSIP project can be summarized as follows:

- Easy implementation of REST-based modelling and data service back-ends for rapid development of modeling services.
- Standardized HTTP/JSON based protocol for client/server communication to support complex, large data structures as input and output by simulation models .
- Support for short/long running models with synchronous and asynchronous service execution with logging and archival of model input/output data to account for traceability and provenance.
- Ensemble execution of models and data services to speedup execution of models for parameter calibration and model optimization.
- High scalability and flexibility of CSIP deployments that can adjust to various infrastructure settings. The CSIP environment can be deployed on a single computer or a cluster of hundreds or thousands of machines.

This manual is organized as follows. Chapter 2 introduces the use of CSIP ModelServices from a client perspective. The JSON protocol for model and data service interaction is introduced and examples are provided. Different usage patterns for model service clients are explained, examples are given in different programming languages and environments. Chapter 3 discusses the development implementation of model and data services from a server perspective. The efficient implementation of services using native bundling of modeling resources, scalability aspects, and others is introduced. Chapter 4 introduces various approaches for implementing, configuring, and deploying a scalable CSIP architecture. Chapter 4 concludes with a detailed description of the server side API for CSIP.

Chapter 2. ModelService - Client RESTful API

The ModelServices REST web service resembles the schema of Web Processing Services (WPS) developed by the OpenGIS Consortium [OGC2007]. However, CSIP simplifies data definitions, data descriptions, and meta-data to allow easy use of the offered web services. The CSIP webservice interface defines three operations that can be requested by a client and performed by a CSIP implementation.

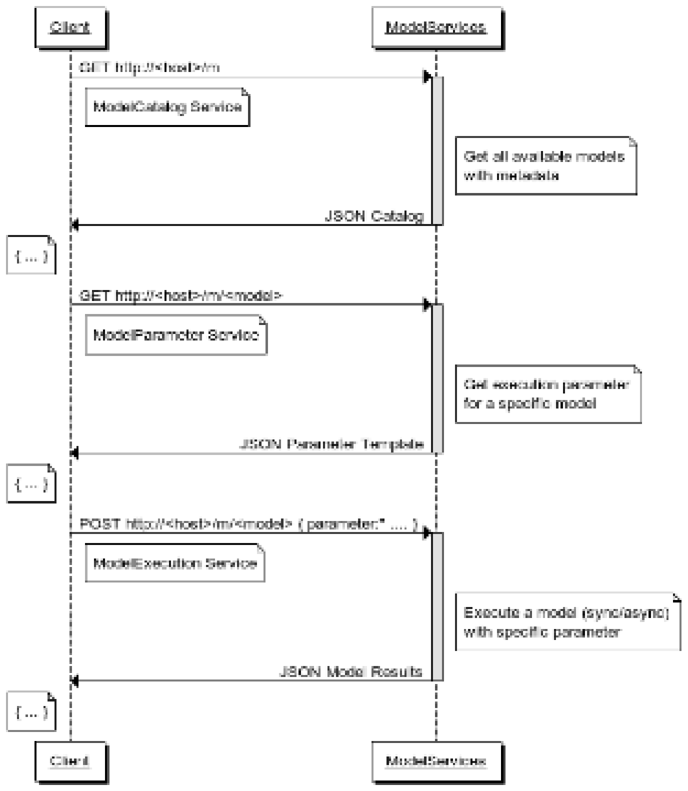
1. *Provide a list of available model and data services.* This operation allows a client to request and receive back service meta-data describing the abilities of the specific implementation. Metadata includes the names and general descriptions of each of the processes offered by a CSIP instance. This operation also supports negotiation of the specification version being used for client-server interactions.

2. *Describe a specific operation in detail.* This operation allows a client to request and receive back detailed metadata about specific model/data services including inputs required, their allowable formats, and the produced outputs.

3. *Execute the model or data service.* This operation allows a client to run a specified process implemented by CSIP, using provided input parameter values and returning the outputs produced. Execution meta-data is returned to provide delayed retrieval of model results, if the model executes over a long period.

This section specifies the structure of the ModelServices, the (i) ModelCatalog service, the (ii) ModelParameter service, and (iii) ModelExecution service. The Figure below shows the purpose of the service calls in sequence.

Figure 2.1. ModelServices Exploration and Invocation Sequence


Client
ModelServices
GET http://<host>/m
ModelCatalog Service
Get all available models with metadata
JSON Catalog
{ ... }
GET http://<host>/m/<model>
ModelParameter Service
Get execution parameter for a specific model
JSON Parameter Template
{ ... }
POST http://<host>/m/<model> { parameter:" .... }
ModelExecution Service
Execute a model (sync/async) with specific parameter
JSON Model Results
{ ... }
Client
ModelServices

Essential for application development is the knowledge of the ModelExecution service. A client application invokes ModelExecution services. The ModelCalalog and ModelParameter Service are used to verify service availability and input parameter requirements. They support exploration of services and service signatures "on-the-fly", the first two steps in the diagram (GET/GET). Given this information a client request can easily be crafted and submitted for execution.

2.1. ModelCatalog Service

The ModelCatalog service supports entry level exploration of available model services. Service descriptions are listed as an array of JSON objects with sufficient metadata to call an individual service.

JSON (JavaScript Object Notation) is a web focused data format. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange format.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, structure, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

JSON format is used by the ModelSevices architecture for data exchange. Model parameter, execution results, meta-information, are all passed as JSON Objects from the client to the web-service. JSON provides support for catalog listing of models, exploration of model capabilities, and execution control.

Example request and response sequences are shown below. The catalog of services is requested by calling a http GET operation against a base URL.

Example 2.1. ModelCatalog Response Format

```
{
  "service-version":
           <model services implementation version>,
  "services": [
    {
        "name": <model1 name>,
        "version": <model1 version>,
        "description": <model1 description>,
        "doc-ref": <model1 doc url>,
        "path": <model1 path>
    },
    {
        "name": <model2 name>,
        "version": <model2 version>,
        "description": <model2 description>,
        "doc-ref": <model2 doc url>,
        "path": <model2 path>
    },
    {
       ...
    }
  ]
}
```

Response elements:

`services-version` (optional)
: ModelServices specification version.

`services`
: Array of all model services available at this server. At least one service should be listed.

`name`
: The name of a model service, usually a single word.

`version` (optional)
: The version of this model service.

`description` (optional)
: A brief, one-line description of the model service.

`doc-ref` (optional)
: A reference to further documentation of the web service

`path`
: The path for ModelParameter/ModelExecution Services. The path can be relative to the base URL or a full URL

An example request for a model catalog and its response is shown in the examples below.

Example 2.2. Example ModelCatalog Request

```
GET /csip-erosion HTTP/1.1
Host: localhost:8080
Accept: application/json
```

Example 2.3. Example ModelCatalog Response

```
"services" : [
  {
    "name": "Rusle2",
    "version": "1.1",
    "description": "Revised Universal Soil Loss Equation Rill and Sheet Erosion",
    "path": "http://localhost/csip-erosion/m/rusle2/1.1"
  },
  {
    "name": "EFH2",
    "version": "1.1",
    "description": "Storm runoff model (Engineering Field Handbook.)",
    "path": "http://localhost/csip-erosion/m/efh2/1.1"
  }
  ...
]
```

The paths in this example are relative to the base URL used to obtain that catalog.

2.2. ModelParameter Service

The ModelParameter service provides detailed execution information for a specific ModelService. Input parameters are described including the name, default value, and physical unit. The generated ‘`parameter`’ JSON object can be used as a template for a client's ExecutionService call. Default parameter values are expected to produce valid ModelExecution results.

The ModelParameter Request is performed as an HTTP GET with the <path> and <version> obtained from the ModelCatalog result.

Example 2.4. ModelParameter Response Format

```
{
    "parameter": [
        {
            "name": "<parameter name>",
            "value": "<default value>",
            "unit": "<physical unit>",
            "min": "<minimum value>",
            "max": "<maximum value>",
            "description": "<parameter description>"
        },
        {
            "name": "<parameter name>",
            "value": "<default value>",
            "unit": "<physical unit>",
        },
        ..
    ],
    "metainfo": { }
}
```

Response elements:

`name`
: parameter name.

description (optional)
: brief parameter description.

value
: default value

unit
: the physical unit of the parameter value.

min (optional)
: the minimum value.

max (optional)
: the maximum value.

Example 2.5. Example ModelParameter Request

```
GET /csip-hydrotools/m/efh2/1.1 HTTP/1.1
Host: localhost:8080
Accept: application/json
```

Calling a ModelParameter request requires a URL that is being constructed from the ModelCatalog information. The host name of that URL is the base URL of the ModelCatalog call if the path catalog entry is relative. Note that the specific service version as provided in the catalog must be appended to the path, hence: `GET /csip/m/efh2/1.1`

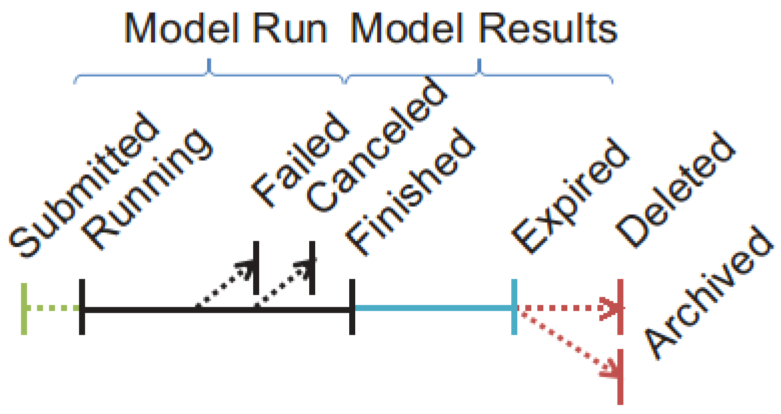
Example 2.6. Example ModelParameter response for EFH2

```
{
    "parameter": [
        {
            "name": "precip",
            "value": 14,
            "unit": "inch",
            "min": 0,
            "max": 100
        },
        {
            "name": "runoffcurvenumber",
            "value": 90,
            "min": 0,
            "max": 100
        },
        {
            "name": "stormtype",
            "value": "I"
        },
        {
            "name": "watershedlength",
            "value": 1500,
            "unit": "ft",
            "min": 0,
            "max": 100000000
        },
        {
            "name": "watershedslope",
            "value": 0.5,
            "unit": "%",
            "min": 0,
            "max": 100
        }
    ],
    "metainfo": {}
}
```

2.3. ModelExecution Service

A ModelExecution Service runs the simulation model. It expects model parameters as input in a JSON object and optional metadata controlling the model execution. This service call provides the model output as a result JSON object.

Figure 2.2. Execution phases



The Figure above shows execution phases common for all model services. There are two main groups with respect to the actual model run and the management of model results. Since a model service can be submitted in two modes "`sync`" and "`async`" there are two variants for managing those phases (Further details below). A client sends a ModelExecution request to initiate a model run. After initialization the service enters a "`Running`" state. Model execution can successfully complete ("`Finished`"), it can fail ("`Failed`"), or be canceled by user request ("`Cancelled`"). Additional phases relate to output data management. Model output data can be kept available for retrieval until it expires ("`Expired`"). The time to keep output data available can be controlled via the request metainfo entry "`keep_results`". In addition, there can also be a meta info entry to decide what will happen next, deleting the output data "`Delete`" or archive it for long term storage "`Archive`"

Each ModelExecution request payload contains two sections, as shown below.

Example 2.7. ModelExecution Request

```
{
  "metainfo" : { ❶
   },
  "parameter" : [ ❷
   ]
}
```

❶ metainfo (optional), provides metadata to the service to control execution, and returned by the service to describe execution status, statistics, and other information.

❷ parameter, provides input data for the model as an array of singe parameter JSON objects.

Each ModelExecution response payload contains three sections, as shown below.

Example 2.8. ModelExecution Response

```
{
  "metainfo" : {
   },
  "parameter" : [
   ],
  "result" : [     ❶
   ]
}
```

❶ result data after model execution as an array of JSON data objects

The metainfo and parameter entries are taken from the request, a basic requirement for RESTful service behavior. The metainfo section will be annotated with additional runtime information.

Example 2.9. ModelExecution request example

```
{
    "metainfo": {
          "mode":"sync",
          "keep_results: "200000"
    },
    "parameter": [
        {
            "name": "precip",
            "value": 14,
            "unit": "in"
        },
        {
            "name": "runoffcurvenumber",
            "value": 90
        },
        {
            "name": "stormtype",
            "value": "I"
        },
        {
            "name": "watershedlength",
            "value": 1500,
            "unit": "ft"
        },
        {
            "name": "watershedslope",
            "value": 0.5,
            "unit": "%"
        }
    ]
}
```

The example above shows a request JSON for calling the model service (EFH2).

Example 2.10. ModelExecution invocation

```
POST /csip-hydrotools/m/efh2/1.1 HTTP/1.1
Host: locahost:8080
Accept: application/json
```

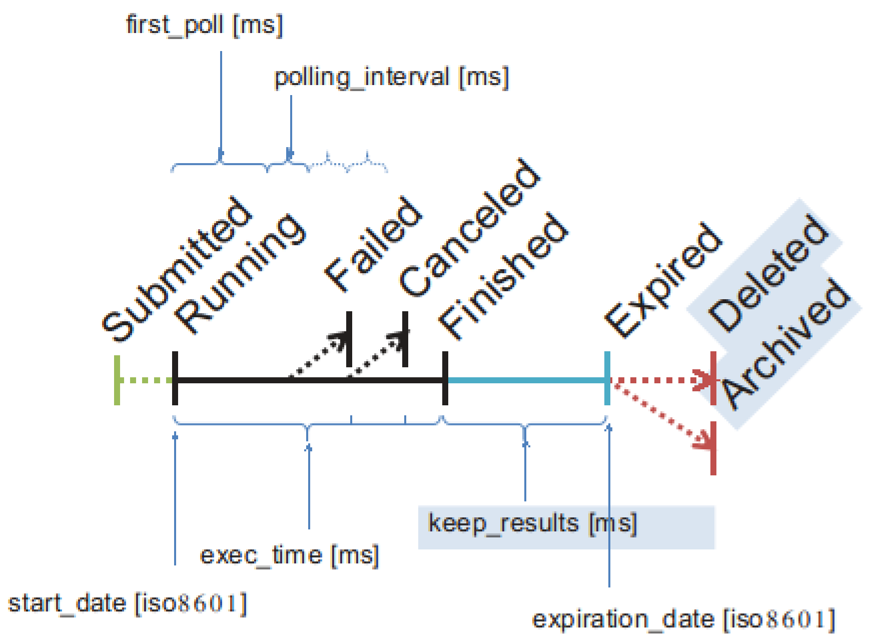
Example 2.11. ModelExecution response example

```
{
  "metainfo": {}
  "parameter": [
        {
            "name": "precip",
            "value": 14,
            "unit": "inch"
        },
        {
            "name": "runoffcurvenumber",
            "value": 90
        },
        {
            "name": "stormtype",
            "value": "I"
        },
        {
            "name": "watershedlength",
            "value": 1500,
            "unit": "ft"
        },
        {
            "name": "watershedslope",
            "value": 0.5,
            "unit": "%"
        }
    ],

}
```

2.3.1. Service Control Variables (Metainfo)

The metainfo JSON object contains variables that control model execution or provide feedback from the execution.

Figure 2.3. Variables



The result data (output JSON and output files) will be kept for a period of time after the model run successfully finished. If the model run failed or was cancelled no output will be stored. The time to keep results can be specified in the request using the `"keep-results"` entry within the `"metadata"` element.

In asynchronous execution mode, the returned `metadata` property `"polling_interval"` provides guidance for when the client should poll again for a status update. This value is model specific. Expect higher values for long running models and smaller values for models that run only for a short period. A client should **not** poll at intervals that are smaller than this value. Example: If a model takes on average 1 minute to finish, there is no need for the client to poll every 100 ms for a status update. The `"polling_interval"` in such case may be 5000 ms (5 seconds). There is no advantage for a client to use a shorter polling interval. The polling interval is provided by CSIP based on the average model runtime.

Table 2.1. Metainfo Elements

Key	Description	Default	Provided By	Example
`mode`	Execution mode of the service, `"sync"` or `"async"`	`sync`	request	`"mode":"async"`
`keep_results`	Number of milliseconds to keep results **after** model finishes execution	300000 ms (5 minutes)	request	`"keep_results": 600000`
`timezone`	timezone	UTC	request	`"timezone": "America/Denver"`
`status`	Status of the model execution: `"Sub-`	-	response	`"status": "Finished"`

Key	Description	Default	Provided By	Example
	mitted" | "Finished" | "Canceled" | "Failed" | "Running" | "Unknown"			
cpu_time	Time in ms for service execution	-	response	"cpu_time": "12324"
start_date	Time stamp for service execution	-	response	"start_date": "2012-03-21T15:23:42-0700"
expiration_date	Date when the output date will expire.	-	response	"expiration_date": "2012-03-21T15:29:42-0700"
next_poll	Recommended number of milliseconds between two client polls.	-	response	"next_poll":2000
first_poll	Recommended time in ms to wait before the first poll	-	response	"first_poll":2000
suid	Simulation unique identifier	-	response	"suid": "8553567a-ee1f-4270-81fb-ec9e9a5676a6"

Notes:

- All Dates are provided in ISO-8601 date format with timezone field (e.g. '2012-03-21T15:23:42-0700'). If a request provides timezone information all dates will be mapped to this timezone. If not, dates are UTC.
- Time parameters are always described using milliseconds, for both, request and response metainfo values.
- Suid's are Version 1 UUID's (see RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace). Thus, the suid creation time can be retrieved using the suid. Suid stands for "simulation unique identifier".

2.3.2. Service Execution

A synchronous CSIP request returns when the service run finishes, thus blocking the calling client during service execution. CSIP services default to synchronous execution. Conversely asynchronous CSIP requests initiate a model/data service and return immediately to the client. Subsequent service calls must be made to query the service status to determine if execution is complete. Model requests service can be processed both ways, synchronously and asynchronously. The metainfo key "mode" controls the execution mode when submitting the initial request.

The example below demonstrates asynchronous service execution.

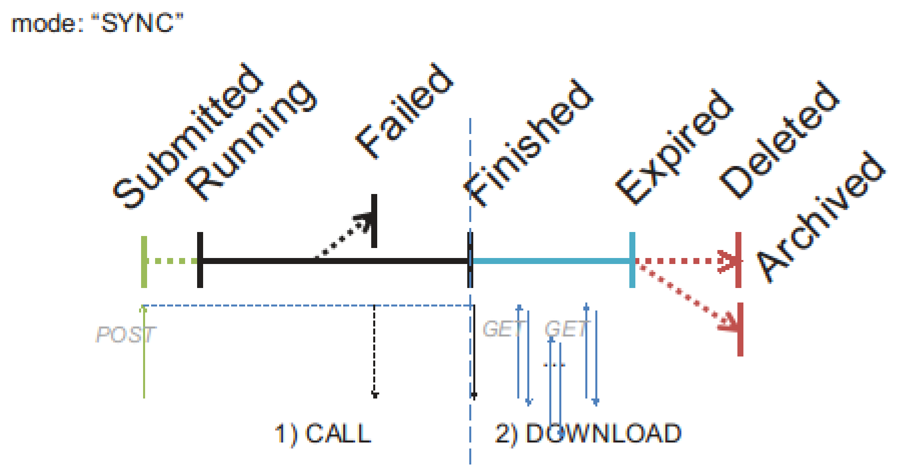
Example 2.12. ModelExecution response example

```
{
    "metainfo": {
            "mode": "async",
             ...
     }
   …
}
```

2.3.2.1. Synchronous Execution

The figure below shows the client calling sequence for synchronous execution. The server responds with a response JSON object which includes the client request input parameterization regardless if the model finished successfully or not. During execution the client thread is blocked. The client must account for issues like "call threading" to support a responsive user interface.

Figure 2.4. Synchronous Service Phases

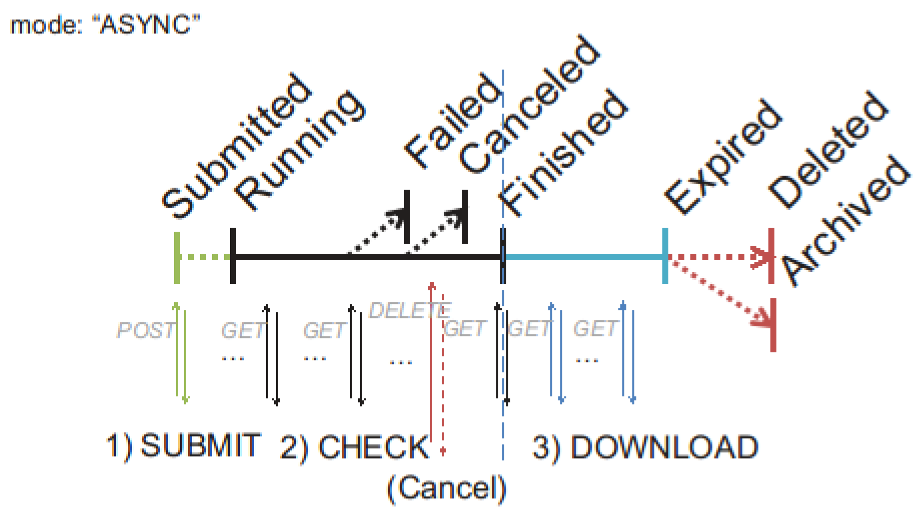


After service execution completes the output data and optional report output data can be fetched at anytime before expiration from the session backend datastore.

2.3.2.2. Asynchronous Execution

While synchronous execution may be acceptable for short running CSIP services (execution time < 10 sec) it is unpractical for long running requests. Such services may execute for minutes, hours, or even days. The figure below shows the principal flow of calls for such asynchronous execution. If mode is set to "async", the initial service call (POST) submits the JSON request for execution and returns immediately.

