DO MORE .

TYPEFI®

TECHNICAL DOCUMENTATION:

Typefi 8 Plugin SDK

Automation for print, online and mobile

release: 1.0.0

Australia . USA . UK . Netherlands . typefi.com

Typefi Systems Pty Ltd
Suite 1 / 61–63 Primary School Ct
Maroochydore QLD 4558
+61 7 3102 5444
www.typefi.com

Release: 1.0.0

Contents

Typefi 8 Plugin SDK

Introduction

The Typefi 8 Plugin SDK contains documentation and some code to teach you how to extend Typefi 8 through new workflow actions.

A *workflow* is made up of one or more actions that perform a sequence of tasks used to automatically generate output. A workflow can be as simple as a single action performing just one task or it can contain multiple actions that perform a series of tasks. When you run a workflow, the actions are performed in sequence from top to bottom.

An *action* performs a single task. Add multiple actions to a workflow to perform a series of tasks. Actions may receive input from any previous action. Some actions require specific types of input while others are used for generic functions. Actions are packaged inside plugins.

A *plugin* is a Java (Apache Tomcat) webapp that contains one or more related actions. Plugins are self-contained webapps that provide:

1. Complete isolation for code and dependencies between plugins; two different plugins can link to different versions of the same jar file.
2. An efficient way to eliminate redundancy and ship common dependencies.

Tutorial

Overview

We will create a plugin called "Markdown" containing a single action *Create Html Document* which creates an Html document from a Markdown document.

Getting started

Software installation

Download and install

- Eclipse for Java EE Developers
- Gradle Buildship Eclipse plugin
- Apache Tomcat 8

Eclipse project setup

Create a project folder named `typefi-plugin-markdown`. Note that the folder name, which is converted by Gradle to the context root of your deployed webapp, must start with the prefix `typefi-plugin-`. This enables Typefi 8 to automatically detect your plugin when running in the same Tomcat instance. The project will use the Gradle conventions of storing Java files in `src/main/java` and the webapp in `src/main/webapp`.

Next, create the following `gradle.build` file within the project folder. This can be used to build the plugin war file, along with eclipse project files. It fails on version conflict; otherwise, where dependencies include multiple versions of an external jar all will automatically be upgraded to the

latest required version, which may introduce errors. The Java and Gradle versions are set to 1.8 and 2.14 respectively.

```
apply plugin: 'war'
apply plugin: 'eclipse'
apply plugin: 'eclipse-wtp'
configurations.all {
    resolutionStrategy {
        failOnVersionConflict()
    }
}
sourceCompatibility = JavaVersion.VERSION_1_8
targetCompatibility = JavaVersion.VERSION_1_8
task wrapper(type: Wrapper) {
    gradleVersion = '2.14'
}
```

Dependencies upon `common-server-sdk-1.0.0.jar` and `txtmark-0.13.jar` should be appeneded to `gradle.build`. Internal developers will have access to the Typefi Artifactory repository and should therefore append the following to the `gradle.build` file:

```
repositories {
        maven {
            url 'http://artifactory.typefi.com:8081/artifactory/jcenter'
        }
        maven {
            url 'http://artifactory.typefi.com:8081/artifactory/list/libs-
release-local/'
        }
}
dependencies {
    compile 'com.typefi:common-server-sdk:1.0.0'
    compile 'com.github.rjeschke:txtmark:0.13'
}
```

External developers should download `common-server-sdk-1.0.0.jar` from typefi.com to a `libs` folder within the project and append the following to to the `gradle.build` file:

```
repositories {
        jcenter()
}
dependencies {
        compile 'com.github.rjeschke:txtmark:0.13'
        compile 'com.google.code.gson:gson:2.6.2'
        compile 'javax.servlet:javax.servlet-api:3.1.0'
        compile fileTree(dir: 'libs')
}
```

Next, open Eclipse and import the created project as a Gradle Project. Then open the Gradle Tasks view within Eclipse and run the eclipseWtp task for the project. Finally, right click on the project within Gradle and select `Gradle...Refresh Gradle Project`.