Figure 2.5. Asynchronous Service Phases



The response metainfo element contains the `suid`, a unique key, that uniquely identifies the service request. The service status indicates its state of execution. In addition, the metainfo contains "`first_poll`" and "`next_poll`" properties. The client should wait "`first_poll`" milliseconds before first querying for a service result.

Example 2.13. Result for submitted model execution

```
{
  "metainfo": {
    "status": "Submitted",
    "suid": "182b8505-7534-11e1-b93f-1d1c56c85325",
    "tstamp": "2012-03-23T22:04:00+0000",
    "next_poll": "2000",
    "first_poll": "25000",
  },
  "parameter": [
..
  ]
}
```

A REST query is being executed using the `suid`. The call shown below is constructed using the '`/csip/q`' path and the appended `suid`.

Example 2.14. Async Query

```
GET /csip/q/182b8505-7534-11e1-b93f-1d1c56c85325 HTTP/1.1
Host: csip.engr.colostate.edu:8083
Accept: application/json
```

The query result shows that the model is still running, and has not finished. Subsequent queries for the service status as in the example should be done after "`polling_interval`" milliseconds.

Example 2.15. Query Result

```
{
  "metainfo": {
    "status": "Running",
    "suid": "182b8505-7534-11e1-b93f-1d1c56c85325",
    "tstamp": "2012-03-23T22:04:00+0000",
    "polling_interval": "2000",
    "first_poll": "25000",
  },
  "parameter": [
..
  ]
}
```

Upon completion the service status indicates `Finished` and the service output is provided in the `result` JSON section as fetched from CSIP. The `suid` must be retained by the client during service execution as it serves as a token to query the service status and its results.

2.3.3. Attaching Request Input Data

CSIP services may require input files to execute. Not all service parameters can and should be mapped into JSON format. A service wrapping a legacy scientific model might consume input files directly with no modifications. In some cases, service input requirements may be so extensive their representation in JSON is impractical to implement.

CSIP services can easily consume client requests with files attached. Files are attached to CSIP HTTP/POST requests as multipart/form-data. See Example 2.16.

Example 2.16. Input with multipart/form-data

```
POST /m/swat/1.0 HTTP/1.0
Host: localhost
Content-type: multipart/form-data, boundary=AaB03x
Content-Length: $requestlen

--AaB03x
content-disposition: form-data; name="param"; filename="req.json"  ❶
Content-Type: application/octet-stream

$param
--AaB03x
content-disposition: form-data; name="file1"; filename="hru.dat"   ❷
Content-Type: application/octed-stream
$file1
--AaB03x
content-disposition: form-data; name="file2"; filename="input.std" ❸
Content-Type: application/octed-stream

$binarydata
--AaB03x--
```

❶ The json request file that needs to be attached as "`param`". The name "`param`" is required.
❷ The first file attachment, "`hru.dat`". The value of the name parameter is ignored.
❸ The second file attachment, "`input.std`". The value of the name parameter is ignored.

Input file naming is flexible, except for the JSON parameter file. While the JSON file name can vary, the multi-part request parameter must be "`param`". The example below shows a request for Example 2.16 using the command line command CURL.

```
$ curl -X POST -H "Accept:application/json" \
        -F param=@/tmp/req.json \
        -F file1=@/tmp/hru.dat \
        -F file2=@/tmp/input.std \
        ...
          "http://localhost:8080/csip-swat/m/swat/1.0"
```

More examples on how to attach files to the HTTP POST request using different clients and languages are shown in Section 2.4.

2.3.4. Download Result Data

Legacy scientific models typically produce data output files. CSIP streamlines providing model output as `key value pairs in the results JSON array` to the client. Such output can be obtained from the model directly, or parsed from model output files. In some cases it is beneficial for the client to access the native model output files. For this use case, CSIP provides a query service to obtain the files. An example is shown below.

A model finishes and provides in `the result` JSON array a list of file pointers. A client can fetch the files as referenced by the provided URLs. The file size is provided in bytes.

Example 2.17. Model response with data download

```
{
  "metainfo": {
    "status": "Finished",
    "suid": "182b8505-7534-11e1-b93f-1d1c56c85325",
    "tstamp": "2012-03-23T22:04:00+0000",
    "polling_interval": "2000",
    "first_poll": "25000",
    "cputime": "28155",
    "expiration_date": "2012-03-23T22:09:28+0000"
  },
  "parameter": [
    {
      "name": "wepsrun",
      "value": "weps.run"
    },
    {
      "name": "climate",
      "value": "cli_gen.cli"
    },
    {
      "name": "wind",
      "value": "win_gen.win"
    },
    {
      "name": "management",
      "value": "S Wheat-Beet-SB CMZ 1.man"
    },
    {
      "name": "soils",
      "value": "Heldt_48_90_CL.ifc"
    }
  ],
  "result": [
    {
      "name": "sci",
      "value": "http://localhost:8080/csip-erosion/q/182b8505-7534-11e1-b93f-1d1c56c85325/sci_energy.out"
      "size": "21729"
    },
    {
      "name": "stir",
      "value": "http://localhost:8080/csip-erosion/q/182b8505-7534-11e1-b93f-1d1c56c85325/stir_energy.out"
      "size": "91638"
    },
  ]
```

Clients must fetch temporarily stored server-side data before it expires. The expiration date is specified `in the metainfo`. In addition, the `suid` is used to identify the files of specific model runs, `suid 182b8505-7534-11e1-b93f-1d1c56c85325` in the example above.

A client parses the result JSON to obtain the URLs for downloadable files. It can then issue an HTTP GET request to fetch the files.

Example 2.18. GET Request for result file download

```
GET /csip-erosion/q/182b8505-7534-11e1-b93f-1d1c56c85325/stir_energy.out HTTP/1.1
Host: localhost:8080
Accept: */*
```

Example 2.18 shows the GET request to download the stir_energy.out file.

2.3.5. Error Messages

When client data or model service requests fail the service provides an error message. This message is embedded into the `metainfo` section of the JSON response.

The figure below shows a response containing an error message. The presence of an error key in `metainfo` indicates an execution problem.

Example 2.19. Error Handling

```
{
    "metainfo": {
        "status":"Failed",
        "error": "Insufficient input data"
    }
    "parameter":[
      ...
    ]
}
```

The JSON response containing the error contains the original service parameterization and request metainfo.

2.3.6. Ensemble Execution

The method of submitting multiple sets of parameters for model execution is called an ensemble run. Each individual parameter set represents an individual request. CSIP splits up parameter sets and executes a single run for each parameter set. Service requests will execute in parallel using the configured number of worker threads.

An ensemble run completes all individual requests runs finish. The `result` JSON contains the results of the entire ensemble request. Like the array of parameters, results are returned as an array of result sets in request order. The first result set relates to the first parameter set.

Example 2.20. Ensemble Execution

```
{
   "metainfo": {
       "parametersets":25
   }
   "parameter":[
    }
     "name:"p1",
     "value":"23",
      ...
    {,
    {
     "name":"p1",
     "value:"24",
      ...
    }
    ...
   ]
}
```

Ensemble runs like single runs can be executed in asynchronous and synchronous mode.

2.3.7. Service Testing

CSIP services can be automatically tested. Unit tests can easily be set up to test individual services or collections. Tests may run at the command line or within unit test frameworks such as JUnit [1] or TestNG[2]. CSIP provides a separate library for testing (`csip-test.jar`). It can be directly executed or added to the JUnit test environment.

Service test setup follows simple naming conventions and is fully descriptive. Each service end point may be tested with one or more JSON requests files. Each JSON request may have file attachments as input for the service. The CSIP test environment automatically submits the JSON to the service endpoint and attaches all input. After successful service invocation the response is stored at the client. If the service produces result files they will be downloaded and stored on the client side. "Golden" response files can be used to automatically compare the retrieved response against known valid values.

All tests for one service endpoint reside in one folder. The folder contains the test descriptor, the JSON request(s), optional request input(s), and optional golden files.

The following directory structure is used for CSIP tests:

<folder>/
: The test folder

  <service-name1>/
  : The folder for testing "service-name1".

    `service.properties`
    : The test descriptor, contains properties that control the test. (required)

    <test1>`-req.json`
    : The test1 request for the service. (required)

    <test1>`-res.json`
    : The service response for the test1 request (output)

    <test1>`-req/`
    : The directory containing all input files to be attached to the service request. (optional)

    <test1>`-res/`
    : The directory containing all output files from the service response. (output)

    <test1>`-ref.json`
    : The golden service output reference file. (optional)

    <test2>`-req.json`
    : The second test ...

    ...
    : (The same set op files for test 2)

  <service-name2>/
  : The folder for testing "service-name2"

    `service.properties`
    : Test descriptor "for service-name2"

[1]http://www.junit.org
[2]http://www.testng.org

...
: The same set of request, response, input, output, and golden files.

Service test description follows a simple naming convention. A service test must have the "`service.properties`" file. Each service test consists of a base name and a set of suffixes for different aspects of the test (`*-req.json`, `*-res.json`, etc.). The CSIP test framework will recognize the file's bases and generate the service outputs accordingly.

The file `service.properties` contains property settings controlling service test execution. The url entry is the only required entry which specifies the service endpoint to test.

Example 2.21. service.properties

```
# service endpoint, required
url=http://localhost:8080/csip-rzwqm/m/rzwqm/1.0

# number of concurrent tests, optional, defaults to '1'
concurrency=4

# timeout for a single test, optional, defaults to '3000'
timeout=5000

# test method: "keysonly" | "keyvalue" | "keyvalueorder" | "none", ❶
# defaults to 'none'
verify=keysonly

# should this service test be ignored? ,optional, defaults to 'false'
ignore=false

# should result files be downloaded?, optional, defaults to 'true'
download_files=false
```

❶ If the verify property is set to "`keysonly`", the presence of keys is checked against the golden reference file. The "`keyvalue`" property triggers a key/value comparison, the "`keyvalueorder`" considers even the order of the key/value pairs. If verify is set to none, the service is being executed, no verification of the results is performed.

Example: Service Test

A service for temperature conversion might be tested using the following setup. A test folder was created using the structure as seen in Example 2.22. There are 2 tests defined within "`tempservice_V1_0`". Both are using a reference file.

Example 2.22. Service Test Example

```
work
+--csip
   +-- tests
       +-- tempservice_V1_0
       |    service.properties
       |    tempconv1-req.json
       |    tempconv1-ref.json
       |    tempconv2-req.json
       |    tempconv2-ref.json
       |
       +-- otherservice_V2_0/
            service.properties
            ...
```

The file `service.properties` in tempservice_V1_0 contains the test properties as shown in Example 2.23. The key/value pairs of the response results should be compared against the reference file.

Example 2.23. Test settings for 'tempconv' in service.properties

```
# service endpoint
url=http://localhost:8080/csip-example/m/tempconv/1.0
# verify the results
verify=keyvalue
```

The test can be executed with the command below. The command line argument '`/work/csip/tests/tempservice_V1_0`' is the path to the folder of the `tempservice` tests.

```
> java -jar csip-test.jar /work/csip/tests/tempservice_V1_0
=================
 Total:     2
 Failed:    0
 Succeeded: 2
 Skipped:   0
 Ignored:   0

>_
```

As a result, the two tests run successfully against the '`tempconv`' endpoint. Upon completion, the output summary of the test command is provided.

Integration into existing unit testing frameworks is easy. The csip-test.jar file has to be added to the `CLASSPATH` used for testing. The Example 2.24 shows the use of service test example within a JUnit test. Similar to the command line interface, a service test is executed with the path to the test directory as an argument.

Example 2.24. JUnit based Service Testing

```
import csip.test.ServiceTest;
import junit.framework.Assert;
import org.junit.Test;

public class STest {
  @Test
  public void stest() throws Exception {
    ServiceTest.Results r = ServiceTest.run(
          "/work/csip/tests/tempservice_V1_0");
    Assert.assertTrue(r.getTotal() == r.getSucceeded());
  }
}
```

JUnit Assert methods can be used to validate the test result.

2.4. Invocation Examples

The ModelServices API is callable from various clients such as CURL, Java, or the RestClient browser plugin. CURL is a command line tool for HTTP requests. Java can be used with several libraries to perform Restful HTTP service calls. RestClient exists as a UI plugin for several web browsers and can be obtained online at: https://github.com/rest-client/rest-client.

This chapter provides several examples of how to implement a CSIP client for a variety of programming languages. Parameter and result elements may change during development because of requirement adjustments and service improvements.

All ModelService URLs provided by CSIP have a common structure. The service context path starts with '`csip`', followed with an '`m`' indicating the modeling services category. Other service categories in CSIP include: database services ('d'), the query service 'q' described earlier, and console services 'c'.

Example 2.25. CSIP Service URL

```
http://<host>:<port>/csip-<context>/{m|d}/<service>/<version>"
```

Description:

<host>
  The service host name.

<port>
  The service port, if not specified the port is 80.

<context>
  The context of the CSIP service.

<service>
  The name of the model service. It usually corresponds to a known simulation model.

<version>
  The version of the model service. This can be an arbitrary string, it does not have to be a numerical value.

An example CSIP service URL is shown below for the temperature conversion service.

Example 2.26. CSIP URL example for the temperature conversion service.

```
http://localhost:8080/csip-example/m/temp/1.0
```

This URL represents the version 1.0 of the temperature conversion service at port `8080` on `localhost`.

2.4.1. CURL

For simple service usage, CURL is a useful tool to submit an `http` request and receive a response. CURL is a simple command line tool for Linux, Windows, OSX, and other operating systems. A typical usage of CURL for CSIP is shown below.

The example fetches the model parameterization requirements for the temperature conversion as a JSON object using an HTTP GET (default curl operation)

```
curl -H "Accept: application/json"
    "http://localhost:8080/csip-example/m/temp/1.0"
```

The following example executes the temperature service by sending an HTTP POST request by submitting the request data.

The file `temp.json` contains the service request data, the parameter data "temp" with a value of 25.

```
{
   metainfo: {},
   parameter: [
     { name: "temp",
        value: 25
      }
    ]
}
```

The command below performs a POST request against the CSIP endpoint.

```
curl -X POST \
     -H "content-type: application/json" \
     -H "accept: application/json" \
     "http://localhost:8080/csip-example/m/temp/1.0" -d @temp.json
```

The "-X" flag selects the POST request method to use when communicating with the CSIP server. The header flag "-H" specifies the content type as "`application/json`" and the JSON request data is obtained from the file "`temp.json`" (using the "-d" flag).

2.4.2. Java

There are different methods for invoking CSIP services from Java. In the examples below we present four approaches to write CSIP REST clients, including an example using the CSIP library itself.

2.4.2.1. Java HTTP client library

Like almost all programming languages, Java has built-in support for HTTP allowing basic POST and GET requests. This API is built around two classes, java.net.URL and java.net.HttpURLConnection. The URL class is just a Java representation of a URL.

From a URL, one can create an HttpURLConnection that allows to invoke specific requests. Here is an example of doing a CSIP POST request:

Example 2.27. CSIP service execution in core Java

```
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.BufferedReader;

...
URL url = new URL("http://localhost:8080/csip/m/temp/1.0");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
conn.setDoOutput(true);
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-Type", "application/json");
conn.setUseCaches(false);
conn.setDoInput(true);
conn.setDoOutput(true);
OutputStream os = conn.getOutputStream();
os.write(("{\n"
        + "    \"metainfo\": {\n"
        + "    },\n"
        + "    \"parameter\": [ \n"
        + "     {\n"
        + "      \"name\": \"temp\",\n"
        + "      \"value\": 25\n"
        + "     }\n"
        + "    ]\n"
        + "}").getBytes());
os.flush();
os.close();

if (conn.getResponseCode() != HttpURLConnection.HTTP_OK) {
    throw new RuntimeException("Failed.");
}

InputStream is = conn.getInputStream();
BufferedReader rd = new BufferedReader(new InputStreamReader(is));
StringBuilder response = new StringBuilder();
String line;
while ((line = rd.readLine()) != null) {
     response.append(line);
     response.append('\n');
}
connection.disconnect();

System.out.println("Response: " + response.toString());
```

By calling `HttpURLConnection.setDoOutput(true)` a body for the request can be written. We then call `setRequestMethod()` to tell the connection we're making a POST request. The `setRequestProperty()` method is called to set the Content-Type of our request. We then get a `java.io.OutputStream` to write out the data to the CSIP endpoint. The output of the request is then obtained using the `getInputStream()` method and output o of the call is stored in the response String. Finally, `disconnect()` is called to clean up the HTTP connection.

2.4.2.2. Java RX RS Client Library

The following client exercises the Java API for RESTful Web Services (JAX-RS). JAX-RS Client API is another Java based API for communication with RESTful Web services[3]. It is also part of Java EE 7 and it is designed to make

[3]https://jersey.java.net

it very easy to consume a Web service exposed via HTTP protocol to enable developers to concisely and efficiently implement portable client-side solutions.

The `ClientBuilder` is a JAX-RS API used to create new instances of Client. In a slightly more advanced scenarios, `ClientBuilder` can be used to configure additional client instance properties, such as a SSL transport settings.

Example 2.28. CSIP service execution using Java RX RS Client

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;

...
String req = new String("{\n"
      + "    \"metainfo\": {\n"
      + "    },\n"
      + "    \"parameter\": [ \n"
      + "     {\n"
      + "      \"name\": \"temp\",\n"
      + "      \"value\": 25\n"
      + "     }\n"
      + "    ]\n"
      + "}");

String url = "http://localhost:8080/csip/m/temp/1.0";

String resp = ClientBuilder.newClient()
         .target(UriBuilder.fromUri(url).build())
         .service.request(MediaType.APPLICATION_JSON)
         .post(Entity.entity(req, MediaType.APPLICATION_JSON_TYPE), String.class);

System.out.println(resp);
```

Once there is a Client instance created, a `WebTarget` object can be obtained from it. A `Client` contains several `target(...)` methods that allow for creation of `WebTarget` instance. In this case we're using `target(String uri)` version. The uri passed to the method as a String is the URI of the targeted web resource.

This compact example demonstrates another advantage of the JAX-RS client API. The fluency of JAX-RS Client API is convenient especially with simple use cases like CSIP.

2.4.2.3. Apache HTTPClient library

The following client harnesses the Apache HTTPClient library[4]. The Apache foundation is providing an extensible HTTP client library called HttpClient. Although it is not JAX-RS-aware, it does have facilities for preemptive authentication and APIs for dealing with a few different media types like forms and multipart which is essential for calling a CSIP service. Some of its other features are a full interceptor model, automatic cookie handling between requests, or pluggable authentication.

[4]http://hc.apache.org

Example 2.29. CSIP service execution in Java

```
import java.io.BufferedReader;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.ContentType;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.HttpClientBuilder;

...
StringEntity e = new StringEntity(
              "{\n"
            + "    \"metainfo\": {\n"
            + "    },\n"
            + "    \"parameter\": [ \n"
            + "     {\n"
            + "      \"name\": \"temp\",\n"
            + "      \"value\": 25\n"
            + "     }\n"
            + "    ]\n"
            + "}", ContentType.APPLICATION_JSON);

String url = "http://localhost:8080/csip/m/temp/1.0";

// create the client and post the request:
HttpPost post = new HttpPost(url);
post.setEntity(e);
HttpClient client = HttpClientBuilder.create().build();
HttpResponse response = client.execute(post);

BufferedReader rd = new BufferedReader(
       new InputStreamReader(response.getEntity().getContent()));

StringBuilder result = new StringBuilder();
String line;
while ((line = rd.readLine()) != null) {
    result.append(line);
    result.append('\n');
}
System.out.println("Response: " + result.toString());
```

In Apache HttpClient 4, the org.apache.http.client.DefaultHttpClient class is responsible for managing HTTP connections. It handles the default authentication settings, pools and manages persistent HTTP connections (`keepalive`), and any other default configuration settings. It is also responsible for executing requests.

There are related classes in the `org.apache.http.client.methods` package for performing GET, POST, PUT, and DELETE invocations. To push service input data to the CSIP server via a POST operation, one needs to encapsulate the data within an instance of the `org.apache.http.HttpEntity` interface. The framework has some simple prebuilt ones for sending strings, forms, byte arrays, and input streams. A `org.apache.http.entity.StringEntity` encapsulates the JSON that a client wants to send across the network. The correct Content-Type is set by calling `StringEntity.setContentType()`, the entity is passed to the request by calling `HttpPost.setEntity()`.

Once a request is built, it is executed by passing and calling `DefaultHttpClient.execute()`. This returns an `org.apache.http.HttpResponse` object. The `HttpResponse.getEntity()` method returns an `org.apache.http.HttpEntity` object, which represents the message body of the response. From it the client can get the Content-Type by executing `HttpEntity.getContentType()` as well as a `java.io.InputStream` to read the response. Finally, connections require cleanup by calling `HttpClient.getConnectionManager().shutdown()`.

2.4.2.4. CSIP HTTPClient

The CSIP distribution provides its own HTTP client which is internally based on the Apache `HTTPClient` library. However, the CSIP client offers a more concise API to invoke CSIP service endpoints with only a single line of code.

The CSIP client library is entitled `csip-client.jar` within the distribution. Add this to the `CLASSPATH` of any Java CSIP client application. The example CSIP call is shown below:

Example 2.30. CSIP service execution in Java (CSIP client)

```
import org.codehaus.jettison.json.JSONObject;
import csip.Client;
...

String req = "{\n"
           + "    \"metainfo\": {\n"
           + "    },\n"
           + "    \"parameter\": [ \n"
           + "     {\n"
           + "      \"name\": \"temp\",\n"
           + "      \"value\": 25\n"
           + "     }\n"
           + "    ]\n"
           + "}";
String url = "http://localhost:8080/csip/m/temp/1.0";
JSONObject res = new Client().doPOST(url, new JSONObject(req));

System.out.println(res.toString(4));
```

The CSIP service gets executed via the `doPOST()` method. A variant of this method allows also the attachment of files to the HTTP/POST CSIP request, ensuring proper setting of headers and mime types.

Using the CSIP client jar is the recommended method for calling CSIP services in Java.

2.4.3. C#

The Microsoft .NET Framework provides a layered, extensible, and managed implementation of Internet services that can be quickly and easily integrated into your applications. Both examples, C# and VB.NET use the same .NET Framework classes for performing a CSIP service call.

The following example describes the steps used to execute send input data to a CSIP service and fetch its response.

The .NET Framework provides protocol-specific classes derived from `WebRequest` and `WebResponse` for "http:" URIs. Both classes are the core if this example.

The following example shows the CSIP client in C# for the .NET platform.

Example 2.31. CSIP service call in C#

```
using System;
using System.Net;
using System.IO;
using System.Text;

namespace httpclient
{
 class MainClass
 {
  public static void Main(string[] args)
  {
   string json = "{ " +
    "  metainfo: {}," +
    "  parameter: [ " +
    "    { name: \"temp\"," +
    "      value: 25 " +
    "    } " +
    "  ] " +
    "}";

   // Create a request using a URL that can receive a post.
   WebRequest request = WebRequest.Create(           ❶
              "http://localhost:8080/csip/m/temp/1.0");
   request.Method = "POST";                          ❷

   // Create POST data and convert it to a byte array.
   byte[] byteArray = Encoding.UTF8.GetBytes(json);
   // Set the ContentType property of the WebRequest.
   request.ContentType = "application/json";
   request.ContentLength = byteArray.Length;         ❸
   // Get the request stream.
   Stream dataStream = request.GetRequestStream();
   dataStream.Write(byteArray, 0, byteArray.Length);  ❹
   dataStream.Close();

   // Get the response.
   WebResponse response = request.GetResponse();      ❺

   StreamReader reader = new StreamReader(response.GetResponseStream());
   string responseFromServer = reader.ReadToEnd();    ❻

   Console.WriteLine(((HttpWebResponse)response).StatusDescription);
   Console.WriteLine(responseFromServer);             ❼

   reader.Close();
   response.Close();
  }
 }
}
```

❶ Create a `WebRequest` instance by calling `Create()` with the URI of the CSIP service resource.

❶ Specify a protocol method that permits data to be sent with a request, such as the HTTP POST method.

❶ Set the content type and length

❶ Write the JSON data to the Stream object returned by the `GetRequestStream()` method.

❶ Send the request to the CSIP server by calling `GetResponse()`. This method returns an object containing the server's response.

❶ Read the JSON response into a string.

❶ Print out the response.

2.4.4. VB.Net

This example explain the use of VB.NET for posting a a CSIP model request to a server using the .NET framework classes `WebRequest` and `WebResponse` as is was shown in the previous section for C#. Because the .NET framework classes being used are the same, the examples only differ with respect to language constructs.

Example 2.32. CSIP service call in VB .Net

```
Imports System.Net
Imports System.IO

Public Class Application
 Public Shared Sub Main()

  Dim URL As String =
    "http://localhost:8080/csip/m/temp/1.0"

  Dim req As String = "{ " +
    "  metainfo: {}," +     "  parameter: [ " +
    "    { name: ""temp""," +
    "      value: 25 " +
    "    } " +
    "  ] " +
    "}"
  Dim res As String = ""

  Try
    Dim client As HttpWebRequest = Webrequest.Create(URL)
    client.Method =  "POST"
    client.ContentType = "application/json"
    Dim encoding As New Text.ASCIIEncoding()
    Dim postByteArray() As Byte = encoding.GetBytes(req)

    client.ContentLength = postByteArray.Length
    Dim postStream As Stream = client.GetRequestStream()
    postStream.Write(postByteArray, 0, postByteArray.Length)
    postStream.Close()

    Dim clentResponse As HttpWebResponse = client.GetResponse()

    If clentResponse.StatusCode = HttpStatusCode.OK Then
      Dim responseStream As StreamReader = _
        New StreamReader(clentResponse.GetResponseStream())
      res = responseStream.ReadToEnd()
    End If
    clentResponse.Close()

  Catch e As Exception
    res = "CSIP error occurred: " & e.Message
  End Try

  System.Console.WriteLine(res)
 End Sub
End Class
```

The VB.NET example employs exception handling for error management in tis example.

2.4.5. Python

Python is an object-oriented, interactive, general purpose programming language that has characteristics such as high level dynamic data types, dynamic typing and a very clear syntax [5]. The `httplib2` python library[6] is a comprehensive HTTP client library that handles caching, keep-alive, compression, redirects, and many kinds of authentication. The `httplib2.py` library supports many features left out of other Python HTTP libraries. This library is installed using the `pip` command.

```
#pip install httplib2
```

CSIP model execution using the Python/httplib2 is shown below. The http2 library and the json library must be imported.

Example 2.33. CSIP service call in Python

```
import httplib2, json

req = json.dumps(
 { 'metainfo': {},
   'parameter' : [
   {
    'name' : 'temp',
    'value': 25
   }]
 })

headers = {"content-type": "application/json",
           "accept": "application/json"}
url = "http://localhost:8080/csip/m/temp/1.0"

http = httplib2.Http()
res, content = http.request(url, 'POST',
                            headers=headers, body=req.encode())

print res.status, res.reason
print content
```

The `json.dumps()` method serializes the argument to a JSON formatted string. Note that we have to use the `encode()` function from json to encode the string before using it as the POST body.

2.4.6. C/C++

C is a general-purpose, imperative computer programming language, while C++ adds object-orientation to C. The most complete library for C/C++ programs is `libcurl`[7], a free and easy-to-use client-side URL transfer library. Curl builds and works identically on all operating systems. `libcurl` is the most used C-based highly portable transfer library in the world.

libcurl is often pre-installed on linux and unix based systems. However, package managers such as apt, yum, rpm, etc. can be used to fetch and install the binary packages as needed.

libcurl introduces the "easy" interface. All operations in the easy interface are prefixed with '`curl_easy`'. The easy interface provides for single transfers with a synchronous and blocking function call. The example CSIP client program below can be compiled using `gcc` and must be linked against the `libcurl` library by using the `-lcurl` flag.

[5]https://www.python.org
[6]https://github.com/jcgregorio/httplib2
[7]http://curl.haxx.se/libcurl

Example 2.34. CSIP service call in C/C++

```
// install libcurl
// compile: gcc Call.c -lcurl

#include <string.h>
#include <curl/curl.h>

int main() {
  CURL *curl;
  CURLcode res;
  struct curl_slist *headers = NULL;

  curl = curl_easy_init();
  if (curl) {
    const char *req =
        "{ "
        "  metainfo: {},"
        "  parameter: [ "
        "    { name: \"temp\","
        "      value: 25 "
        "    } "
        "  ] "
        "}";

    const char *url = "http://localhost:8080/csip/m/temp/1.0";

    headers = curl_slist_append(headers, "content-type:application/json");
    headers = curl_slist_append(headers, "accept:application/json");

    curl_easy_setopt(curl, CURLOPT_URL, url);
    curl_easy_setopt(curl, CURLOPT_POST, 1L);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, req);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, strlen(req));
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);

    res = curl_easy_perform(curl);
    if (CURLE_OK != res) {
      printf("Error: %s\n", strerror(res));
      return 1;
    }
    curl_easy_cleanup(curl);
    curl_slist_free_all(headers);
  }
  return 0;
}
```

The program initializes curl, creates header information, sets various options, and performs the POST request. Finally, all resources are released.

The code above will also compile with a C++ compiler (e.g. g++), which uses the same library.

2.4.7. JavaScript

JavaScript is an object-oriented programming language commonly used to create interactive effects within web browsers. The presented JavaScript CSIP example uses jQuery, a fast, small, and feature-rich JavaScript library[8]. JQuery simplifies HTML document traversal and manipulation, event handling, animation, and Ajax with an easy-to-use API that works across a multitude of browsers.

[8]http://jquery.org

Example 2.35. CSIP service call in JavaScript

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>
...

var reqData =
 '{' +
  '"metainfo": {},'+
  '"parameter": [' +
    '{' +
    '  "name": "temp",' +
    '  "value": 25' +
    '}' +
  ']' +
 '}';

// ajax call to the service
$.ajax({
  type: "POST",
  headers:{'accept': 'application/json'},
  contentType: "application/json",
  url: "http://localhost:8080/csip/m/temp/1.0",
  data: reqData,
  success: function (data, textStatus, xhr) {
      success(data);
  },
  error: function(jqXHR, textStatus, errorThrown){
      alert("Error! " + errorThrown);
  }
});
...
```

Simple AJAX calls in JavaScript can be performed to invoke CSIP services as shown in Example 2.35. This fragment constitutes population of a request object (`reqData`) and the client service invocation.

The success callback function is passed the returned data, which will be a JSON string depending on the MIME type of the response. It is also passed in the text status of the response.

Tomcat application server security defaults permit Javascript REST clients to only invoke services on the originating web server that provides the Javascript content to the user. Ajax requests are subject to the same origin policy; the request can not successfully retrieve data from a different domain, subdomain, port, or protocol. The CORS (Cross-origin resource sharing) setting in tomcat can be enabled to overcome this limitation.

2.4.8. Groovy

Groovy is a dynamic language with features similar to those of Python, Ruby, Perl, and Smalltalk. It can be used as a scripting language for the Java Platform, is dynamically compiled to Java Virtual Machine (JVM) bytecode, and interoperates with other Java code and libraries.

Groovy's `HTTPBuilder` [9]is the easiest way to manipulate HTTP-based resources in a Java VM. `HTTPBuilder` is a wrapper for Apache's `HttpClient` with some Groovy syntactical extensions. The request/response model is also inspired by `Prototype.js`.

`HTTPBuilder` is the main API class used to make requests and parse responses as shown below.

[9]http://groovy.codehaus.org/modules/http-builder

Example 2.36. CSIP service call in Groovy

```
@Grab(group='org.codehaus.groovy.modules.http-builder',
      module='http-builder', version='0.7' )

import groovyx.net.http.HTTPBuilder
import static groovyx.net.http.ContentType.JSON
import static groovyx.net.http.Method.POST

def http = new HTTPBuilder('http://localhost:8080')
http.request(POST) {
    uri.path = '/csip/m/temp/1.0'
    requestContentType = JSON
    body = [metainfo:[dummy:0], parameter:[[name: 'temp', value: 25]]]

    response.success = { resp, json ->
        println "Success! ${resp.status}"
        println "Response: ${json}"
    }

    response.failure = { resp ->
        println "Request failed with status ${resp.status}"
    }
}
```

The `@Grab` command manages installation of the http-builder library using the Grape tool. `HttpBuilder` is not part of the Groovy distribution. The request content type is set to JSON, while the JSON content is provided in Groovy syntax. The response handler is a closure which prints out the response body on successful execution.

2.4.9. Go

Go is an open source programming language developed by Google and many other contributors from the open source community [10]. It is a fast, statically typed, compiled language with support for garbage collection and run-time reflection.

The Go package `"GoRequest"` [11]provides a simplified HTTP client library for HTTP client implementations. In the example, this library is used for exercising the CSIP service call for temperature conversion. The library is installed using the `go get` command. The `gorequest` library simplifies the submission of JSON request data compared to standard Go libraries. JSON data can be provided directly.

[10]https://golang.org
[11]https://github.com/parnurzeal/gorequest

Example 2.37. CSIP service call in Go

```
package main

// install go get github.com/parnurzeal/gorequest
// go run Call.go

import (
  "fmt"
  "github.com/parnurzeal/gorequest"
)

func main() {
  request := gorequest.New()
  resp, body, errs := request.Post(
        "http://localhost:8080/csip/m/temp/1.0").
  Set("accept","application/json").
  Set("content-type","application/json").
  Send(`{
    "metainfo": {},
    "parameter": [
       {
         "name": "temp",
         "value": 25
       }
      ]
     }`).
   End()
   fmt.Println("response Status:", resp.Status)
   fmt.Println("result:", body)
   fmt.Println("errors:", errs)
}
```

The Go client source code shown above creates a CSIP POST request and sends the JSON parameter request to the `simpleservice` endpoint. The response status, body, and errors are printed out.

2.4.10. Ruby

Ruby is a dynamic, reflective, object-oriented, general-purpose programming language. Most Ruby distributions contain '`Net::HTTP`' [12]classes which are part of the Ruby Standard Library. `Net::HTTP` provides a rich library which can be used to build HTTP user-agents. `Net::HTTP` is designed to work closely with URI. `URI::HTTP#host`, `URI::HTTP#port` and `URI::HTTP#request_uri` are used within `Net::HTTP`.

[12]http://ruby-doc.org

Example 2.38. CSIP service call in Ruby

```
require 'net/http'
require 'json'

def do_csip
    uri  = URI('http://localhost:8080/csip/m/temp/1.0')
    http = Net::HTTP.new(uri. host)
    req  = Net::HTTP::Post.new(uri.path, initheader =
                   {'content-type' =>'application/json',
                    'accept' =>'application/json'})
    req.body = {
                metainfo: {},
                parameter: [
                    { name: "temp",
                      value: 25 }
                ]
               }.to_json
    res = http.request(req)
    puts "response #{res.body}"
rescue => e
    puts "failed #{e}"
end

do_csip
```

A POST request can be made using the `Net::HTTP::Post` class. The example above creates a JSON encoded request body containing the temperature conversion input. The function `do_csip` performs the request and prints out the response.

2.4.11. Client - Best Practices

It is recommended for the client to follow some rules when executing CSIP Services:

- For asynchronous requests, if service execution is expected to take a significant time (>2 sec), the returned `metadata` property "`next_poll`" and "`first_poll`" should be respected by the client. A client should start polling after "`first_poll`" milliseconds, and then in subsequent "`next_poll`" milliseconds. There is no advantage for a client to use a shorter polling interval.
- All timestamps are given in ISO-8601 format with timezone. In Java the pattern "`yyyy-MM-dd'T'HH:mm:ssZ`" can be used to parse such information into a '`java.util.Date`' object and further process it into `Calendar` or other representations.
- For all CSIP service requests the JSON payload may contain embedded JSON objects in addition to `metainfo` and `parameter`. The service will retain those entries and will send them back untouched in the response. This enables the client to use the service payload to carry additional (state) information with the service, such as user interface controls or database data.

Chapter 3. ModelServices - Server API

The CSIP server API allows the implementation of custom model services. There are various approaches implementing such services. The most generic method is by directly sub-classing the `ModelDataService` and implementing the required methods. This approach (see Section 3.1) provides maximum flexibility and can be used for all kind of service implementations.

If the primary purpose is to provide a service endpoint to host external native executables (e.g. wrapper), the `ModelDataService` class should be simply annotated. This is explained in detail in Section 3.1. As a second option, an OMS based model can be easily turned into a CSIP service by using the `ModelDataService` class with the annotation for OMS_DSL resources(Section 3.1).

3.1. Model and Data Service (ModelDataService)

The `ModelDataService` class is the base class for all CSIP modeling services. Service implementations usually subclass `ModelDataService` to offer a custom service endpoint. Concrete service implementations must subclass `ModelDataService`. Example 3.1 shows the generic service structure.

Example 3.1. General Service Structure

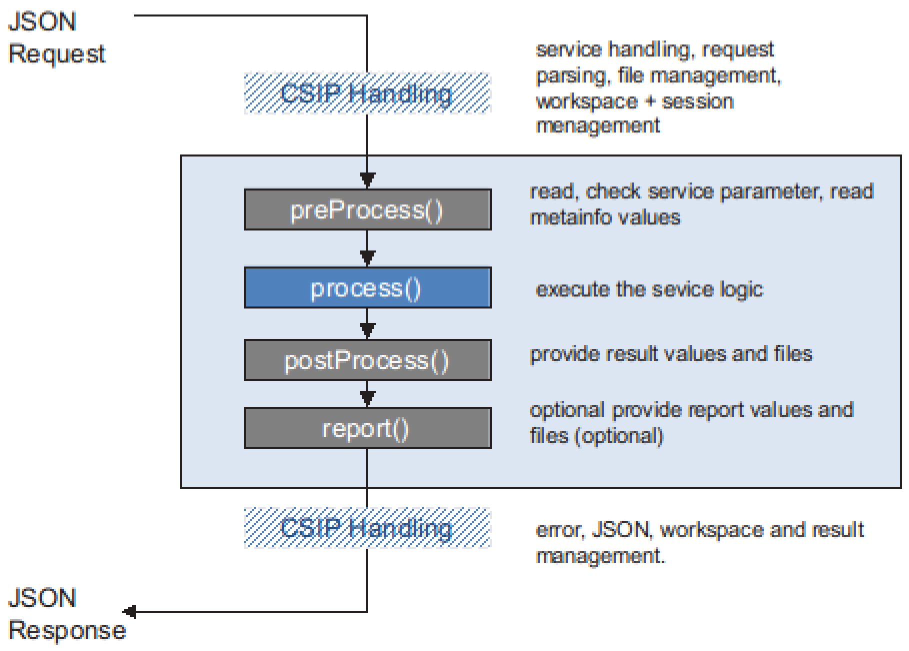
```
import javax.ws.rs.*;                                  ❶
import oms3.annotations.*;
import csip.*;

@Name("Energy")                                        ❷
@Description("Energy calculations")                    ❸
@Path("m/energy/1.0")                                  ❹
public class EnergyService extends ModelDataService {  ❺
   // service implementation
}
```

❶ Required package imports
❷ Name annotation, identifies the service by name in catalog listing. (optional)
❸ Brief service description. (optional)
❹ Service endpoint. (required)
❺ Subclassing `ModelDataService` for the `EnergyService`.

The listing above shows the core skeleton of a CSIP service. The new service class is named `EnergyService` which can be invoked using the specified endpoint `"<host>/m/energy/1.0"`. The class name can be seen as the internal representation of the service whereas the endpoint serves as the publicly accessible interface. The name and description of the service are optional, since they are only being used for catalog listing of available service endpoints within the same context.

A custom service may implement `ModelDataService` methods to facilitate preparation, execution, and results generation. These methods are invoked throughout the lifecycle of a service request. Figure 3.1 shows the sequence of method invocations.

Figure 3.1. Server-side Service Life-cycle


JSON Request
CSIP Handling
service handling, request parsing, file management, workspace + session menagement
preProcess()
read, check service parameter, read metainfo values
process()
execute the sevice logic
postProcess()
provide result values and files
report()
optional provide report values and files (optional)
CSIP Handling
error, JSON, workspace and result management.
JSON Response

JSON HTTP/POST requests are received by the server from the client. The CSIP framework will initially handles this content. It parses the payload, verifies its structural integrity, extracts attached input files from the payload, and creates a workspace on the server for this session. The workspace is initialized and identified with an SHA-1 hash. The `ModelDataService` method `getSUID()` can be used to obtain that UUID for the session. This hash is *guaranteed* to be unique across all active modeling service sessions and CSIP workers for a given CSIP deployment.

Four framework methods are provided to implement service specific handling of business logic as indicated with the blue box in Figure 3.1. At a bare minimum, the `process()` method must be implemented to realize the custom logic.

The processing methods are:

`void preProcess()`
: Implement to verify and obtain service request parameters (this is optional). Use the methods as described in Section 3.3 to fetch the request input and service metadata. All retrieved inputs should be stored as service instance data.

`String process()`
: This method is used to provide the core computational function of the service. Implementing this method is required. The computation may be executed within `process()` directly, or it can be performed as an external process. Further information about how to bundle native executable and other resources can be found in Section 3.5. Upon conclusion, this method must return a status indicator describing the outcome of execution (e.g. success, failure) Returning `null` indicates success, a `String` return signals an error, where the content of the `String` is the error message.

`void postProcess()`
: The postProcess() method prepares the output of the service (this is optional). Use the methods described in Section 3.4 to bundle result values, files, and other content to be returned.

`void report()`

This method can be implemented to provide optional reporting output for the service. Reports can detail additional output data that can optionally be fetched by the client on demand. An HTTP/GET client request is required to fetch report output. Implementation of the report() method is optional.

The final step of the service life-cycle wraps up the service response, manages the workspace and stores results and report files into the session datastore for the specified period of time before being archived and purged.

Example: A custom Service

An example of a simple service that implements the life-cycle methods is shown in Example 3.2.

Example 3.2. Service life-cycle example

```
@Name("EFH2")
@Description("Storm runoff model based on " +
             "conventions in Engineering Field Handbook.")
@Path("m/efh2/1.0")
public class V1_0 extends ModelDataService {

    EFH2HydrologyModel efh2 = new EFH2HydrologyModel(); ❶

    @Override
    protected void preProcess() throws Exception {
        efh2.setPrecip(getDoubleParam("precip"));       ❷
        efh2.setRunoffCurveNumber(getIntParam("runoffcurvenumber"));
        efh2.setStormType(getStringParam("stormtype"));
        efh2.setWatershedLength(getDoubleParam("watershedlength"));
        efh2.setWatershedSlope(getDoubleParam("watershedslope"));
    }

    @Override
    protected String process() throws Exception {
        return efh2.simulate() == 0 ? null ? "error";   ❸
    }

    @Override
    protected void postProcess() throws Exception {
        putResult("runoff", efh2.getRunoffQ());         ❹
        putResult("timeofconcentration", efh2.getTimeOfConcentration());
        putResult("unitpeakdischarge", efh2.getUnitPeakDischarge());
    }
}
```

❶ The model (`EFH2`) is instantiated as a service field.

❷ In `preProcess()`, all service inputs are fetched (e.g. `getDoubleParam()`) and passed to the model `EFH2`.

❸ in `process()`, the external model is executed and the result returned.

❹ in `postProcess()`, model results are obtained from EFH2 and prepared for the service response using the `putResult()` method.

Further discussion on methods for service request parameterization can be found in Section 3.3, output generation is explained in Section 3.4.

During service processing it is sometimes necessary to access the context of the service. `ModelDataService` offers various methods that can be used to access information such as the current ID of this session (SUID), find out more about the origin of the request (`getRequestURL(), getRequestHost(), getRequestContext(), getRemoteAddr()`), and to retrieve the current codebase of the service context Example 3.3.

Example 3.3. Service Context Methods

```
String getSUID()
String getRemoteAddr()
String getCodebase()
String getServicePath()
String getRequestURL()
String getRequestHost()
String getRequestContext()
```

All CSIP framework methods are further described in the appendix of this document.