Plugin descriptor

Every plugin must implement the following endpoint:

```
GET /descriptor
```

This endpoint must return JSON that describes all the actions (and their UIs) offered by the plugin and conforms to the Plugin Schema

For the markdown plugin the following file is created in the project:

`src/main/webapp/WEB-INF/conf/descriptor.json`

It contains the following JSON

```
{
    "displayname": "Markdown",
    "description": "Markdown plugin",
    "vendor": "Typefi Systems Pty Ltd",
    "actions": [
        {
            "type": "md-to-html",
            "uiVersion": 1,
            "displayname": "Create HTML Document",
            "description": "Create HTML Document",
            "inputs": [
                {
                    "name": "input",
                    "label": "Input",
                    "type": "file",
                    "extension": "md",
                    "value": "",
                    "required": true
                    },
                {
                    "name": "output",
                    "label": "Output",
                    "type": "file",
                    "extension": "html",
                    "value": "",
                    "output": true,
                    "required": true
                    }
                ]
        }
    ]
}
```

For the descriptor servlet the following file is created in the project:

`src/main/java/com/typefi/actions/markdown/servlets/DescriptorServlet.java`

It contains the following Java, which simply uses the `Descriptor.printDescriptor` method and `Annotations.DESCRIPTOR` from `common-server-sdk` to return the contents of the descriptor file:

```
package com.typefi.actions.markdown.servlets;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.typefi.sdk.plugin.Annotations;
import com.typefi.sdk.plugin.Descriptor;
@WebServlet(Annotations.DESCRIPTOR)
public class DescriptorServlet extends HttpServlet {
    @Override
    protected void doGet(final HttpServletRequest request, final
HttpServletResponse response)
            throws ServletException, IOException {
        Descriptor.printDescriptor(request, response);
    }
}
```

When this servlet is deployed on Apache Tomcat then visiting the URL `http://localhost:8080/typefi-plugin-markdown/descriptor` in a browser will display the contents of `descriptor.json`.

Plugin signature

The following endpoint must be implemented by every plugin:

```
GET /signature
```

This endpoint must return the signature corresponding to the plugin descriptor, encoded as a UTF-8 string using base 64 encoding. When a plugin is licensed by Typefi a `descriptor.typefi_signature` file is generated from the `descriptor.json` file.

For the markdown plugin, once the plugin has been licensed by Typefi the signature file returned by Typefi is copied into in the project:

```
src/main/webapp/WEB-INF/conf/descriptor.typefi_signature
```

For the signature servlet the following file is created in the project:

```
src/main/java/com/typefi/actions/markdown/servlets/SignatureServlet.java
```

The servlet uses the `Signature.printSignature` method method and `Annotations.SIGNATURE` from `common-server-sdk` to return the contents of the signature file as a base 64 encoded UTF-8 string:

```
package com.typefi.actions.markdown.servlets;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.typefi.sdk.plugin.Annotations;
import com.typefi.sdk.plugin.Signature;
@WebServlet(Annotations.SIGNATURE)
public class SignatureServlet extends HttpServlet {
    @Override
    protected void doGet(final HttpServletRequest request, final
HttpServletResponse response)
            throws ServletException, IOException {
        Signature.printSignature(request, response);
    }
}
```

Icon

A plugin may optionally implement the following endpoint:

```
GET /icon
```

This endpoint must return a Scalable Vector Graphics (SVG) icon for the plugin. Note that the icon must have its `fill` attribute set to `currentColor` to ensure that it correctly inherits the colours used througout Server.

For the markdown plugin the following file is created in the project:

```
src/main/webapp/WEB-INF/conf/icon/action-markdown.svg
```

The file contains the following icon data:

```
<svg id="Layer_1" data-name="Layer 1"
xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 48 48">
<title>action-markdown</title>
```

```
<path d="M24,4.1a24,24,0,1,0,0,24,24A23.94,23.94,0,0,0,0,24,
4.1Zm3.2,32.6H21.4V28.1l-4.3,5.5-4.3-5.5v8.6H7V19.4h5.8l4.3,
5.8,4.3-5.8h5.8V36.7Zm8.6,1.4h0L28.6,
28h4.3V19.4h5.8V28H43L35.8,38.1h0Z"
transform="translate(0 -4.1)"
fill="currentColor"/>
</svg>
```

The icon servlet is created as the following file within the project:

`src/main/java/com/typefi/actions/markdown/servlets/IconServlet.java`

The servlet uses the `Icon.printIcon` method and `Annotations.ICON` from `common-server-sdk` to return the icon file:

```
package com.typefi.actions.markdown.servlets;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.typefi.sdk.plugin.Annotations;
import com.typefi.sdk.plugin.Icon;
@WebServlet(Annotations.ICON)
public class IconServlet extends HttpServlet {
    @Override
    protected void doGet(final HttpServletRequest request, final
HttpServletResponse response)
            throws ServletException, IOException {
        Icon.printIcon(request, response);
    }
}
```