3.2. Workspace Management

CSIP creates a workspace for each model session. This is a temporary directory residing within the CSIP work directory (configuration string "`csip.work`" Table 4.1). This temporary directory will be named with the simulation id (SUID). The SUID is a time and network address based universally unique identifier that uniquely identifies the modeling session and its associated resources.

The workspace is created and managed by CSIP and the methods listed in Example 3.4. These methods enable accessing the current workspace and the results directory. Both directories are named using the SUID.

Example 3.4. Workspace Management Methods

```
boolean hasWorkspaceDir()   ❶
File getWorkspaceDir()      ❷
```

❶ The workspace methods allow checking the existence of a workspace directory and obtaining a `File` reference.

❷ This method gets the workspace directory (provided by CSIP).

Some general notes about workspace and result directories and their management in CSIP:

1. A workspace is created by CSIP if: (1) there are files attached to the HTTP/POST call to invoke the service, or (2) the service calls `getWorkspaceDir()`. The `getWorkspace()` returns a valid, existing workspace folder, whereas the `hasWorkspace()` method just probes for its existence.
2. CSIP only provisions required resources for a service. If a service implementation includes no input or external executable files or generates no output, then the workspace folder is not created. This improves server side resource management.
3. The workspace is deleted after successful service execution. Only the files tagged using `putRestult(..)` (see Example 3.8) are preserved for download in the session datastore.

Example: Workspace usage:

If a service needs to generate an output file within the workspace it can be accomplished by accessing the `getWorkspace()` method. If the workspace does not exist, it will be created.

```
..
File dbg_out = new File(getWorkspaceDir(), "dbg.txt");
..
```

CSIP ensures that the workspace has proper file permissions for the workspace folder.

3.3. Handling Service Input

Each custom service must handle input. Service input must be provided in two forms:

1. JSON model services payload as the body of an HTTP/POST request according to the specification in Section 2.3, and
2. Input files as an HTTP/POST multipart attachment. One of the files must be the JSON model services payload.

A service might only have a JSON model service payload, if all the inputs can be provided in this form. In other cases, modeling services need multiple input files to run the model. A mixture of JSON input data and file attachments is common for model simulations.

3.3.1. JSON Input Data and Parameter

A CSIP client call provides input to the service using JSON content. The '`parameter`' section of the JSON file contains all literal input. The custom CSIP service, however, does not have to process, parse, and extract the data from JSON directly. This is accomplished from within CSIP infrastructure (ModelDataService). A service should always use the methods as shown in Example 3.5 to obtain parameter and metadata. ModelDataService methods handle the correct conversion of data types and validation of values as applicable.

Example 3.5. Parameter Input Methods

```
boolean hasParam(String name)                      ❶
int getParamCount()
Set<String> getParamNames()

String getStringParam(String name)                  ❷
String getStringParam(String name, String def)
int getIntParam(String name)
int getIntParam(String name, int def)
double getDoubleParam(String name)
double getDoubleParam(String name, double def)
boolean getBooleanParam(String name)
boolean getBooleanParam(String name, boolean def)
long getLongParam(String name)
long getLongParam(String name, long def)
JSONOjetct getJSONParam(String name)
JSONOjetct getJSONParam(String name, JSONOjetct def)

String getParamUnit(String name)                    ❸
String getParamDescr(String name)
JSONObject getParamGeometry(String name)
```

❶ Reports on the existence, number, and list of parameters.

❷ Returns the parameter value as a specific data type. If conversion is not possible, a `ServiceException` is thrown. All methods have variants that allow specification of default values if the parameter is not present.

❸ Get the parameter metadata. A custom service may want to fetch the parameter's unit description or the geometry. If a metadata element is not present, a `ServiceException` is thrown.

All parameter input should be obtained in the `preProcess()` or `process()` method of a service call. Usually, the request parameter data is placed into service fields or passed to the service's internal data structures.

A detailed listing of parameter methods can be found in ???.

Example: Accessing Model Parameter

A `ModelDataService` receives a JSON request that contains a parameter called "`precip`". This parameter has a value "`14.4`" and unit "`inches`".

```
{
    "parameter" : [
      ...
      {
         "name" : "precip",
         "value": 14.9
         "unit" : "inch"
      },
      ...
    ]
}
```

The above "`precip`" request parameter of a request can be read in by the service using the `getDoubleParam()` and `getParamUnit()` methods.

```
@Override
protected void preProcess() throws Exception {
    ...
    double precip = getDoubleParam("precip"));
    String precip_unit = getParamUnit("precip"));
    ...
}
```

If a JSON parameter cannot be converted into the desired type, a `ServiceException` is thrown.

3.3.2. Reading Metainfo

Service `metainfo` is a part of every service request. This is not direct input to the model, it rather provides context data to the service. As an example, a flag controlling the verbosity of service output, or requesting the use of a certain model backend would be part of the metainfo section and not parameter.

The Example 3.6 shows all methods available in CSIP to fetch metainfo content. It is not needed to parse the JSON request directly.

Example 3.6. Metainfo methods

```
Set<String> getMetainfoNames()          ❶
int getMetainfoCount()
boolean hasMetainfo(String name)

String getStringMetainfo(String name)   ❷
int getIntMetainfo(String name)
double getDoubleMetainfo(String name)
boolean getBooleanMetainfo(String name)
```

❶ Get available metainfo parameter names and quantity. Check if a metainfo entry is present.

❷ Get a metainfo value by name. If a metainfo element is not present, a `ServiceException` is thrown

Metainfo is typically processed in the `preProcess()` or the `process()` method of a service call to control service behavior. A detailed listing of all metainfo methods can be found in ???.

Example: Accessing MetaInfo

Within the metainfo section of a service request there can be any number of custom key/value entries. As an example, the metainfo section contains a verbosity flag, that controls the level of output for the service.

```
{
    "metainfo": {
        "verbose": true,
        ...
    }
    ...
}
```

Metainfo parameters are obtained from the JSON request object using the `get???MetaInfo()` methods as shown below.

```
if (hasMetainfo("verbose") {
    boolean verbose = getBooleanMetainfo("verbose")
}
```

If a metainfo element does not exist or has the wrong type, a `ServiceException` is thrown.

The use of custom metainfo content should be carefully considered. Metainfo is not input data to parameterize the service- rather it provides context and configuration data to direct HOW the service should run. Metainfo should be considered optional, as service defaults should enable the service to run when metainfo is absent.

Client provided metainfo is read-only in CSIP. The CSIP infrastructure appends useful metainfo to describe status information, execution time, compute node, etc. which is sent back to the client in the response JSON .

3.3.3. File attachments

CSIP Model service requests support file attachments as service input. Files are attached as FormDataMultiPart in the POST request. The CSIP infrastructure manages files by extracting them from the request, storing them on the workspace of the service session, and publishing URIs to make them available. Example 3.7 lists the service methods supporting file access.

Example 3.7. File input methods

```
Set<File> getFileInputs()        ❶
int getFileInputsCount()         ❷
File getFileInput(String name)   ❸
boolean hasFileInput(String name)❹
```

❶ Get all service request input files.

❷ Get the total number of input files.

❸ Get a single file by name. The name must be the file name or the relative path. This method returns the File object containing the full path name within the workspace.

❹ Check if a file exists in the workspace.

All files that are part of the request are copied unchanged into the session workspace. If the incoming file is an archive (`*.zip, *.gz, *.tar, *tgz, ...`) CSIP automatically uncompresses and extracts it into the workspace. It is good practice to compress large files to reduce network I/O requirements.

Example: File attachment

Files are included in the CSIP client request as HTTP mulitpart attachments. The CSIP service infrastructure automatically handles recognition and extraction of input files into the service session workspace. Input files are unpacked from the request and stored in their original form. Using the process() methods of a service the files can referenced by name.

The curl call below invokes a CSIP service and attaches a file. The file "scott.kmz" is attached in this example.

```
curl -X POST -H "Accept:application/json" \
  "http://localhost/csip-lamps/m/lamps/1.0" -F file1=@c:/scott.kmz
```

The submitted file can now be referenced in the service implementation.

```
protected void preProcess() {
    ...
    File gem = getFileInput("scott.kmz")
    ...
}
```

If the `getFileInput()` method returns a reference to the file, it is guaranteed to exist.

3.4. Providing Service Output

Service output is usually constructed in the `postProcess()` method of a service. Similar to processing parameter input and metainfo, it is not necessary to manually generate the JSON data structure of the response. A family of `putResult(...)` helper methods are provided to add result values with `metainfo` to the `result` section of the response in the proper form.

The Example 3.8 shows methods available for output generation.

Example 3.8. Service Output Generation

```
void putResult(String name, String val, String unit, String descr)  ❶
void putResult(String name, String val, String unit)
void putResult(String name, String val)
void putResult(String name, int val, String unit, String descr)
void putResult(String name, int val, String unit)
void putResult(String name, int val)
void putResult(String name, double val, String unit, String descr)
void putResult(String name, double val, String unit)
void putResult(String name, double val)
void putResult(String name, boolean val, String unit, String descr)
void putResult(String name, boolean val, String unit)
void putResult(String name, boolean val)
void putResult(String name, JSONObject val, String unit, String descr)
void putResult(String name, JSONObject val, String unit)
void putResult(String name, JSONObject val)

void putResult(File file)                                           ❷
void putResult(File file, String descr)
void putResult(File... file)
```

❶ The `putResult(...)` methods have variants to add a result entry with value, unit, and description. Different value types are supported.

❷ The `putResult(File ...)` method adds a file to the result section. A URL is provided to the client in the response JSON to support retrieving the file in a separate `HTTP/GET` request.

Example: Result Generation

Results are generated in the `postProcess()` method. Result entries are added using `putResult()` helper methods. The Example 3.9 shows a service code fragment creating result entries.

Example 3.9. Example Results

```
...
    @Override
    protected void postProcess() throws Exception {
        putResult("runoff", model.getRunoffQ(), "cfs", "Runoff value");
        putResult("timeofconcentration", model.getTimeOfConcentration());
        putResult("unitpeakdischarge", model.getUnitPeakDischarge());
    }
...
```

The example above creates three result entries for a service response, `runoff`, `timeofconcentration`, and `unitpeakdischarge`. The resulting response is shown below.

```
...
  "result": [
    {
      "name": "runoff",
      "value": 12.75,
      "unit": "cfs",
      "description":"Runoff value"
    },
    {
      "name": "timeofconcentration",
      "value": 0.727178
    },
    {
      "name": "unitpeakdischarge",
      "value": 0.370022
    }
  ]
...
```

No JSON structure management is required. The `putResult()` methods manage proper JSON creation for the response.

3.4.1. Report Generation

Report generation can optionally be incorporated in post processing. It allows sever-side preparation of additional service output that can optionally be fetched by the client. The report typically contains secondary output that has lower significance than the primary service result output. Reports are generated in the `report()` method of a service.

The list of all `putReport()` methods is shown in Example 3.10.

Example 3.10. Report generation API

```
    void putReport(String name, String val, String unit, String descr)
    void putReport(String name, String val, String unit)
    void putReport(String name, String val)
    void putReport(String name, int val, String unit, String descr)
    void putReport(String name, int val, String unit)
    void putReport(String name, int val)
    void putReport(String name, double val, String unit, String descr)
    void putReport(String name, double val, String unit)
    void putReport(String name, double val)
    void putReport(String name, boolean val, String unit, String descr)
    void putReport(String name, boolean val, String unit)
    void putReport(String name, boolean val)
    void putReport(String name, JSONObject val, String unit, String descr)
    void putReport(String name, JSONObject val, String unit)
    void putReport(String name, JSONObject val)
    void putReport(File file)

    void putReport(File file, String descr)
    void putReport(File... file)
```

`The putReport()` helper methods have semantics similar to `the putResult()` methods.

Example: Report Generation

Report generation must be implemented within the report() method. Report entries are added using the `putReport()` family of helper methods. The example shows a service code fragment creating report entries.

Example 3.11. Example Reports

```
...
double runoff;

public void report() {
   putReport("peakrunoff", 2.34, "cfs");
   putReport(new File("summary-report.pdf"), "summary");
}
...
```

3.5. Resources

Every service requires resources, such as static data files, lookup tables, or native executables that contribute to service execution. Resources can be bundled with the service and extracted and referenced at service run time.

CSIP provides flexible bundling at development time and referencing at runtime of all supplemental resources. CSIP Service annotations provide a means to describe resources. Resource annotations can be found in the package `csip.annotations`.

Resource annotation elements are shown in Example 3.12. Resource annotations can be used to describe artifacts such as data archives, executable files, configuration files, references to existing resources, java classes, or output files to capture.

Example 3.12. Resource Annotations

```
public @interface Resource {

    /** The path to the file within the war file or file system.
     */
    String file();

    /** The type of the resource.
     */
    ResourceType type();

    /** The id of that resource.
     */
    String id() default "";

    /** Should the file be executed via wine.
     */
    boolean wine() default false;

    /** Default executable arguments, separated by space
     */
    String args() default "";

    /** Default environment variables to be used for execution.
     */
    String env() default "";
}
```

The specification above lists all annotation methods within the Resource interface. The methods may be applicable for different `ResourceTypes`, as they are listed in Example 3.13. The methods `wine()`, `args()`, and `env()` can only be used if the resource to described is an executable (`EXECUTABLE`), an OMS DSL element (`OMS_DSL`), or a Java class name (`CLASSNAME`). In these cases, additional information about the execution environment can be provided. Most importantly, the `file()` method specifies the resource bundled in the service WAR file, the `type()` method categorizes it, and the `id()` method allows referencing from the service implementation code. The Example 3.13 shows supported resource types in CSIP.

Example 3.13. Resource types categories

```
public enum ResourceType {
    OUTPUT,       // The resource describes output of the model service.
    ARCHIVE,      // This resource is a zip file and it will be unpacked.
    FILE,         // This resource is a file.
    EXECUTABLE,   // This resource is an executable file.
    REFERENCE,    // This resource is a file reference.
    JAR,          // This resource is a jar file.
    OMS_DSL,      // This resource is an OMS DSL file.
    CLASSNAME     // This resource is a java class name.
}
```

Within a service implementation, resources can be accessed by their ID. The CSIP infrastructure ensures that the resource is extracted from the service package and mapped to the ID as provided in the annotation. Two methods are shown in Example 3.14.

Example 3.14. Resource access by ID

```
File getResourceFile(String id)         ❶
Executable getResourceExe(String id)    ❷
```

❶ Provides a reference to the resource on the local file system. The reference can be to a file or directory if the resource description refers to an archive or directory.

❷ Provides an executable for a given ID. The resource is expected to be a native executable. CSIP unpacks the file and maps it into an executable interface to allow the service to control its execution.

The following are example applications of resource annotations.

Example: Defining File Resources

Resource definitions are included in the service implementation as annotations. Annotations prefix the service class as shown in Example 3.15. Here, a data file that serves as a lookup table for a service is used.

Example 3.15. Resource definition and access by ID

```
import static csip.annotations.ResourceType.*;

@Resource(file="/data/lookup.txt", type=FILE, id="lkp")   ❶
public class V_1 extends ModelDataService {

    public void process() {
         // ...
         File f = getResourceFile("lkp");                 ❷
         // do something with f.
    }
}
```

❶ The resource annotation identifies the file "`/data/lookup.txt`" with the ID "`lkp`".

❷ Within any service implementation method, the file can be accessed by its ID.
The file "`/data/lookup.txt`" has to be bundled with the service war using the specified path.

Example: Bundling Archives

Data archives (compressed folders with sub-folders) can be bundled with a service. If a resource type is set to `ARCHIVE`, the file will be uncompressed/unpacked. Looking up the archive resource by ID will return the unpacked root folder of the archive (Example 3.16).

Example 3.16. Resource definition of data archives

```
import static csip.annotations.ResourceType.*;

@Resource(file="/data/arcdems.zip", type=ARCHIVE, id="dems")
public class V_1 extends ModelDataService {

    public void process() {
         // ...
         File f = getResourceFile("dems");
         // do something with f.
    }
}
```

Example: Bundling Executables

Like any other file, executable files can be bundled with the service. The `EXECUTABLE` type denotes such a resource. The `getResourceExe()` method fetches the unbundled exe by ID and returns an Executable interface for it. The interface methods support interaction, such as redirecting output, setting command line arguments and environment variables, and execution.

Example 3.17 show the use of the resource annotations for an executable. The resource annotations allow multiple resource definitions to be attached to the same service.

Note

In Java 8, resource annotations are not needed since annotations of the same type are repeatable.

Example 3.17. Resource definition of Executables

```
import static csip.annotations.ResourceType.*;

@Resource({
  @Resource(file="/bin/tr20.exe", type=EXECUTABLE, id="tr20"),  ❶
  @Resource(file="*.out *.dbg", type=OUTPUT)                    ❷
})
public class V_1 extends ModelDataService {

    public void process() {
         // ...
         Executable e = getResourceExe("tr20");                 ❸
         e.exec();
         // do something with f.
    }
}
```

❶ This annotation describes the '`tr20.exe`' executable which is mapped to the "tr20" ID.

❷ The OUTPUT annotation lists files that should be captured as output for a service. It may contain wildcards to specify groups of files. An `id` is not required here.

❸ The reference to the resource is obtained via the Executable interface, and the program gets invoked in the following line.

3.5.1. Architecture Variants

CSIP services can bundle executables for different architectures to enable flexible deployment to different operating systems. The variable `${arch}` can be used in the string referencing the resource to resolve the actual architecture at service runtime. This is shown below.

```
 @Resource(file="/bin/${arch}/tr20.exe", type=EXECUTABLE, id="tr20")
```

When the executable is requested using the `getResourceExe()` method, the "arch" variable will be replaced with the host machine's actual architecture fingerprint:

"win-x86"	Windows, 32bit
"win-amd64"	Windows, 64bit
"lin-x86"	Linux, 32bit
"lin-amd64"	Linux, 64bit
"mac-amd64"	Mac OSX, 64bit

Example: To deploy the same service to various architectures, a `tr20.exe` native binary must exist as `/bin/win-amd64/tr20.exe` and `/bin/lin-amd64/tr20.exe`, respectively. The executable must be compiled for both architectures. Only one resource definition is sufficient to reference the executable. At service runtime, the correct executable is selected based on the service hosting architecture.

3.5.2. Automated Execution

Automated execution of a model within a service simplifies the integration of existing executables into services. Annotation-only based services should be sufficient for most external models requiring a fixed set of inputs.

The executable is annotated with the ID "`auto`". An example of a `ModelDataService` is shown in Example 3.18. Here, no custom service code is needed.

An external model data service must subclass `ModelDataService`. All service annotations should be added to this subclass. In addition to `@Resource`, the `@Name`, `@Description`, `@Path`, and `@Polling` annotations should be used. `@Path is required.`

Example 3.18. SWAT External Executable Service

```
import csip.ModelDataService;
import csip.annotations.*;
import static csip.annotations.ResourceType.*;
import javax.ws.rs.Path;
import oms3.annotations.*;

/**
 * SWAT Service. Plain execution of the original executable.
 *
 * @author od
 */
@Name("SWAT")
@Description("Soil Water Assessment Tool model service. (SWAT2012 Rev. 627)")
@Path("m/swat/1.0")
@Polling(first = 5000, next = 2000)
@Resources({
  @Resource(file = "/bin/lin-amd64/swat2012_627.exe", type = EXECUTABLE, id = "auto"),❶
  @Resource(file = "output.* *.out chan.deg fin.fin " +
                   "watout.dat input.std stdout.txt stderr.txt", type = OUTPUT)       ❷
})
public class V1_0 extends ModelDataService {
  // done.
}
```

❶ The "`auto`" identifier tags the resource for simple auto-execution when the service is invoked. There is no need to explicitly fetch the resource in the `process()` method of the service and programmatically execute it there.

❷ The resource definition lists all files (wildcards allowed) that should be returned as references for download as part of the response. The `file` entry lists names and patterns of those files separated by whitespaces.

The same auto-execution approach can be applied to Java and OMS models.

3.6. Client Control

3.6.1. Service Polling

Model services may require significant time to execute, based on the underlying model and logic. It is not uncommon that services run for seconds, minutes or hours per service call. Asynchronous service invocations for longer running services allow clients to stay responsive during service execution. Clients must periodically check the status of the service completion as described in Section 2.3.2.2.

To support a reasonable polling frequency for clients, the service should describe when and how often a client should poll. The `@Polling` annotation, or the methods listed in the examples provide two alternatives to indicate:

1. the first time that a client should query the service for model results,

2. and a subsequent polling interval.

A custom model service can either overwrite the methods in Example 3.19 , or use the annotations in Example 3.20 to specify the desired polling frequency. An accurate polling indication by the model service and its respectful handling at the client side avoids flooding the network with unnecessary requests. For example: If a model runs for 10 minutes, it is not necessary to query the execution status every second after asynchronous submission.

Example 3.19. Polling methods

```
    long getNextPoll()
    long getFirstPoll()
```

Polling methods are preferred if the service execution time varies based on the size of the input data or other factors. In this case the service can provide a dynamic estimate for a polling frequency based on the size or nature of the inputs.

Example 3.20. Polling Annotations

```
public @interface Polling {
    long first() default -1;
    long next() default -1;
}
```

Polling annotation are preferred if service execution time is mostly constant, it does not depend on the size of the inputs.

The `long` return values for first/next polling should be specified in milliseconds.

3.6.1.1. Example

A model service is expected to run for at least 3 minutes. After that, the service status may be queried every 2 seconds. A polling annotation based service configuration is shown in Example 3.21.

Example 3.21. Polling Annotation Example

```
@Polling(first = 180000, next = 2000)
public class MService1 extends ModelDataService {
...
}
```

For a more dynamic setup a computed polling interval can provide better accuracy and reduce unnecessary network traffic.

Example 3.22. Polling Method Example

```
public class MService1 extends ModelDataService {

  //....

  protected long getFirstPoll() {
     return no_years_in_data * 100;
  }

  protected long getNextPoll() {
     return 1000;
  }
}
```

In the example Example 3.22, model execution requires approximately 100 ms per simulation year. Therefore, the first poll can be computed as shown; the next poll is constant.