Action

Create Html Document Action

The endpoint name is the `type` value as specified in the descriptor JSON above plus the string `/run`, e.g.

```
POST /md-to-html/run
```

A workflow can be created on the server containing the action. When the server runs the action it will `POST` to `/md-to-html/run` and pass in two parameters as JSON. The first, `env`, is a `GsonEnvironment` object:

```
{
  "jobId": 2147480214,
  "jobFolder": "/Md/2016-07-11 14.16.41",
  "systemType": "SERVER",
  "locale": "en-US",
  "filestorePath":
  "/users/jbloggs/Documents/Typefi",
  "groups": [
    "Editors"
  ],
  "versioningEnabled": true,
  "user": "Joe Bloggs",
  "actionId": "2147480214-1",
  "actionLogFile":
  "/Md/2016-07-11 14.16.41/logs
  /01 Create HTML Document.log",
  "workflowLogFile":
  "/Md/2016-07-11 14.16.41/workflow.log",
```

```
  "serverLogFile":
  "/Library/Application Support/Typefi/tpss.log",
  "engineVersion": "",
  "host": "",
  "port": "",
  "inDesignFilestorePath": "",
  "tpsFilestoreLocation": ""
}
```

The second parameter, `action`, is an `Action` object

```
{
  "type": "md-to-html",
  "uiVersion": 1,
  "plugin": "typefi-plugin-markdown",
  "displayname": "Create HTML Document",
  "description": "Create HTML Document",
  "pluginversion": "?",
  "id": "1",
  "inputs": [
    {
      "name": "input",
      "value": "/Md/sdk.md"
    },
    {
      "name": "output",
      "value":
      "/Md/2016-07-11 14.16.41/output-1.html"
    }
  ],
  "disabled": false
}
```

To implement an action, a class must be created which extends the abstract `com.typefi.sdk.action.ActionRun` class and overrides its `doAction` method. The class automatically logs the start and end of the action and returns an `ActionStatus` object containing the status of the action. It also stores the above JSON parameters as properties.

The following is created within the project to override the `ActionRun` class:

`src/main/java/com/typefi/actions/markdown/run/MdToHtmlRun.java`

It implements the *Create Html Document* action, which uses the above JSON objects:

```
package com.typefi.actions.markdown.run;
import java.io.InputStream;
import java.io.UnsupportedEncodingException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.github.rjeschke.txtmark.Processor;
import com.typefi.sdk.action.ActionRun;
public class MdToHtmlRun extends ActionRun {
    public MdToHtmlRun(final HttpServletRequest request, final
HttpServletResponse response)
            throws UnsupportedEncodingException {
        super(request, response);
    }
    @Override
    public void doAction() throws Exception {
        try (final InputStream inputStream = Files.newInputStream(Paths.
get(getValueMap().get("input")))) {
```

```
            final StringBuilder sb = new StringBuilder();
            sb.append("<!DOCTYPE html><html><head><title></title></head><body>");
            sb.append(Processor.process(inputStream));
            sb.append("</body></html>");
            Files.write(Paths.get(getValueMap().get("output")), sb.toString().
getBytes(StandardCharsets.UTF_8));
        }
    }
}
```

The following sevlet is then created:

`src/main/java/com/typefi/actions/markdown/servlets/MdToHtmlServlet.java`

It creates an instance of the above `MdToHtmlRun` class, calls its `run` method, creates a new `ActionResult` object containing the results and returns this as JSON using the `Response.printActionResultJson` method from `common-server-sdk`.

```
package com.typefi.actions.markdown.servlets;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.typefi.actions.markdown.run.MdToHtmlRun;
import com.typefi.sdk.action.ActionResult;
import com.typefi.sdk.http.Response;
import com.typefi.sdk.plugin.Annotations;
@WebServlet("/md-to-html" + Annotations.RUN)
public class MdToHtmlServlet extends HttpServlet {
    private static final long serialVersionUID = -6721520390257511183L;
    @Override
    protected void doPost(final HttpServletRequest request, final
HttpServletResponse response)
            throws ServletException, IOException {
        Response.printActionResultJson(response, new ActionResult(new
MdToHtmlRun(request, response).run()));
    }
}
```

When the action has finished running it returns the result as JSON, containing its status, page number, percent complete and message, e.g.

```
    {
      "statusCode" : "DONE",
      "pageNumber" : 0,
      "percentComplete" : 100,
      "message" : ""
    }
```

Future extensions

The SDK may be extended to support languages other than Java.