3.7. Reporting Progress

If a service is executed asynchronously because of expected long execution time, the service may communicate its completion status to the client to give meaningful feedback. Such feedback can be numerical to indicate a percent completeness, or text can describe the current state of service execution.

Example 3.23. Progress Reporting

```
    void setProgress(String progress)
    void setProgress(int progress)
```

The first method allows reporting of simple messages, the second variant takes a numerical value between 0 and 100.

Example: Progress Reporting

Anytime during service execution progress can be reported. If a client queries the service status, the most recent message is returned.

```
    setProgress("Fetching landuse data ...");
    // more processing
    setProgress("Parsing polygon information ...");
    // more processing

    // -or-
    setProgress(10);   // means 10% completed ..
```

3.8. Error handling and Error Propagation

If a service fails the custom service implementation can throw a `ServiceException` (Example 3.24). The message passed into a `ServiceException` is propagated back to the client as a part of the `metainfo` of the response.

Example 3.24. ServiceException

```
public class ServiceException extends Exception {
    public ServiceException(String message);
    public ServiceException(String message, Throwable cause);
    public ServiceException(Throwable cause);
}
```

The `ServiceException` class is a checked exception.

Example: Throwing a ServiceException

The example below shows the use of a `ServiceException` within the `process()` method. The presence of an input file is checked and if it is missing an exception is thrown.

```
import csip.*;

public class V_1 extends ModelDataService {

    public void process() throws Exception {
        //..
        File f = getResourceFile("dem.asc");
        if (f==null) {
           throw new ServiceException("Missing input DEM input file");
        }
        // ...
    }
}
```

The exception appears in the client response in the response metainfo as an exception message. This is shown below.

```
{
   "metainfo" : {
       status: "Failed",
       error: "Missing input DEM input file"
```

```
        ...
    }
    "parameter" : [
        ...
    ]
}
```

If the status of a service run is "Failed", an error message can always be found. A client can check the status code and the content of the "error" entry.

If an unchecked exception occurs during service execution, the error value will be the full stack trace.

3.8.1. Logging

Logging is enabled and configured by default for each CSIP session. The session ID (suid) is used to identify the logging output. This support separation and filtering of logging by session. The CSIP configuration property "`csip.logging.enabled`" is a boolean that can modified at any time to disable or enable logging.

The `ModelDataService` class provides a protected logger called "`LOG`" that can be accessed in a custom service implementation Example 3.25. The LOG field is final, thus it cannot be altered. The default log level is set to "`INFO`", however it can be set using the configuration property "`csip.logging.level`". Table 4.1 describes all logging related properties in detail.

Example 3.25. Logging Example

```
public void process() throws Exception {
    //...
    if (LOG.isLoggable(Level.INFO) {
        LOG.info("connecting do database ...");
    }
    // ..
```

CSIP uses standard Java logging. Predefined log levels include: `OFF, ALL, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST`.

Logging information can be captured using a separate server as it is described in Section 4.2.5. The session user interface provides a link to the log record for each individual session. This way logging information can be obtained from any client for debugging and tracing purposes without having direct access to the application server. Logging for a CSIP deployment is centrally configured. Each CSIP deployment can specify its own logging infrastructure configuration.

Chapter 4. CSIP Infrastructure

This chapter provides an overview of different deployment alternatives for the CSIP infrastructure. CSIP infrastructure requirements depend on development preferences and service hosting requirements (e.g. service compute, disk, and network I/O requirements) CSIP may be deployed for development using Linux, Windows, or Mac OS X. For production deployment Linux or Windows are preferred.

The following components are required to run CSIP

- Java 7+
- Apache Tomcat 7+
- CSIP libraries can be downloaded from http://csip.javaforge.com.
- Redis (NoSQL database, optional) [1]

For a highly scalable CSIP environment it is recommended to use a software or hardware load balancer. Load balancers serve as an HTTP proxy to distribute requests across multiple CSIP servers. HAProxy is a popular software load balancer.

4.1. CSIP Configurations

CSIP configuration is property based. Properties control the internal behavior of CSIP and allow administrator(s) to adjust configuration according to infrastructure constraints, scalability, fail-over requirements, development needs, and architecture partitioning strategies.

4.1.1. Configuration properties

The table below lists CSIP properties that can be customized for specific CSIP deployments .

Table 4.1. CSIP System Properties

Key	Default Value	Description
Session		
`csip.session.backend`	`hazelcast`	The backend to use for session management. Valid choices are `hazelcast` or `redis`. If set to `hazelcast` no further configuration is needed. If set to `redis` the properties `csip.session.redis.server` and `csip.redis.session.port` can be used to control the redis connectivity for session management.
`csip.session.redis.server`	`localhost`	The hostname or address of the redis server for session management. The property `csip.session.backend` has to be set to `redis` to use this setting.
`csip.session.redis.port`	`6379`	The port of the redis server for session management. The property `csip.session.backend` has to be set to `redis` to use this setting.

[1]http://www.redis.io

Key	Default Value	Description
`csip.session.ttl`	`300`	The default time in seconds for a session to stay active after the model finishes. All model results will be available for that period. After this period expires the session will be removed and the model results will be removed or archived. This value can be altered using the "`keep_results`" metainfo value of a request.
Archive		
`csip.archive.enabled`	`false`	If this property is set to true, the model request/results session data will be archived to a redis server. If set to false the model results are removed only. Only redis is supported for session archival.
`csip.archive.server`	`localhost`	The hostname/ipadress for the redis archive server. The property `csip.archive.enabled` has to be set to `true` to use this setting.
`csip.archive.port`	`6379`	The port for the redis archive server. The property "`csip.archive.enabled`" has to be set to `true` to use this setting.
`csip.archive.ttl`	`86400`	The default time in seconds for an entry to stay in the archive. All archived model results will be retained for the TTL period after the session expired. After expiration the archive will be removed, a value of `-1` specifies no expiration, the archive entry stays forever. 3600 - one hour, 86400 - one day, 604800 - one week 2592000 - one month, etc.
Logging		
`csip.logging.enabled`	`false`	If set to true remote service logging will be enabled and a specific log handler submits the log for each session to a redis server instance. Only redis is supported as the remote logging server. Log records are always sent to Tomcat (local) regardless of the setting of this property.
`csip.logging.level`	`INFO`	The log level for remote service logging. This is only used if `csip.logging.enabled` is set to true. All java.util.logging.Level log levels are usable.
`csip.logging.server`	`localhost`	The remote logging server hostname or ip address. This is only used if `csip.logging.enabled` is set to true.
`csip.logging.port`	`6379`	The remote logging server port. This is only used if `csip.logging.enabled` is set to true.
`csip.logging.ttl`	`86400`	The time to live for a log entry within the remote logging server in seconds. A value of "-1" indicates no expiration.
`csip.logging.keepsevere`	`false`	If set to true, a LOG will never expire in the log store for that session if it contains a "SEVERE" log entry regardless of the "csip.logging.ttl" value.
Directories		

Key	Default Value	Description
`csip.dir`	`/tmp/csip`	The CSIP home directory
`csip.bin.dir`	`/tmp/csip/bin`	The CSIP directories for executable.
`csip.work.dir`	`/tmp/csip/work`	The CSIP directory for sessions.
`csip.results.dir`	`/tmp/csip/results`	The CSIP directory to store results.
`csip.cache.dir`	`/tmp/csip/cache`	The CSIP cache file directory.
`csip.data.dir`	`/tmp/csip/data`	The CSIP data directory.
Miscellaneous		
`csip.timezone`	`MST7MDT`	The default CSIP timezone to be used for all time management.
`csip.ui.enabled`	`true`	If set to false, no service UI for sessions, logging, and archives are provided.
`csip.redis.timeout`	`0`	The connection timeout for all redis connections.
`csip.hazelcast.attempt.period`	`10000`	The period between two Hazelcast connection attempts in milliseconds
`csip.hazelcast.attempt.limit`	`10`	The max number of Hazelcast connection attempts.
`csip.hazelcast.group.name`	`csip`	The Hazelcast group configuration name to use.
`csip.hazelcast.group.password`	`csip-pass`	The Hazelcast group password to use.
`csip.hazelcast.server`	`false`	If set to `true`, this CSIP instance will start and use its own internal hazelcast server. The `csip.hazelcast.group.password` and `csip.hazelcast.group.name` setting for Hazelcast are used for the group configuration. The network configuration defaults to multicast.

4.1.2. Changing the Configuration

It is recommended to perform configuration changes right after context loading and before any the first CSIP service request is made. Changes of the session backend are only supported at initialization to avoid inconsistencies for session management.

Property changes should be described in a JSON file containing proper key/value pairs, according to Example 4.1. An HTTP/POST request is used to load the JSON config file and apply the settings to the CSIP server.

Example 4.1. Example CSIP configuration file (service-conf.json)

```
{
    "csip.session.backend" : "redis",
    "csip.session.redis.server" : "192.168.1.100",
    "csip.session.ttl" : 600
}
```

All property values can be passed in as native types or as a string. The entry `csip.session.ttl` can be set with `600` or `"600"`.

The `curl` command below performs the configuration update by issuing the HTTP POST request with the file `service-conf.json`. The service path to be used is always `"../c/conf"`.

```
$ curl -X POST -d @service-conf.json http://localhost:8080/csip-test/c/conf
```

The configuration service returns the current configuration or an error message.

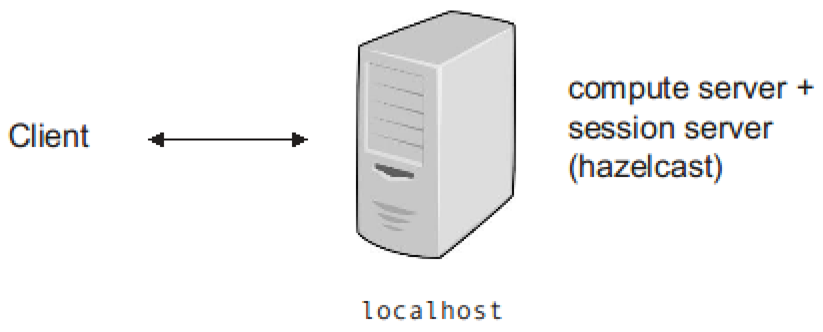
4.2. Deployments

This section shows typical configurations for different CSIP deployments highlighting development and deployment of CSIP services using various environments. If a single server deployment is chosen, all packages have to be installed on the same machine. A multi-server deployment should be used for production level deployments to support separation of infrastructure components to ensure redundancy, failover, and scalability. Infrastructure components are partitioned to reside on separate machines to eliminate resource contention.

4.2.1. Simplest Single Server Deployment

On a single server deployment all CSIP components run on the same server. Sessions are managed using the Hazelcast backend and remain available for 10 minutes after compleition. Session archival and logging are not enabled. This is the default CSIP deployment configuration.

Figure 4.1. Singe server deployment



Example 4.2. Single Server Deployment

```
{
    "csip.session.backend" : "hazelcast",
    "csip.session.ttl" : "600",
    "csip.logging.enabled" : false,
    "csip.archive.enabled" : false,
    "csip.hazelcast.server" : true
}
```

This CSIP instance uses an internal hazelcast server for session management. There is no need to start and manage it manually.

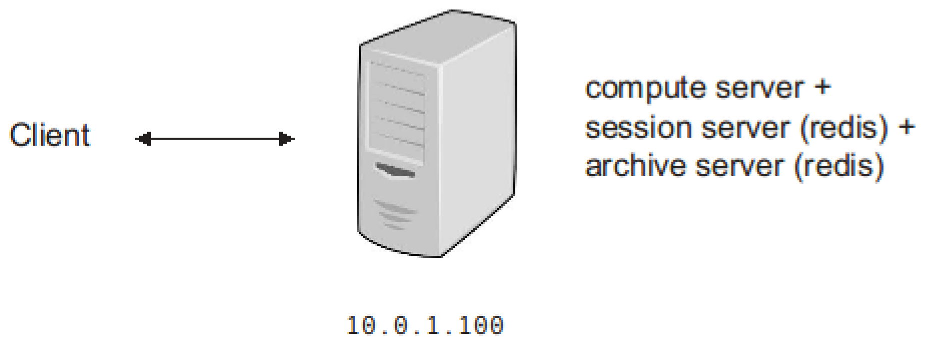
Server Configuration

Server A (localhost)
    Tomcat, Hazelcast

4.2.2. Single Server Deployment with Archival

In this single server configuration all sessions are managed using the redis backend and will stay available for 5 minutes after finish. Session are archived to the redis server after 5 minutes. Logging is disabled.

Figure 4.2. Singe server deployment



The configuration is shown below.

Example 4.3. Single Server Deployment (ii)

```
{
    "csip.session.backend" : "redis",
    "csip.session.redis.server" : "10.0.1.100",
    "csip.session.ttl" : "300",
    "csip.archive.enabled" : true,
    "csip.archive.server" : "10.0.1.100"
    "csip.archive.ttl" : 604800
}
```

Archived session data is retained for a week (604800 sec) before deletion.

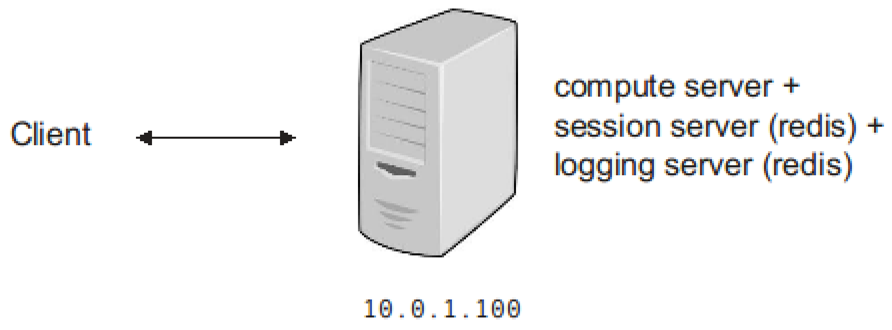
Server Configuration

Server A (10.0.1.100)
    tomcat, Redis

4.2.3. Single Server Deployment with Logging

In this single server configuration all sessions are managed using the redis backend and will stay available for 5 minutes. Session data is deleted afterwards. Logging is enabled.

Figure 4.3. Singe-Server Deployment for Development



Example 4.4. Single-Server Deployment for Development

```
{
    "csip.session.backend" : "redis",
    "csip.session.redis.server" : "10.0.1.100",
    "csip.session.ttl" : "600"
    "csip.archive.enabled" : false,
    "csip.logging.enabled" : true
    "csip.logging.server" : 10.0.1.100,
    "csip.logging.level" : ALL,
}
```

This single server configuration is suitable for local development of CSIP services since archives are not required. The log level is set to `ALL` to capture all logging output. If development occurs on the local machine, the `~.server` properties can be set to `localhost`.

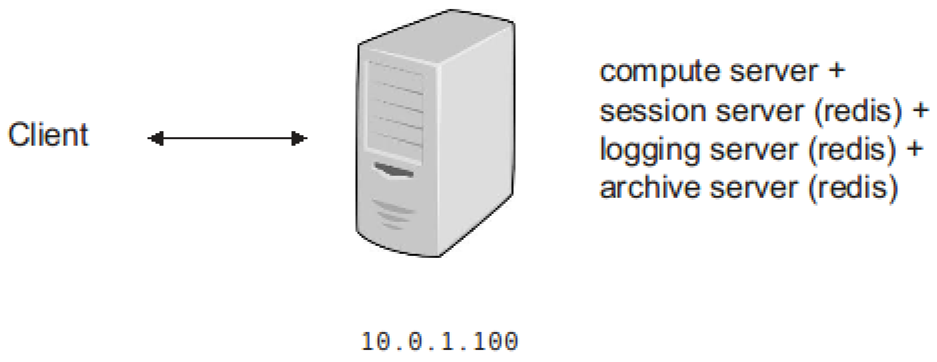
Server Configuration

Server A (10.0.1.100)
 Tomcat CSIP , Redis

4.2.4. Single Server Deployment with Archival and Logging

In this single server configuration all sessions are managed using the Redis backend and stay available for 10 minutes after completion. Sessions are archived to the local Redis server. Logging is enabled.

Figure 4.4. Single-Server Deployment with Archival



The CSIP configuration is shown below.

Example 4.5. Single-Server Deployment with Archival

```
{
    "csip.session.backend" : "redis",
    "csip.session.redis.server" : "10.0.1.100",
    "csip.session.ttl" : "600"
    "csip.archive.enabled" : true,
    "csip.archive.server" : 10.0.1.100,
    "csip.archive.ttl" : -1,
    "csip.logging.enabled" : true,
    "csip.logging.server" : 10.0.1.100,
    "csip.logging.level" : WARNING,
}
```

This single server configuration is most suitable for local deployment of CSIP services. The log level is set to `WARNING` to capture only relevant logging output. Note that the same Redis instance is used for session management, archival, and logging. This simplifies data management but may represent a bottleneck when service traffic is high. Redis memory resources should be carefully monitored since archive entries will not expire.

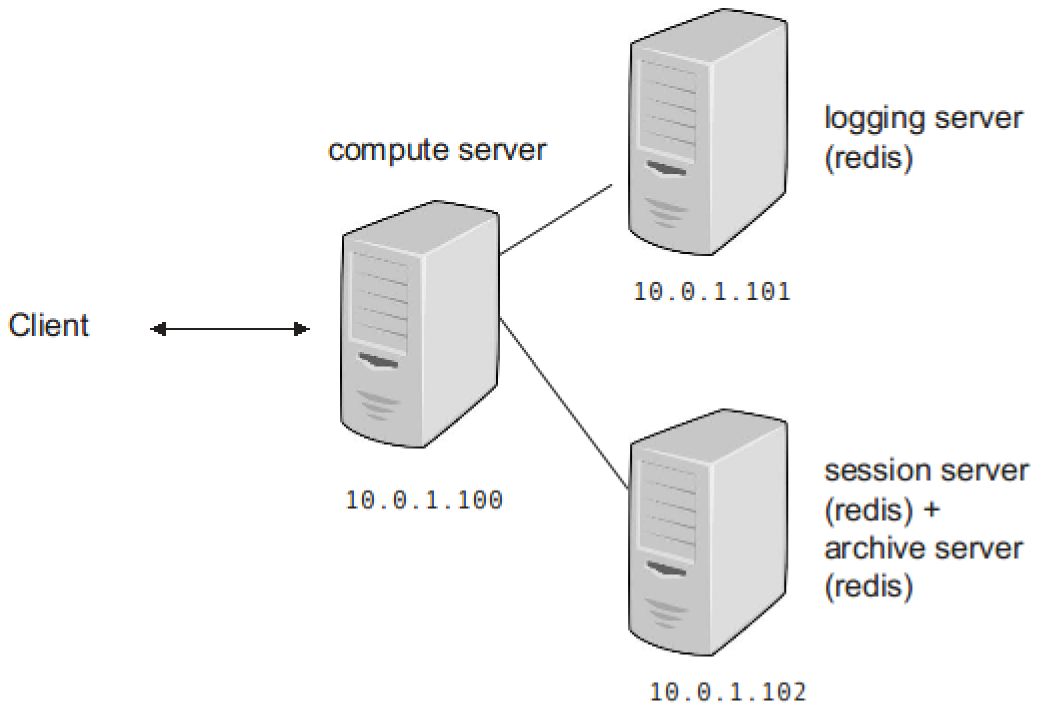
Server Configuration

Server A (10.0.1.100)
    Tomcat CSIP, Redis

4.2.5. Multi-Server Deployment

Multi-server deployment addresses production needs such as scalability and failover. Infrastructure components are partitioned so that they reside on separate machines to eliminate resource contention.

Figure 4.5. Multi-Server Deployment: separate Logging and Archival



Typically multiple CSIP servers are used to service modeling and data service requests. The configuration below uses one Tomcat and two Redis servers for session management, archival, and logging.

Example 4.6. Multi-Server Deployment: separate Logging and Archival

```
{
    "csip.session.backend" : "redis",
    "csip.session.redis.server" : "10.0.1.101",
    "csip.session.ttl" : "600"
    "csip.archive.enabled" : true,
    "csip.archive.server" : 10.0.1.101,
    "csip.archive.ttl" : -1,
    "csip.logging.enabled" : true,
    "csip.logging.server" : 10.0.1.102,
    "csip.logging.level" : WARNING,
}
```

The archive entries never expire, only warning messages are logged.

Server Configuration

Server A (10.0.1.100)
    Tomcat CSIP

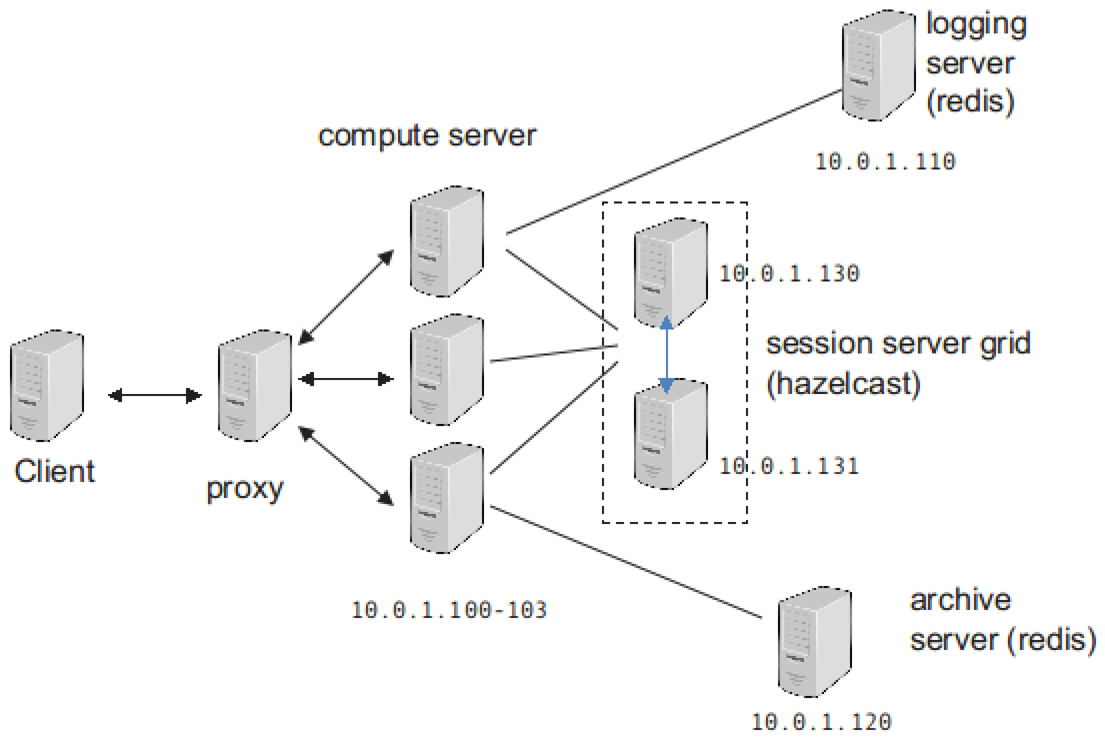
Server B (10.0.1.101)
    Redis (session, archive)

Server C (10.0.1.102)
    Redis (logging)

4.2.6. Scalable Deployment (Multiple Server)

A typical production level deployment accounts for scalability, failover, and redundancy of resources. CSIP can be configured to cover all of those aspects.

Figure 4.6. Multi Server Deployment in Production



4.2.7. Validating the Configuration

[TBD]

4.2.8. Tips / Best Practices / FAQ

Do I have to restart CSIP when I restart hazelcast?
: No, hazelcast will reconnect to a running CSIP instance.

How do I delete keys for all logging entries in Redis?

```
# This operation erases log entries from Redis
$ redis-cli keys "log:*" | xargs redis-cli DEL
```

How do I list all archive keys in Redis?

```
$ redis-cli keys "archive:*"
```

How do I flush all keys?

```
# This operation erases all redis server content
$ redis-cli flushall
```

4.3. ModelServices User Interface

CSIP provides a web-based user interface to monitor services. This includes: (i) current session status, (ii) a view to explore logging and exception information for debugging purposes, (iii) a user interface for managing archives of session data, and (iv) a configuration query which describes the current CSIP configuration.

The session, logging, and archive GUI are always bound to the context of a web application. Only services that are part of that context can be explored here. If logging or archival are disabled for a context, no GUI can be used.

The CSIP property "`csip.ui.enabled`" controls the general accessibility of the UI for a CSIP context. It defaults to true.

There are three URLs available to access the user interfaces:

`http://<host>/<context>/c/session`
: Shows the current session status for the service context.

`http://<host>/<context>/c/logging`
: Shows all the logging records by session for the given service context.

`http://<host>/<context>/c/archive`
: Shows all archived sessions.

The "c" path element delineates CSIP console requests and actions.

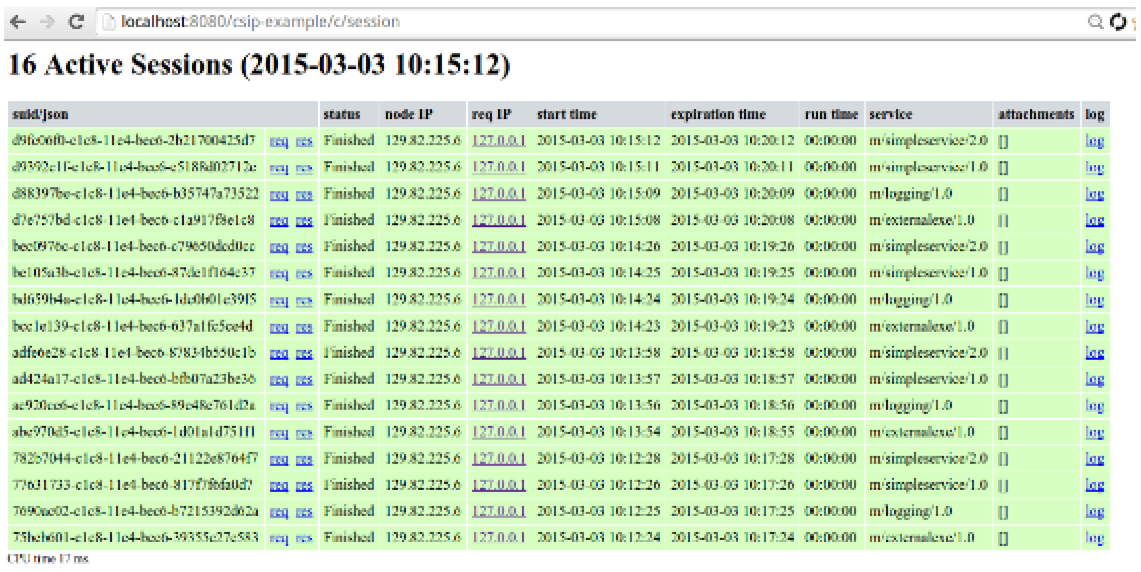
4.3.1. Session User Interface

The CSIP session UI for a given context can be accessed using the following URL:

```
GET http://<host>/<context>/c/session
```

The session UI is shown in Figure 4.7. The table shows all relevant information for a session.

Figure 4.7. Session User Interface



localhost:8080/csip-example/c/session

16 Active Sessions (2015-03-03 10:15:12)

suid/json		status	node IP	req IP	start time	expiration time	run time	service	attachments	log
d9fe06f0-c1c8-11e4-bec6-2b21700425d7	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:15:12	2015-03-03 10:20:12	00:00:00	m/simpleservice/2.0	[]	log
d9392e1f-c1c8-11e4-bec6-e5188d02712e	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:15:11	2015-03-03 10:20:11	00:00:00	m/simpleservice/1.0	[]	log
d88397be-c1c8-11e4-bec6-b35747a73522	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:15:09	2015-03-03 10:20:09	00:00:00	m/logging/1.0	[]	log
d7e757bd-c1c8-11e4-bec6-c1a917f8e1c8	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:15:08	2015-03-03 10:20:08	00:00:00	m/externalexe/1.0	[]	log
bec0976c-c1c8-11e4-bec6-c79650dcd0cc	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:14:26	2015-03-03 10:19:26	00:00:00	m/simpleservice/2.0	[]	log
be105a3b-c1c8-11e4-bec6-87de1f164e37	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:14:25	2015-03-03 10:19:25	00:00:00	m/simpleservice/1.0	[]	log
bd639b4a-c1c8-11e4-bec6-1da0b01e39f5	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:14:24	2015-03-03 10:19:24	00:00:00	m/logging/1.0	[]	log
bcc1e139-c1c8-11e4-bec6-637a1fc5ce4d	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:14:23	2015-03-03 10:19:23	00:00:00	m/externalexe/1.0	[]	log
adfe6e28-c1c8-11e4-bec6-87834b550c1b	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:13:58	2015-03-03 10:18:58	00:00:00	m/simpleservice/2.0	[]	log
ad424a17-c1c8-11e4-bec6-bfb07a23be36	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:13:57	2015-03-03 10:18:57	00:00:00	m/simpleservice/1.0	[]	log
ac920ce6-c1c8-11e4-bec6-89e48e761d2a	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:13:56	2015-03-03 10:18:56	00:00:00	m/logging/1.0	[]	log
abe970d5-c1c8-11e4-bec6-1d01a1d751f1	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:13:54	2015-03-03 10:18:55	00:00:00	m/externalexe/1.0	[]	log
782b7044-c1c8-11e4-bec6-21122e8764f7	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:12:28	2015-03-03 10:17:28	00:00:00	m/simpleservice/2.0	[]	log
77631733-c1c8-11e4-bec6-817f7fbfa0d7	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:12:26	2015-03-03 10:17:26	00:00:00	m/simpleservice/1.0	[]	log
7690ae02-c1c8-11e4-bec6-b7215392d62a	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:12:25	2015-03-03 10:17:25	00:00:00	m/logging/1.0	[]	log
75bcb601-c1c8-11e4-bec6-39355e27e583	req res	Finished	129.82.225.6	127.0.0.1	2015-03-03 10:12:24	2015-03-03 10:17:24	00:00:00	m/externalexe/1.0	[]	log

CPU time 17 ms

The first column lists the simulation id (suid) for a given service session with request and response JSON links. The response JSON is available once the session is completes. The status, client IP, and worker node IP is provided followed by the time of the request, the time when the service session (and its output data) will expire, the execution time of the service, and the service end point. Finally, request attachment filenames are listed and the session log is provided.

The rows in this view are colored according to the status. Color changes with the status.

For the request, response, requesting IP address, and log, hyperlinks provide additional information to the user when clicked (e.g. IP geolocation [2], or log listing).

Query parameters can be appended to the session URL to control the appearance of the table. As a default the most recent session is shown on top of this table.

col
: The column to sort the table on, e.g. `... ?col=5...`, default is 5

order
: The direction for sorting, ascending (`"a"`) or descending (`"d"`), e.g. `..?order=d..`, default is "d"

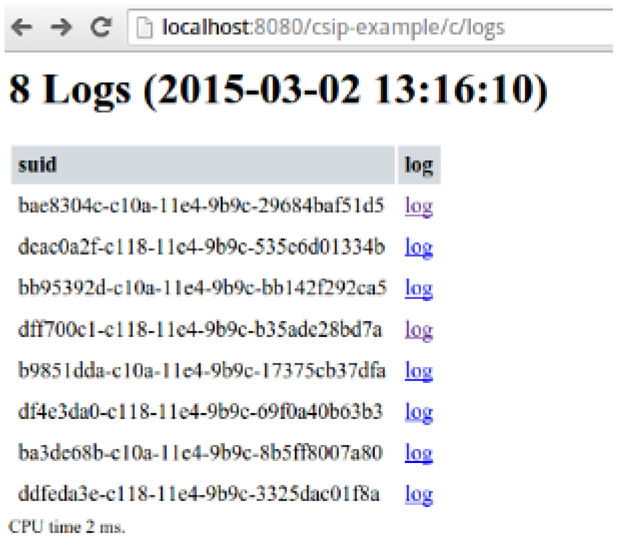
Note that the session view is not updated on a periodic basis. Users van harness browser extensions to refresh the content so often (e.g. every 5 seconds).

4.3.2. Logging User Interface

The Logging user interface allows access to log records on a "per-session" basis. Sessions are listed with their session id and a link to the log record Figure 4.8. Note that those sessions are not ordered. A user search for sessions by id can fetch the associated log entry.

[2]http://freegeoip.net

Figure 4.8. Logging User Interface



8 Logs (2015-03-02 13:16:10)

suid	log
bae8304c-c10a-11e4-9b9c-29684baf51d5	log
deac0a2f-c118-11e4-9b9c-535e6d01334b	log
bb95392d-c10a-11e4-9b9c-bb142f292ca5	log
dff700c1-c118-11e4-9b9c-b35ade28bd7a	log
b9851dda-c10a-11e4-9b9c-17375cb37dfa	log
df4e3da0-c118-11e4-9b9c-69f0a40b63b3	log
ba3de68b-c10a-11e4-9b9c-8b5ff8007a80	log
ddfeda3e-c118-11e4-9b9c-3325dac01f8a	log

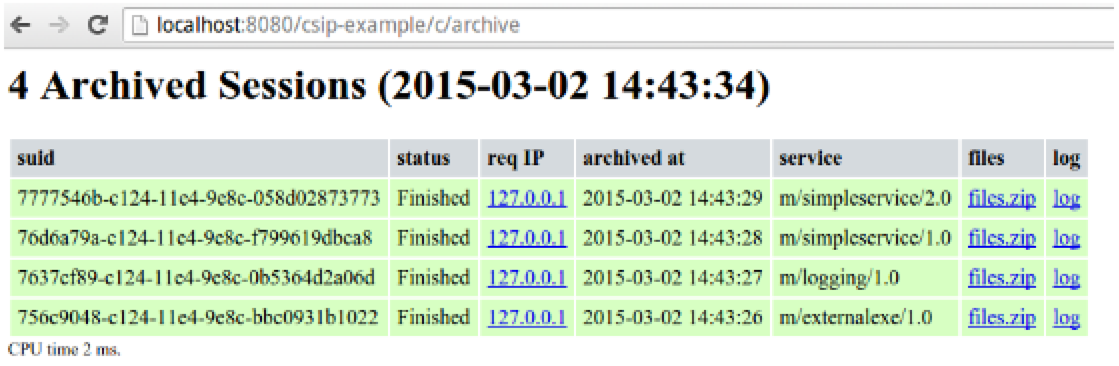
CPU time 2 ms.

The logging UI link is identical to the session UI link. Logging entries do not expire with the session. They remain in the logging server until they are expired based on the value in "`csip.logging.ttl`", which defaults to 24 hours. Alternatively, they can be deleted explicitly.

4.3.3. Archive User Interface

The Archive UI provides a complete listing of all archived sessions Figure 4.9. It shows a table, where each row represents a session archive. The session id (suid), the completion status at the end of service execution, the originating request IP address, the time of archival, the service endpoint, the session files, and a link to the log are provided in each session row.

Figure 4.9. Archive User Interface



localhost:8080/csip-example/c/archive

4 Archived Sessions (2015-03-02 14:43:34)

suid	status	req IP	archived at	service	files	log
7777546b-c124-11e4-9e8c-058d02873773	Finished	127.0.0.1	2015-03-02 14:43:29	m/simpleservice/2.0	files.zip	log
76d6a79a-c124-11e4-9e8c-f799619dbca8	Finished	127.0.0.1	2015-03-02 14:43:28	m/simpleservice/1.0	files.zip	log
7637cf89-c124-11e4-9e8c-0b5364d2a06d	Finished	127.0.0.1	2015-03-02 14:43:27	m/logging/1.0	files.zip	log
756c9048-c124-11e4-9e8c-bbc0931b1022	Finished	127.0.0.1	2015-03-02 14:43:26	m/externalexe/1.0	files.zip	log

CPU time 2 ms.

The archive table can be sorted similar to the session UI using the `previously described col/order` query parameter in the URL. As a default, archived sessions are sorted by archival date with the most recent archive shown on top.

Appendix A. ModelServices API

A.1. Executable

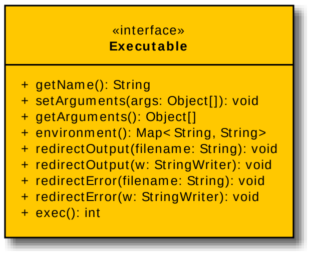
Interface to manage an executable.

A.1.1. Synopsis

```
 public interface Executable {
// Public Methods

  public abstract Map<String, String> environment();

  public abstract int exec()
    throws IOException;

  public abstract Object[] getArguments();

  public abstract String getName();

  public abstract void redirectError(StringWriter w)
    throws IOException;

  public abstract void redirectError(String filename)
    throws IOException;

  public abstract void redirectOutput(StringWriter w)
    throws IOException;

  public abstract void redirectOutput(String filename)
    throws IOException;

  public abstract void setArguments(Object[] args);

}
```

Author

od


«interface»
Executable
+ getName(): String
+ setArguments(args: Object[]): void
+ getArguments(): Object[]
+ environment(): Map< String, String>
+ redirectOutput(filename: String): void
+ redirectOutput(w: StringWriter): void
+ redirectError(filename: String): void
+ redirectError(w: StringWriter): void
+ exec(): int

A.1.2. environment()

```
  public abstract Map<String, String> environment();
```

Get the current environment map

Parameters	
return	the environment.

A.1.3. exec()

```
  public abstract int exec()
    throws IOException;
```

run it.

Parameters	
return	0 if successful, !=0 otherwise

Exceptions

`java.io.IOException`

A.1.4. getArguments()

```
  public abstract Object[] getArguments();
```

Get the executables arguments

Parameters	
return	the arguments

A.1.5. getName()

```
  public abstract String getName();
```

Get the name of the executable.

Parameters	
return	the name of the executable

A.1.6. redirectError(String)

```
  public abstract void redirectError(String filename)
    throws IOException;
```

Redirect stderr to a file in the workspace.

Parameters	
filename	the filename to use relative to the workspace.

Exceptions

`java.io.IOException`

A.1.7. redirectOutput(String)

```
  public abstract void redirectOutput(String filename)
    throws IOException;
```

Redirect stdout to a file in the workspace.

Parameters

filename	the filename to use relative to the workspace.

Exceptions

`java.io.IOException`

A.1.8. setArguments(Object...)

```
  public abstract void setArguments(Object[] args);
```

Set the executable arguments.

Parameters	
args	the executable arguments

A.2. ModelDataService

Base class for all modeling and data services. A service implementation will subclass ModelDataService.

A.2.1. Synopsis

```
 public abstract class ModelDataService {
// Public Static Fields

  public static final String ASYNC = "async";

  public static final String CANCELED = "Canceled";

  public static final String DESCR = "descr";

  public static final String ERROR = "error";

  public static final String EXEC_FAILED = "Error";

  public static final String EXEC_OK ;

  public static final String FAILED = "Failed";

  public static final String FINISHED = "Finished";

  public static final String FORM_PARAM = "param";

  public static final String GEOMETRY = "geometry";

  public static final String IN = "in";

  public static final String INTENT = "intent";

  public static final String KEY_CLOUD_NODE = "cloud_node";

  public static final String KEY_CPU_TIME = "cpu_time";

  public static final String KEY_DESC = "description";

  public static final String KEY_EXPIRATION_DATE = "expiration_date";

  public static final String KEY_FIRST_POLL = "first_poll";

  public static final String KEY_KEEP_RESULTS = "keep_results";

  public static final String KEY_METAINFO = "metainfo";

  public static final String KEY_MODE = "mode";
public static final String KEY_NAME = "name";
public static final String KEY_NEXT_POLL = "next_poll";
public static final String KEY_PARAMETER = "parameter";
public static final String KEY_PARAMETERSETS = "parametersets";
public static final String KEY_PROGRESS = "progress";
public static final String KEY_REPORT = "report";
public static final String KEY_REQUEST_RESULTS = "request-results";
public static final String KEY_REQ_IP = "request_ip";
public static final String KEY_RESULT = "result";
public static final String KEY_SERVICE_URL = "service_url";
public static final String KEY_STATUS = "status";
public static final String KEY_SUUID = "suid";
public static final String KEY_TIME_CLIMATE_QUERY = "timeClimateQuery";
public static final String KEY_TIME_FILEIO = "timeFileIO";
public static final String KEY_TIME_LOGGING = "timeLogging";
public static final String KEY_TIME_MODEL = "timeModel";
public static final String KEY_TIME_SOIL_QUERY = "timeSoilQuery";
public static final String KEY_TIME_TOTAL = "timeTotal";
public static final String KEY_TSTAMP = "tstamp";
public static final String KEY_TZ = "tz";
public static final String KEY_UNIT = "unit";
public static final String KEY_URL = "url";
public static final String MAX = "max";
public static final String MIN = "min";
public static final String OUT = "out";
public static final String RANGE = "range";
public static final String REPORT_DESC = "description";
public static final String REPORT_DIM = "dim";
public static final String REPORT_DIM0 = "dimension0";
public static final String REPORT_FILE = "report.json";
public static final String REPORT_NAME = "name";
public static final String REPORT_TYPE = "type";
public static final String REPORT_UNITS = "units";
public static final String REPORT_VALUE = "value";
public static final String RUNNING = "Running";
public static final String SUBMITTED = "Submitted";
```

```
  public static final String SYNC = "sync";

  public static final String UNIT = "unit";

  public static final String UNKNOWN = "Unknown";

  public static final String VALUE = "value";

// Public Fields

  public Task mt ;

// Protected Fields

  protected final Logger LOG ;

  protected String tz ;

// Public Constructors

  public ModelDataService();

// Public Methods

  @GET @Produces(value="application/json") public final String describeJSON();

  @POST @Produces(value="application/json") @Consumes(value="application/json") public final String execu


  @POST @Produces(value="application/json") @Consumes(value="multipart/form-data") public final String ex


  public final void setMetainfo(JSONObject mi);

  public void setParam(JSONArray parameter);

  public final void setParamMap(Map<String, JSONObject> pm);

  public final void setRequest(JSONObject req);

// Protected Methods

  @Deprecated protected Callable<String> createCallable()
    throws Exception;

  @Deprecated protected JSONArray createReport()
    throws Exception;

  @Deprecated protected JSONArray createResults()
    throws Exception;

  protected boolean getBooleanMetainfo(String name)
    throws ServiceException;

  protected boolean getBooleanParam(String name)
    throws ServiceException;

  protected boolean getBooleanParam(String name,
                                    boolean def)
    throws ServiceException;

  protected final String getCodebase();

  protected double getDoubleMetainfo(String name)
    throws ServiceException;

  protected double getDoubleParam(String name)
```