Localization

Overview

Localization can be used in plugins by defining a collection of strings that have different values according to the locale of the application. The `Descriptor.printDescriptor` method within `common-server-sdk` can automatically localize plugin descriptors. The `common-server-sdk` also supports methods for localizing strings elsewhere in the plugin.

Setup

For localization the following folder must be created in the project:

```
src/main/webapp/WEB-INF/conf/typefi-localizations
```

This should contain a `LocalizedMessagesDescriptor.js` file, containing an empty object for every string that is to be localized. For every locale supported, a `LocalizedMessagesBundle_<locale>.js` file should be created, containing a localized entry for every object in `LocalizedMessagesDescriptor.js`; for example, `LocalizedMessagesBundle_en_US.js` for English.

Before localization is used within plugins, the `com.typefi.sdk.localization.PluginLocalizationHelper.configure` method within `common-server-sdk` must be called. For example, to localize as English in response to an `HttpServletRequest` stored within a variable named `request` the method `PluginLocalizationHelper.configure(request, "en_US")` would be called. This step is not required for the `com.typefi.sdk.action.ActionRun.run` and `com.typefi.sdk.plugin.Descriptor.printDescriptor` methods since the call is made automatically.

For the markdown plugin `LocalizedMessagesDescriptor.js` contains the following JavaScript

```
TYPEFI = {};
TYPEFI.plugin = {};
TYPEFI.plugin.action = {};
TYPEFI.plugin.action.displayname = {};
TYPEFI.plugin.action.description = {};
TYPEFI.plugin.action.inputs = {};
TYPEFI.plugin.action.inputs.input = {};
TYPEFI.plugin.action.inputs.input.label = {};
TYPEFI.plugin.action.inputs.output = {};
TYPEFI.plugin.action.inputs.output.label = {};
TYPEFI.plugin.description = {};
TYPEFI.plugin.displayname = {};
TYPEFI.plugin.vendor = {};
```

The `LocalizedMessagesBundle_en_US.js` file contains the following JavaScript

```
TYPEFI.plugin.action.displayname = "Create HTML Document";
TYPEFI.plugin.action.description = "Create HTML Document";
TYPEFI.plugin.action.inputs.input.label = "Input";
TYPEFI.plugin.action.inputs.output.label = "Output";
TYPEFI.plugin.description = "Markdown plugin";
TYPEFI.plugin.displayname = "Markdown";
TYPEFI.plugin.vendor = "Typefi Systems Pty Ltd";
```

The markdown plugin uses the `com.typefi.sdk.action.ActionRun.run` and `com.typefi.sdk.plugin.Descriptor.printDescriptor` methods and therefore does not need to configure localization before use.

Descriptor localization

Descriptor entries such as display names and labels can be localized by simply replacing them with objects within the localized messages bundle file. The `com.typefi.sdk.localization.PluginLocalizationHelper.localizePluginDescriptor` method can then be called, which returns a localized version of the descriptor passed as a parameter. Alternatively, if the `com.typefi.sdk.plugin.Descriptor.printDescriptor` method is used then the descriptor data will be localized automatically.

The localized version of the `descriptor.json` file is:

```
{
    "displayname": TYPEFI.plugin.displayname,
    "description": TYPEFI.plugin.description,
    "vendor": TYPEFI.plugin.vendor,
    "actions": [
        {
            "type": "md-to-html",
            "uiVersion": 1,
            "displayname": TYPEFI.plugin.action.displayname,
            "description": TYPEFI.plugin.action.description,
            "inputs": [
                {
                    "name": "input",
                    "label": TYPEFI.plugin.action.inputs.input.label,
                    "type": "file",
                    "extension": "md",
                    "value": "",
                    "required": true
                    },
                {
                    "name": "output",
                    "label": TYPEFI.plugin.action.inputs.output.label,
                    "type": "file",
                    "extension": "html",
                    "value": "",
                    "output": true,
                    "required": true
                    }
                ]
        }
    ]
}
```

Localization methods

The `com.typefi.sdk.localization.PluginLocalizationHelper.getString` method returns a localized version of the object within the localized messages bundle passed as a string parameter. For example, `PluginLocalizationHelper.getString("TYPEFI.plugin.vendor")` returns the localized version of the `TYPEFI.plugin.vendor` object, which for the markdown plugin will return the value "Typefi Systems Pty Ltd".