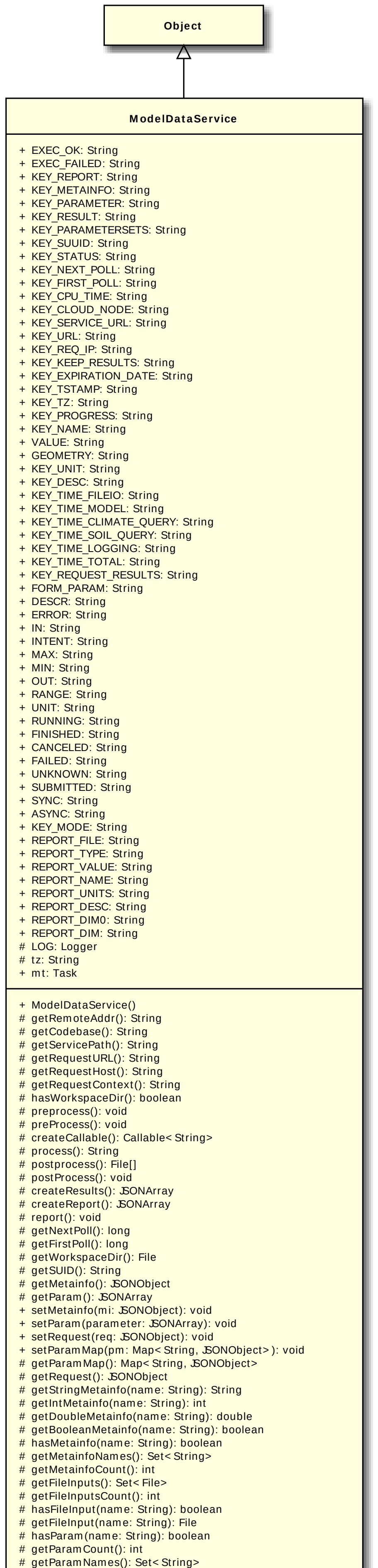
```
    throws ServiceException;

  protected double getDoubleParam(String name,
                                  double def)
    throws ServiceException;

  protected File getFileInput(String name)
    throws ServiceException;

  protected Set<File> getFileInputs();

  protected int getFileInputsCount();

  protected long getFirstPoll();

  protected int getIntMetainfo(String name)
    throws ServiceException;

  protected int getIntParam(String name)
    throws ServiceException;

  protected int getIntParam(String name,
                            int def)
    throws ServiceException;

  protected JSONObject getJSONParam(String name)
    throws ServiceException;

  protected JSONObject getJSONParam(String name,
                                    JSONObject def)
    throws ServiceException;

  protected long getLongParam(String name)
    throws ServiceException;

  protected long getLongParam(String name,
                              long def)
    throws ServiceException;

  protected final JSONObject getMetainfo();

  protected int getMetainfoCount();

  protected Set<String> getMetainfoNames();

  protected long getNextPoll();

  protected final JSONArray getParam();

  protected int getParamCount();

  protected String getParamDescr(String name)
    throws ServiceException;

  protected JSONObject getParamGeometry(String name)
    throws ServiceException;

  protected final Map<String, JSONObject> getParamMap();

  protected Set<String> getParamNames();

  protected String getParamUnit(String name)
    throws ServiceException;

  protected final String getRemoteAddr();

  protected final JSONObject getRequest();

  protected final String getRequestContext();
```

```
protected final String getRequestHost();

protected final String getRequestURL();

protected Executable getResourceExe(String id)
  throws ServiceException;

protected File getResourceFile(String id)
  throws ServiceException;

protected final String getSUID();

protected final String getServicePath();

protected String getStringMetainfo(String name)
  throws ServiceException;

protected String getStringParam(String name)
  throws ServiceException;

protected String getStringParam(String name,
                                String def)
  throws ServiceException;

protected final File getWorkspaceDir();

protected boolean hasFileInput(String name)
  throws ServiceException;

protected boolean hasMetainfo(String name);

protected boolean hasParam(String name);

protected final boolean hasWorkspaceDir();

protected void postProcess()
  throws Exception;

@Deprecated protected File[] postprocess()
  throws Exception;

protected void preProcess()
  throws Exception;

@Deprecated protected void preprocess()
  throws Exception;

protected String process()
  throws Exception;

protected void putReport(File file);

protected void putReport(File file,
                         String descr);

protected void putReport(File[] file);

protected void putReport(String name,
                         boolean val);

protected void putReport(String name,
                         boolean val,
                         String unit);

protected void putReport(String name,
                         boolean val,
                         String unit,
                         String descr);

protected void putReport(String name,
                         double val);

  protected void putReport(String name,
                         double val,
                         String unit);

  protected void putReport(String name,
                         double val,
                         String unit,
                         String descr);

  protected void putReport(String name,
                         int val);

  protected void putReport(String name,
                         int val,
                         String unit);

  protected void putReport(String name,
                         int val,
                         String unit,
                         String descr);

  protected void putReport(String name,
                         String val);

  protected void putReport(String name,
                         String val,
                         String unit);

  protected void putReport(String name,
                         String val,
                         String unit,
                         String descr);

  protected void putReport(String name,
                         JSONObject val);

  protected void putReport(String name,
                         JSONObject val,
                         String unit);

  protected void putReport(String name,
                         JSONObject val,
                         String unit,
                         String descr);

  protected void putResult(File file);

  protected void putResult(File file,
                         String descr);

  protected void putResult(File[] file);

  protected void putResult(String name,
                         boolean val);

  protected void putResult(String name,
                         boolean val,
                         String unit);

  protected void putResult(String name,
                         boolean val,
                         String unit,
                         String descr);

  protected void putResult(String name,
                         double val);
  protected void putResult(String name,
                           double val,
                           String unit);

  protected void putResult(String name,
                           double val,
                           String unit,
                           String descr);

  protected void putResult(String name,
                           int val);

  protected void putResult(String name,
                           int val,
                           String unit);

  protected void putResult(String name,
                           int val,
                           String unit,
                           String descr);

  protected void putResult(String name,
                           String val);

  protected void putResult(String name,
                           String val,
                           String unit);

  protected void putResult(String name,
                           String val,
                           String unit,
                           String descr);

  protected void putResult(String name,
                           JSONObject val);

  protected void putResult(String name,
                           JSONObject val,
                           String unit);

  protected void putResult(String name,
                           JSONObject val,
                           String unit,
                           String descr);

  protected void report()
    throws Exception;

  protected void setProgress(int progress)
    throws ServiceException;

  protected void setProgress(String progress)
    throws ServiceException;

}
```

Methods inherited from java.lang.Object: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`

Author

Olaf David


Object
ModelDataService
+ EXEC_OK: String
+ EXEC_FAILED: String
+ KEY_REPORT: String
+ KEY_METAINFO: String
+ KEY_PARAMETER: String
+ KEY_RESULT: String
+ KEY_PARAMETERSETS: String
+ KEY_SUUID: String
+ KEY_STATUS: String
+ KEY_NEXT_POLL: String
+ KEY_FIRST_POLL: String
+ KEY_CPU_TIME: String
+ KEY_CLOUD_NODE: String
+ KEY_SERVICE_URL: String
+ KEY_URL: String
+ KEY_REQ_IP: String
+ KEY_KEEP_RESULTS: String
+ KEY_EXPIRATION_DATE: String
+ KEY_TSTAMP: String
+ KEY_TZ: String
+ KEY_PROGRESS: String
+ KEY_NAME: String
+ VALUE: String
+ GEOMETRY: String
+ KEY_UNIT: String
+ KEY_DESC: String
+ KEY_TIME_FILEIO: String
+ KEY_TIME_MODEL: String
+ KEY_TIME_CLIMATE_QUERY: String
+ KEY_TIME_SOIL_QUERY: String
+ KEY_TIME_LOGGING: String
+ KEY_TIME_TOTAL: String
+ KEY_REQUEST_RESULTS: String
+ FORM_PARAM: String
+ DESCR: String
+ ERROR: String
+ IN: String
+ INTENT: String
+ MAX: String
+ MIN: String
+ OUT: String
+ RANGE: String
+ UNIT: String
+ RUNNING: String
+ FINISHED: String
+ CANCELED: String
+ FAILED: String
+ UNKNOWN: String
+ SUBMITTED: String
+ SYNC: String
+ ASYNC: String
+ KEY_MODE: String
+ REPORT_FILE: String
+ REPORT_TYPE: String
+ REPORT_VALUE: String
+ REPORT_NAME: String
+ REPORT_UNITS: String
+ REPORT_DESC: String
+ REPORT_DIM0: String
+ REPORT_DIM: String
# LOG: Logger
# tz: String
+ mt: Task
+ ModelDataService()
# getRemoteAddr(): String
# getCodebase(): String
# getServicePath(): String
# getRequestURL(): String
# getRequestHost(): String
# getRequestContext(): String
# hasWorkspaceDir(): boolean
# preprocess(): void
# preProcess(): void
# createCallable(): Callable<String>
# process(): String
# postprocess(): File[]
# postProcess(): void
# createResults(): JSONArray
# createReport(): JSONArray
# report(): void
# getNextPoll(): long
# getFirstPoll(): long
# getWorkspaceDir(): File
# getSUID(): String
# getMetainfo(): JSONObject
# getParam(): JSONArray
+ setMetainfo(mi: JSONObject): void
+ setParam(parameter: JSONArray): void
+ setRequest(req: JSONObject): void
+ setParamMap(pm: Map<String, JSONObject>): void
# getParamMap(): Map<String, JSONObject>
# getRequest(): JSONObject
# getStringMetainfo(name: String): String
# getIntMetainfo(name: String): int
# getDoubleMetainfo(name: String): double
# getBooleanMetainfo(name: String): boolean
# hasMetainfo(name: String): boolean
# getMetainfoNames(): Set<String>
# getMetainfoCount(): int
# getFileInputs(): Set<File>
# getFileInputsCount(): int
# hasFileInput(name: String): boolean
# getFileInput(name: String): File
# hasParam(name: String): boolean
# getParamCount(): int
# getParamNames(): Set<String>

A.2.2. createCallable()

```
  @Deprecated protected Callable<String> createCallable()
    throws Exception;
```

Create a callable model run. The callable is returning a string result. The string is 'null' if the model execution succeeded. It is not 'null' if there is an error and the string may contain the error message.

Parameters	
return	a callable

Exceptions

`Exception`
if an error occurred during execution.

Deprecated

overwrite `csip.ModelDataService.process()` instead.

A.2.3. createReport()

```
  @Deprecated protected JSONArray createReport()
    throws Exception;
```

Create a report.

Parameters	
return	The report content as JSONArray

Exceptions

`Exception`

Deprecated

replaced by `csip.ModelDataService.report()`

A.2.4. createResults()

```
  @Deprecated protected JSONArray createResults()
    throws Exception;
```

Step 4: Create the results as JSON, (deprecated)

Parameters	
return	the results as an array of JSON objects.

Exceptions

`Exception`

Deprecated

replaced by `csip.ModelDataService.postProcess()`

A.2.5. describeJSON()

```
@GET @Produces(value="application/json") public final String describeJSON();
```

Describe the service as JSON. (Service endpoint only)

Parameters	
return	The service signature as JSON

A.2.6. execute(UriInfo, HttpServletRequest, FormDataMultiPart)

```
@POST @Produces(value="application/json") @Consumes(value="multipart/form-data") public final String execute(U
                                                                                                              H
                                                                                                              F
```

Handler for model services. Multi-part handling. (Service endpoint only)

Parameters	
uriInfo	the context info
httpReq	the servlet request
multipart	multi part input.
return	the JSON response as String.

A.2.7. execute(UriInfo, HttpServletRequest, String)

```
@POST @Produces(value="application/json") @Consumes(value="application/json") public final String execute(UriI
                                                                                                           Http
                                                                                                           Stri
```

Service Handler for non-multipart requests. There are no form parameter, everything is in the body. (Service endpoint only)

Parameters	
uriInfo	The UriInfo context
req	tye servlet request
requestStr	the request string
return	the JSON response of the service.

A.2.8. getBooleanMetainfo(String)

```
protected boolean getBooleanMetainfo(String name)
  throws ServiceException;
```

Get a metainfo value as boolean

Parameters	
name	the name of the metainfo entry
return	the metaifo value

Exceptions

`ServiceException`
General Service Exception.

A.2.9. getBooleanParam(String)

```
  protected boolean getBooleanParam(String name)
    throws ServiceException;
```

Get a boolean parameter.

Parameters	
name	the parameter name.
return	the parameter value as boolean.

Exceptions

`ServiceException`
General Service Exception.

A.2.10. getBooleanParam(String, boolean)

```
  protected boolean getBooleanParam(String name,
                                    boolean def)
    throws ServiceException;
```

Get a Boolean parameter.

Parameters	
name	the name of the parameter
def	the default value.
return	the boolean value of the parameter.

Exceptions

`ServiceException`
General Service Exception.

A.2.11. getCodebase()

```
  protected final String getCodebase();
```

Get the codebase without the model service path

Parameters	
return	the codebase of the URL as String

A.2.12. getDoubleMetainfo(String)

```
  protected double getDoubleMetainfo(String name)
    throws ServiceException;
```

Get the metainfo value as double.

Parameters	
name	the name of the metainfo entry

return	the metainfo value.

Exceptions

`ServiceException`
General Service Exception.

A.2.13. getDoubleParam(String)

```
  protected double getDoubleParam(String name)
    throws ServiceException;
```

Get an double parameter

Parameters	
name	the parameter name
return	the parameter value as double

Exceptions

`ServiceException`
General Service Exception.

A.2.14. getDoubleParam(String, double)

```
  protected double getDoubleParam(String name,
                                  double def)
    throws ServiceException;
```

Get a double parameter.

Parameters	
name	the name of the parameter
def	the default value if parameter does not exist
return	the double value of the parameter

Exceptions

`ServiceException`
General Service Exception.

A.2.15. getFileInput(String)

```
  protected File getFileInput(String name)
    throws ServiceException;
```

Get a file object for a given file name.

Parameters	
name	the file name (no path)
return	the file object with its absolute file path within the workspace. It returns null if the file is not input or does not exist.

Exceptions

`csip.ServiceException`
General Service Exception.

A.2.16. getFileInputs()

```
protected Set<File> getFileInputs();
```

Get the attached files for this request. This includes the request.

Parameters	
return	The set of files.

A.2.17. getFileInputsCount()

```
protected int getFileInputsCount();
```

Get the number of attachments. This includes the request.

Parameters	
return	the number of files attached.

A.2.18. getFirstPoll()

```
protected long getFirstPoll();
```

Get the time until first poll for async calls in milliseconds.

Parameters	
return	time to first poll in milliseconds

A.2.19. getIntMetainfo(String)

```
protected int getIntMetainfo(String name)
  throws ServiceException;
```

Get metainfo value as int.

Parameters	
name	the name of the metainfo entry
return	the int value of a metainfo entry.

Exceptions

`ServiceException`
General Service Exception.

A.2.20. getIntParam(String)

```
protected int getIntParam(String name)
  throws ServiceException;
```

Get an int parameter.

Parameters	
name	the parameter name
return	the parameter value as int

Exceptions

`ServiceException`
General Service Exception.

A.2.21. getIntParam(String, int)

```
  protected int getIntParam(String name,
                            int def)
    throws ServiceException;
```

Get a int parameter.

Parameters	
name	the name of the parameter
def	the default value if parameter does not exist
return	the int value of the parameter.

Exceptions

`ServiceException`
General Service Exception.

A.2.22. getJSONParam(String)

```
  protected JSONObject getJSONParam(String name)
    throws ServiceException;
```

Get a JSONObject parameter.

Parameters	
name	the parameter name.
return	the parameter value as JSONObject.

Exceptions

`ServiceException`
General Service Exception.

A.2.23. getJSONParam(String, JSONObject)

```
  protected JSONObject getJSONParam(String name,
                                    JSONObject def)
    throws ServiceException;
```

Get a Long parameter.

Parameters

name	the name of the parameter
def	the default value if parameter does not exist
return	the JSONObject of the parameter

Exceptions

`ServiceException`
General Service Exception.

A.2.24. getLongParam(String)

```
  protected long getLongParam(String name)
    throws ServiceException;
```

Get a long parameter.

Parameters	
name	the parameter name.
return	the parameter value as long.

Exceptions

`ServiceException`
General Service Exception.

A.2.25. getLongParam(String, long)

```
  protected long getLongParam(String name,
                              long def)
    throws ServiceException;
```

Get a Long parameter.

Parameters	
name	the name of the parameter
def	the default value if parameter does not exist
return	the long value of the parameter

Exceptions

`ServiceException`
General Service Exception.

A.2.26. getMetainfo()

```
  protected final JSONObject getMetainfo();
```

Get the request metainfo.

Parameters	
return	the metainfo

A.2.27. getMetainfoCount()

```
protected int getMetainfoCount();
```

Get the number of metainfo entries.

Parameters	
return	the number of entries.

A.2.28. getMetainfoNames()

```
protected Set<String> getMetainfoNames();
```

Get all metainfo names.

Parameters	
return	the set of metainfo names.

A.2.29. getNextPoll()

```
protected long getNextPoll();
```

Return the recommended polling interval for async calls in milliseconds.

Parameters	
return	the polling interval value in milliseconds

A.2.30. getParam()

```
protected final JSONArray getParam();
```

Get the request parameter

Parameters	
return	request parameter

A.2.31. getParamCount()

```
protected int getParamCount();
```

Get the number of parameter.

Parameters	
return	the number of parameter

A.2.32. getParamDescr(String)

```
protected String getParamDescr(String name)
  throws ServiceException;
```

Get the description of a parameter.

Parameters

name	the parameter name
return	the description as string, 'null' if there is none.

Exceptions

`ServiceException`
General Service Exception.

A.2.33. getParamGeometry(String)

```
protected JSONObject getParamGeometry(String name)
  throws ServiceException;
```

Get the geometry of a parameter

Parameters	
name	the name if the parameter
return	the geometry of a parameter

Exceptions

`ServiceException`
General Service Exception.

A.2.34. getParamMap()

```
protected final Map<String, JSONObject> getParamMap();
```

Get the Parameter as map "name" -> JSONObject

Parameters	
return	the parameter map

A.2.35. getParamNames()

```
protected Set<String> getParamNames();
```

Get all parameter names.

Parameters	
return	the set of names.

A.2.36. getParamUnit(String)

```
protected String getParamUnit(String name)
  throws ServiceException;
```

Get the unit of a parameter.

Parameters	
name	the parameter name

return	the unit as string, 'null' if there is none.

Exceptions

`ServiceException`
General Service Exception.

A.2.37. getRemoteAddr()

```
protected final String getRemoteAddr();
```

The request ip

Parameters	
return	the request ip

A.2.38. getRequest()

```
protected final JSONObject getRequest();
```

Get the original JSOn request object.

Parameters	
return	the request object.

A.2.39. getRequestContext()

```
protected final String getRequestContext();
```

Get the webapp context name.

Parameters	
return	the context name

A.2.40. getRequestHost()

```
protected final String getRequestHost();
```

Get the complete request URL

Parameters	
return	the full request URL

A.2.41. getRequestURL()

```
protected final String getRequestURL();
```

Get the complete request URL

Parameters	
return	the full request URL

A.2.42. getResourceExe(String)

```
  protected Executable getResourceExe(String id)
    throws ServiceException;
```

Get an service executable from a resource definition. Resources are defined as service annotations.

Parameters	
id	the id of the resource
return	the ProcessExecution for that executable

Exceptions

`ServiceException`
General Service Exception.

See Also
`csip.annotations.Resource`

A.2.43. getResourceFile(String)

```
  protected File getResourceFile(String id)
    throws ServiceException;
```

Get a service resource file. Resources are defined as service annotations.

Parameters	
id	the id of the resource.
return	the extracted file within the local file system.

Exceptions

`ServiceException`
General Service Exception.

See Also
`csip.annotations.Resource`

A.2.44. getServicePath()

```
  protected final String getServicePath();
```

Provide the service path name, e.g. 'weps/1.0'

Parameters	
return	the model service path

A.2.45. getStringMetainfo(String)

```
  protected String getStringMetainfo(String name)
    throws ServiceException;
```

Get the metainfo value as String.

Parameters	
name	the name of the metainfo entry
return	the value of a metainfo entry

Exceptions

`ServiceException`
General Service Exception.

A.2.46. getStringParam(String)

```
  protected String getStringParam(String name)
    throws ServiceException;
```

Get a String parameter

Parameters	
name	the parameter name
return	the parameter value as String

Exceptions

`ServiceException`
General Service Exception.

A.2.47. getStringParam(String, String)

```
  protected String getStringParam(String name,
                                  String def)
    throws ServiceException;
```

Get a String parameter.

Parameters	
name	the name of the parameter
def	the default value if the parameter is missing
return	the value of the parameter

Exceptions

`ServiceException`
General Service Exception.

A.2.48. getSUID()

```
  protected final String getSUID();
```

Get the simulation unique identifier (128 byte UUID)

Parameters	
return	the suid string

A.2.49. getWorkspaceDir()

```
  protected final File getWorkspaceDir();
```

Get a new Workspace folder for this model run. returns null if it has no simulation folder.

Parameters	
return	the workspace or null if there should be none.

A.2.50. hasFileInput(String)

```
  protected boolean hasFileInput(String name)
    throws ServiceException;
```

Check is a input file exists in the workspace.

Parameters	
name	the file name
return	true if the file exist, false otherwise

Exceptions

`ServiceException`
General Service Exception.

A.2.51. hasMetainfo(String)

```
  protected boolean hasMetainfo(String name);
```

Check if a metainfo entry exists.

Parameters	
name	the name of a metainfo name.
return	true if a metainfo entry exists, false otherwise

A.2.52. hasParam(String)

```
  protected boolean hasParam(String name);
```

Check if a parameter exists.

Parameters	
name	the name of the parameter entry
return	true if present false otherwise.

A.2.53. hasWorkspaceDir()

```
  protected final boolean hasWorkspaceDir();
```

Indicate if the service needs a workspace folder (as sandbox)

Parameters

return	true if it is needed, false otherwise.

A.2.54. postprocess()

```
@Deprecated protected File[] postprocess()
  throws Exception;
```

workflow step 3: Postprocess the data.

Parameters	
return	The filenames that should be transferred into the results folder

Exceptions

`Exception`

Deprecated

overwrite `csip.ModelDataService.postProcess()` instead.

A.2.55. postProcess()

```
protected void postProcess()
  throws Exception;
```

workflow step 3: create the response the data.

Exceptions

`Exception`

A.2.56. preprocess()

```
@Deprecated protected void preprocess()
  throws Exception;
```

workflow step 1: Preprocess the data.

Exceptions

`Exception`

Deprecated

replaced by `csip.ModelDataService.postProcess()`

A.2.57. preProcess()

```
protected void preProcess()
  throws Exception;
```

workflow step 1: process the request data.

Exceptions

`Exception`

A.2.58. process()

```
  protected String process()
    throws Exception;
```

Process logic of the service.

Parameters	
return	null if the process ended successfully, the error message otherwise.

Exceptions

`Exception`

A.2.59. putReport(File...)

```
  protected void putReport(File[] file);
```

Put Files into a report.

Parameters	
file	the files to report

A.2.60. putReport(File)

```
  protected void putReport(File file);
```

Put a File into a report.

Parameters	
file	the file to report

A.2.61. putReport(File, String)

```
  protected void putReport(File file,
                           String descr);
```

Put a File into a report.

Parameters	
file	the file
descr	a description

A.2.62. putReport(String, boolean)

```
  protected void putReport(String name,
                           boolean val);
```

Put a boolean value into a report.

Parameters	
name	the result name

val	the value to store

A.2.63. putReport(String, boolean, String)

```
protected void putReport(String name,
                         boolean val,
                         String unit);
```

Put a boolean value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.64. putReport(String, boolean, String, String)

```
protected void putReport(String name,
                         boolean val,
                         String unit,
                         String descr);
```

Put a boolean value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.65. putReport(String, double)

```
protected void putReport(String name,
                         double val);
```

Put a double value into a report.

Parameters	
name	the result name
val	the value to store

A.2.66. putReport(String, double, String)

```
protected void putReport(String name,
                         double val,
                         String unit);
```

Put a double value into a report.

Parameters	
name	the result name
val	the value to store

unit	the physical unit

A.2.67. putReport(String, double, String, String)

```
protected void putReport(String name,
                         double val,
                         String unit,
                         String descr);
```

Put a double value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.68. putReport(String, int)

```
protected void putReport(String name,
                         int val);
```

Put a int value into a report.

Parameters	
name	the result name
val	the value to store

A.2.69. putReport(String, int, String)

```
protected void putReport(String name,
                         int val,
                         String unit);
```

Put a int value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.70. putReport(String, int, String, String)

```
protected void putReport(String name,
                         int val,
                         String unit,
                         String descr);
```

Put a int value into a report.

Parameters	
name	the result name

val	the value to store
unit	the physical unit
descr	a description

A.2.71. putReport(String, JSONObject)

```
protected void putReport(String name,
                         JSONObject val);
```

Put a JSONObject value into a report.

Parameters	
name	the result name
val	the value to store

A.2.72. putReport(String, JSONObject, String)

```
protected void putReport(String name,
                         JSONObject val,
                         String unit);
```

Put a JSONObject value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.73. putReport(String, JSONObject, String, String)

```
protected void putReport(String name,
                         JSONObject val,
                         String unit,
                         String descr);
```

Put a JSONObject value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.74. putReport(String, String)

```
protected void putReport(String name,
                         String val);
```

Put a String value into a report.

Parameters	

name	the result name
val	the value to store

A.2.75. putReport(String, String, String)

```
protected void putReport(String name,
                         String val,
                         String unit);
```

Put a String value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.76. putReport(String, String, String, String)

```
protected void putReport(String name,
                         String val,
                         String unit,
                         String descr);
```

Put a String value into a report.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.77. putResult(File...)

```
protected void putResult(File[] file);
```

Provide multiple files as a result.

Parameters	
file	the files to add as a result

A.2.78. putResult(File)

```
protected void putResult(File file);
```

Provide a file as a result.

Parameters	
file	the file result

A.2.79. putResult(File, String)

```
protected void putResult(File file,
                         String descr);
```

Provide a file with description as a result.

Parameters	
file	
descr	

A.2.80. putResult(String, boolean)

```
protected void putResult(String name,
                         boolean val);
```

Provide a boolean as a result.

Parameters	
name	the result name
val	the value to store

A.2.81. putResult(String, boolean, String)

```
protected void putResult(String name,
                         boolean val,
                         String unit);
```

Provide a boolean as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.82. putResult(String, boolean, String, String)

```
protected void putResult(String name,
                         boolean val,
                         String unit,
                         String descr);
```

Provide a boolean as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.83. putResult(String, double)

```
protected void putResult(String name,
                         double val);
```

Provide a double as a result.

Parameters	
name	the result name
val	the value to store

A.2.84. putResult(String, double, String)

```
  protected void putResult(String name,
                         double val,
                         String unit);
```

Provide a double as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.85. putResult(String, double, String, String)

```
  protected void putResult(String name,
                         double val,
                         String unit,
                         String descr);
```

Provide a double as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.86. putResult(String, int)

```
  protected void putResult(String name,
                         int val);
```

Provide an int as a result.

Parameters	
name	the result name
val	the value to store

A.2.87. putResult(String, int, String)

```
  protected void putResult(String name,
                         int val,
                              String unit);
```

Provide an int as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.88. putResult(String, int, String, String)

```
  protected void putResult(String name,
                           int val,
                           String unit,
                           String descr);
```

Provide an int as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.89. putResult(String, JSONObject)

```
  protected void putResult(String name,
                           JSONObject val);
```

Provide a JSONObject as a result.

Parameters	
name	the result name
val	the value to store

A.2.90. putResult(String, JSONObject, String)

```
  protected void putResult(String name,
                           JSONObject val,
                           String unit);
```

Provide a JSONObject as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.91. putResult(String, JSONObject, String, String)

```
  protected void putResult(String name,
                              JSONObject val,
                              String unit,
                              String descr);
```

Provide a JSONObject as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.92. putResult(String, String)

```
  protected void putResult(String name,
                           String val);
```

Provide a string as a result.

Parameters	
name	the result name
val	the value to store

A.2.93. putResult(String, String, String)

```
  protected void putResult(String name,
                           String val,
                           String unit);
```

Provide a string as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit

A.2.94. putResult(String, String, String, String)

```
  protected void putResult(String name,
                           String val,
                           String unit,
                           String descr);
```

Provide a string as a result.

Parameters	
name	the result name
val	the value to store
unit	the physical unit
descr	a description

A.2.95. report()

```
protected void report()
  throws Exception;
```

Create a report.

Exceptions

`Exception`

A.2.96. setMetainfo(JSONObject)

```
public final void setMetainfo(JSONObject mi);
```

Set the request metainfo.

Deprecated

Deprecated

A.2.97. setParam(JSONArray)

```
public void setParam(JSONArray parameter);
```

Set the request parameter.

Deprecated

Deprecated

A.2.98. setParamMap(Map<String, JSONObject>)

```
public final void setParamMap(Map<String, JSONObject> pm);
```

Set the Parameter map

Deprecated

Deprecated

A.2.99. setProgress(int)

```
protected void setProgress(int progress)
  throws ServiceException;
```

Set the progress as a numerical value (0.-.100)

Parameters	
progress	a value between 0 and 100;

Exceptions

`ServiceException`
General Service Exception.

A.2.100. setProgress(String)

```
  protected void setProgress(String progress)
    throws ServiceException;
```

Set the progress as a string message. Call this message during process() to indicate progress for long running models. If the service is called asynchronously the message will be reported in the metainfo part of the rest call as the 'progress' entry.

Parameters	
progress	a meaningful message

Exceptions

`ServiceException`
General Service Exception.

A.2.101. setRequest(JSONObject)

```
  public final void setRequest(JSONObject req);
```

Set the request

Deprecated

Deprecated

A.3. ModelDataService.Task

Model execution Task.

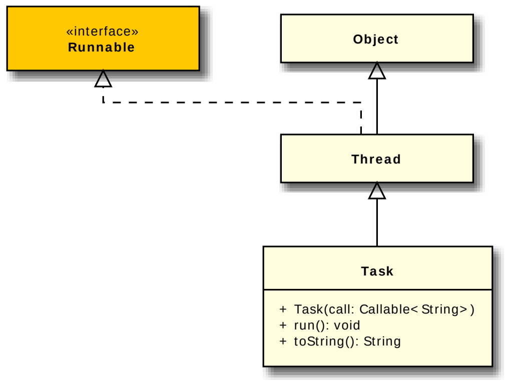
A.3.1. Synopsis

```
 public final class ModelDataService.Task extends, Thread {
// Public Constructors

  public Task(Callable<String> call);

// Public Methods

  public void run();

  public String toString();

}
```

Methods inherited from java.lang.Thread: `activeCount`, `checkAccess`, `clone`, `countStackFrames`, `currentThread`, `destroy`, `dumpStack`, `enumerate`, `getAllStackTraces`, `getContextClassLoader`, `getDefaultUncaughtExceptionHandler`, `getId`, `getName`, `getPriority`, `getStackTrace`, `getState`, `getThreadGroup`, `getUncaughtExceptionHandler`, `holdsLock`, `interrupt`, `interrupted`, `isAlive`, `isDaemon`, `isInterrupted`, `join`, `resume`, `run`, `setContextClassLoader`, `setDaemon`, `setDefaultUncaughtExceptionHandler`, `setName`, `setPriority`, `setUncaughtExceptionHandler`, `sleep`, `start`, `stop`, `suspend`, `toString`, `yield`

Methods inherited from java.lang.Object: `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`

Fields inherited from java.lang.Thread: `MAX_PRIORITY`, `MIN_PRIORITY`, `NORM_PRIORITY`



A.4. ServiceException

1. Resource

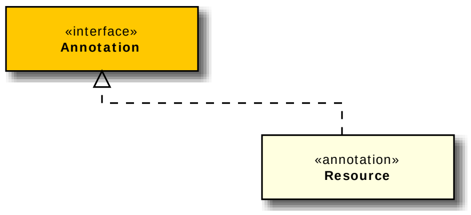
Resource definition.

1.1. Synopsis

```
 @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) @Target(value=java.lang.annotation.ElementType.T

  public String file ;

  public ResourceType type ;

  public String id ;

  public boolean wine ;

  public String args ;

  public String env ;

}
```

Author

od



1.2. args

Default executable arguments, separated by space

Parameters

return	the executable arguments

1.3. env

Default environment variables to be used for execution. ("env1=abc env2=def"}

Parameters	
return	the environment variables for execution.

1.4. file

The path to the file within the war file or file system.

Parameters	
return	the relative path to the file in the war or on the absolute path in the file system.

1.5. id

The id of that resource. Set it only if you want to access the resource.

Parameters	
return	the id of that resource

See Also

csip.AbstractModelService#getResourceExe(java.lang.String)

,

csip.AbstractModelService#getResourceFile(java.lang.String)

1.6. type

The type of the resource. F

Parameters	
return	the specific type of the resource.

1.7. wine

Should the file executed via wine.

Parameters	
return	true if executed via wine, false otherwise.

General Service Exception.

A.4.1. Synopsis

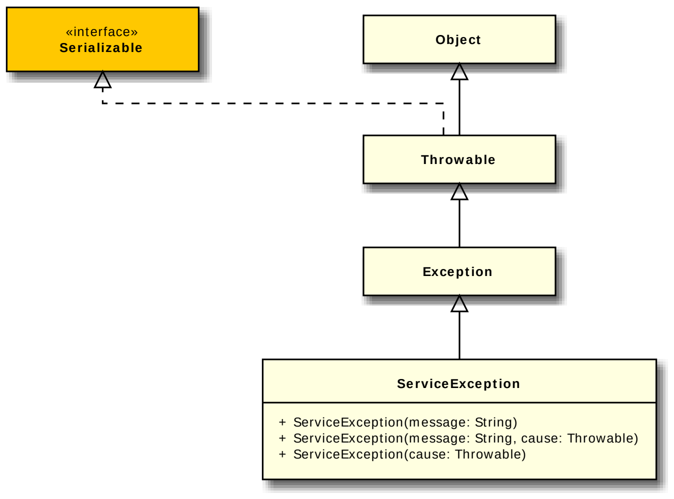
```
 public class ServiceException extends, Exception {
// Public Constructors

  public ServiceException(String message);
  public ServiceException(String message,
                          Throwable cause);

  public ServiceException(Throwable cause);

}
```

Methods inherited from java.lang.Throwable: `addSuppressed` , `fillInStackTrace` , `getCause` , `getLocalizedMessage` , `getMessage` , `getStackTrace` , `getSuppressed` , `initCause` , `printStackTrace` , `setStackTrace` , `toString`

Methods inherited from java.lang.Object: `clone` , `equals` , `finalize` , `getClass` , `hashCode` , `notify` , `notifyAll` , `wait`

Author

Olaf David


«interface»
Serializable
Object
Throwable
Exception
ServiceException
+ ServiceException(message: String)
+ ServiceException(message: String, cause: Throwable)
+ ServiceException(cause: Throwable)

A.4.2. ServiceException(String)

```
  public ServiceException(String message);
```

Exception Constructor

Parameters	
message	the exception message

A.4.3. ServiceException(String, Throwable)

```
  public ServiceException(String message,
                          Throwable cause);
```

Exception Constructor

Parameters	
message	
cause	

A.4.4. ServiceException(Throwable)

```
public ServiceException(Throwable cause);
```

exception Constructor.

Parameters	
cause	

A.5. Resource

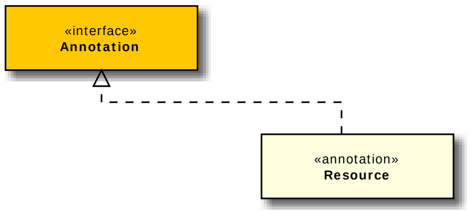
Resource definition.

A.5.1. Synopsis

```
 @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) @Target(value=java.lang.annotation.Elemen
  public String file ;
  public ResourceType type ;
  public String id ;
  public boolean wine ;
  public String args ;
  public String env ;
}
```

Author

od



A.5.2. args

Default executable arguments, separated by space

Parameters	
return	the executable arguments

A.5.3. env

Default environment variables to be used for execution. ("env1=abc env2=def"}

Parameters	

return	the environment variables for execution.

A.5.4. file

The path to the file within the war file or file system.

Parameters	
return	the relative path to the file in the war or on the absolute path in the file system.

A.5.5. id

The id of that resource. Set it only if you want to access the resource.

Parameters	
return	the id of that resource

See Also

csip.AbstractModelService#getResourceExe(java.lang.String)

,

csip.AbstractModelService#getResourceFile(java.lang.String)

A.5.6. type

The type of the resource. F

Parameters	
return	the specific type of the resource.

A.5.7. wine

Should the file executed via wine.

Parameters	
return	true if executed via wine, false otherwise.

A.6. ResourceType

Resource types.

A.6.1. Synopsis

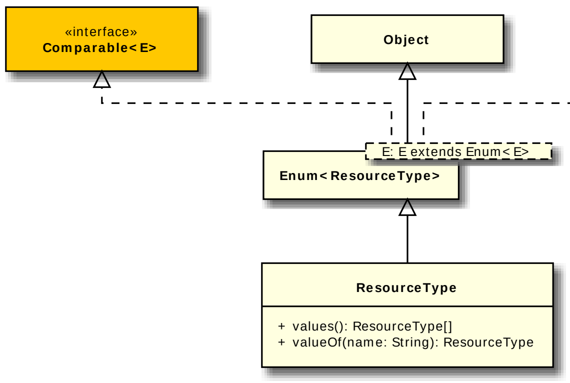
```
 public final class ResourceType extends, Enum<ResourceType> {
// Public Static Fields

  public static final ResourceType ARCHIVE ;

  public static final ResourceType CLASSNAME ;

  public static final ResourceType EXECUTABLE ;

  public static final ResourceType FILE ;
  public static final ResourceType JAR ;

  public static final ResourceType OMS_DSL ;

  public static final ResourceType OUTPUT ;

  public static final ResourceType REFERENCE ;

// Public Static Methods

  public static ResourceType valueOf(String name);

  public static ResourceType[] values();

}
```

Methods inherited from java.lang.Enum: `clone`, `compareTo`, `equals`, `finalize`, `getDeclaringClass`, `hashCode`, `name`, `ordinal`, `toString`, `valueOf`

Methods inherited from java.lang.Object: `getClass`, `notify`, `notifyAll`, `wait`

Author

od



A.6.2. ARCHIVE

```
  public static final ResourceType ARCHIVE ;
```

This resource is a zip file and it should be unpacked.

A.6.3. CLASSNAME

```
  public static final ResourceType CLASSNAME ;
```

This is a file is a java class name.

A.6.4. EXECUTABLE

```
  public static final ResourceType EXECUTABLE ;
```

This resource is a executable.

A.6.5. FILE

```
  public static final ResourceType FILE ;
```

This resource is a file.

A.6.6. JAR

```
  public static final ResourceType JAR ;
```

This is a file is a jar file.

A.6.7. OMS_DSL

```
  public static final ResourceType OMS_DSL ;
```

This is a file is a OMS DSL file.

A.6.8. OUTPUT

```
  public static final ResourceType OUTPUT ;
```

The resource describes output of the model service.

A.6.9. REFERENCE

```
  public static final ResourceType REFERENCE ;
```

This is a file reference to an existing file. Nothing should be extracted.

A.7. Resources

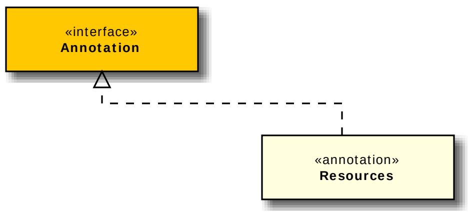
Describe the resources bundled with the service.

A.7.1. Synopsis

```
 @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) @Target(value=java.lang.annotation.ElementType.T

  public Resource[] value ;

}
```

Author

od



A.7.2. value

List of resources.

Parameters	
return	the array of resources.

Appendix B. ModelServices Constant field values

B.1. csip.*

Table B.1. ModelDataService

ASYNC	"async"
CANCELED	"Canceled"
DESCR	"descr"
ERROR	"error"
EXEC_FAILED	"Error"
FAILED	"Failed"
FINISHED	"Finished"
FORM_PARAM	"param"
GEOMETRY	"geometry"
IN	"in"
INTENT	"intent"
KEY_CLOUD_NODE	"cloud_node"
KEY_CPU_TIME	"cpu_time"
KEY_DESC	"description"
KEY_EXPIRATION_DATE	"expiration_date"
KEY_FIRST_POLL	"first_poll"
KEY_KEEP_RESULTS	"keep_results"
KEY_METAINFO	"metainfo"
KEY_MODE	"mode"
KEY_NAME	"name"
KEY_NEXT_POLL	"next_poll"
KEY_PARAMETER	"parameter"
KEY_PARAMETERSETS	"parametersets"
KEY_PROGRESS	"progress"
KEY_REPORT	"report"
KEY_REQUEST_RESULTS	"request-results"
KEY_REQ_IP	"request_ip"
KEY_RESULT	"result"
KEY_SERVICE_URL	"service_url"
KEY_STATUS	"status"
KEY_SUUID	"suid"
KEY_TIME_CLIMATE_QUERY	"timeClimateQuery"

KEY_TIME_FILEIO	"timeFileIO"
KEY_TIME_LOGGING	"timeLogging"
KEY_TIME_MODEL	"timeModel"
KEY_TIME_SOIL_QUERY	"timeSoilQuery"
KEY_TIME_TOTAL	"timeTotal"
KEY_TSTAMP	"tstamp"
KEY_TZ	"tz"
KEY_UNIT	"unit"
KEY_URL	"url"
MAX	"max"
MIN	"min"
OUT	"out"
RANGE	"range"
REPORT_DESC	"description"
REPORT_DIM	"dim"
REPORT_DIM0	"dimension0"
REPORT_FILE	"report.json"
REPORT_NAME	"name"
REPORT_TYPE	"type"
REPORT_UNITS	"units"
REPORT_VALUE	"value"
RUNNING	"Running"
SUBMITTED	"Submitted"
SYNC	"sync"
UNIT	"unit"
UNKNOWN	"Unknown"
VALUE	"value"

Bibliography

[Argent2004] Argent, R.M. An overview of model integration for environmental applications—components, frameworks and semantics Environmental Modelling & Software, Volume 19, Issue 3, Pages 219-234, Mar 2004

[David2002] David O., S.L. Markstrom, K.W. Rojas, L.R. Ahuja and I.W. Schneider, The Object Modeling System. In: L.R. Ahuja, L. Ma and T.A. Howell, Editors, Agricultural System Models in Field Research and Technology Transfer, Lewis Publishers, Boca Raton (2002), pp. 317–330.

[David2014] David, O., Lloyd, W., Rojas, K., Arabi, M., Geter, F., Ascough II, J., Green, T., Leavesley, G. and J. Carlson (2014), Model-as-a-Service (MaaS) using the Cloud Services Innovation Platform (CSIP), In: Ames, D.P., Quinn, N.W.T., Rizzoli, A.E. (Eds.), Proceedings of the 7th International Congress on Environmental Modelling and Software, June 15-19, San Diego, California, USA. ISBN: 978-88-9035-744-2

[Fielding2002] Fielding, R.; Taylor, R., 2002, "Principled Design of the Modern Web Architecture", ACM Transactions on Internet Technology (TOIT) (New York: Association for Computing Machinery) 2 (2): 115–150.

[GeoServices2010] GeoServices REST Specification Version 1.0, ESRI White Paper, 380 New York Street Redlands, California 92373-8100, September 2010

[ISO8601] ISO 8601:2004, Data elements and interchange formats -- Information interchange -- Representation of dates and times. http://www.iso.org/iso/catalogue_detail?csnumber=40874

[Jha2011] Jha, S, D. S. Katz, A. Luckow, A. Merzky, and K. Stamou, 2011, Understanding Scientific Applications for Cloud Environments, in Cloud Computing: Principles and Paradigms, R. Buyya, J. Broberg, and A. Goscinski, Eds. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2011, pp. 345-371

[OGC2007] Open Geospatial Consortium, Inc. OpenGIS® Web Processing Service (WPS) Specification. WWW document, http://portal.opengeospatial.org/files/?artifact_id=24151, 2007.

[RFC4122] RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace, http://www.ietf.org/rfc/rfc4122.txt

[Roman2009] Roman, D; Schade S.; Berre, A. J.; Bodsberg, N. R.; Langlois, J., 2009, Model as a Service (MaaS). In Grid Technologies for Geospatial Applications Workshop, Hannover, Germany, 2009.

[Zou2012] Zou G., Zhang B., Zheng J., Li Y.; Ma J., 2012, MaaS: Model as a Service in Cloud Computing and Cyber-I Space, Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on , vol., no., pp.1125,1130, 27-29 Oct. 2012.

Index