Action logging

Overview

Log files are used within Typefi 8 in the job page and admin/log pages to allow the internal status of actions during execution to be examined. Log entries are particularly useful when debugging workflows, to determine what went wrong and where this occurred. The job page view formats and filters the entries according to their category. The valid category values, the last two of which are not intended for display in a UI, are:

- DBUG, a debug message.
- INFO, an informational message.
- WARN, a warning message.
- ERRO, an error message.
- FILE, a message containing a path to another log file. This enables log files from external applications to be included; FILE entries are replaced with the contents of the log file when the file is viewed within the formatted view.
- PROG n, an integer between 1 and 100. This enables progress bars purely through log tailing.
- STAT, which can be COMMENCED, COMPLETED or CANCELLED. The C7D pattern is recommended. This enables unambiguous job status purely through log tailing.

Although actions can use any log file format, by adopting the following format the above filtering and formatting functionality of Typefi 8 can be used to streamline workflow debugging:

```
YYYY-MM-DDTHH:MM:SS.SSSZ XXXX MESSAGE
```

where

- YYYY-MM-DDTHH:MM:SS.SSSZ is the ISO 8601 date in UTC.
- XXXX is the message category.
- MESSAGE is the log message. This should be a single, sentence case line and end with a period.

The Environment object, passed in the request as Json, contains the path to the action log file, which can be retrieved using

```
Json.getEnvironment(request).getActionLogFile();
```

The action can write entries directly to the action log file; alternatively, if the action makes use of external applications that write to log files then it can write a `FILE` reference to each of these log files, eg

```
YYYY-MM-DDTHH:DD:MM:SS.SSSZ FILE C:\foo\bar\action.log
```

When the action log file is viewed as formatted text within Typefi 8, the contents of the files replace the `FILE` entries.

Logger class

A Logger class is contained within `common-server-sdk` that provides methods for writing to action log files at the above levels. This can be created from the request object:

```
final Logger logger = new Logger(request);
```

or if the class extends `com.typefi.sdk.action.ActionRun` then via a getter:

```
this.getLogger();
```

The Logger class provides methods to write messages at a range of levels, or signify that the action has started or finished. For example, the following code logs that the action has started, writes a message at the info level and logs that the action has completed:

```
logger.statusCommenced();
logger.info("Message");
logger.statusCompleted();
```

Alternatively, for actions that make use of external applications that write to their own log files, the following creates a `FILE` reference to the action log file that the external application will use.

```
logger.file("C:/foo/bar/action.log");
```

Long running actions

This section describes two optional servlets that may be implemented for actions with long execution times (of the order of minutes or hours rather than seconds).

Status

The status servlet returns JSON, in the same format as when the action has completed execution, indicating the current status of the action. To obtain the status of an action, Typefi 8 sends a GET request to the status servlet. For example:

```
GET /md-to-html/status
```

The JSON returned by the servlet includes not only its status but how many percent complete it is; for example:

```
{
  "statusCode" : "INPROGRESS",
  "pageNumber" : 0,
  "percentComplete" : 82,
  "message" : ""
}
```

Cancel

When an action is cancelled within Typefi 8 a POST request is sent to its cancel servlet. The request includes an actionid parameter containing the id of the job to be cancelled; for example:

```
POST /md-to-html/cancel?actionid=1234
```

The cancel servlet, if implemented, should contain code that causes the action to end its execution prematurely.

Reference

Input types

- text
- textarea
- password
- file
- files
- folder
- list
- preset (a preset value)
- checkbox
- buttonbar (a set of buttons, each of which can either be selected or deselected)
- conditions (a set of conditions, each of which has a name, boolean status and colour)

Plugin schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "description": "Typefi plugin",
  "type": "object",
  "required": ["displayname","description","vendor","actions"],
  "properties": {
    "displayname": {
      "description": "Name of plugin displayed within user interface",
      "type": "string"
    },
    "description": {
      "description": "Description of plugin",
      "type": "string"
      },
    "vendor": {
      "description": "Vendor of plugin",
      "type": "string"
      },
    "actions": {
      "description": "Actions within the plugin",
      "type": "array",
      "items": {
        "description": "Action within the plugin",
        "type": "object",
        "required": ["type","uiVersion","displayname","description","inputs"],
        "properties": {
          "type": {
            "description": "Action type, eg cxml-to-indb",
            "type": "string"
          },
          "uiVersion": {
            "description": "User interface version",
            "type": "integer"
          },
          "displayname": {
            "description": "Name of action displayed within user interface",
            "type": "string"
          },
          "description": {
            "description": "Description of action displayed
            within user interface",
            "type": "string"
```

```
          },
          "inputs": {
            "type": "array",
            "items": {
              "description": "Input within the action",
              "type": "object",
              "required": ["name","label","type","value","required"],
              "properties" : {
               "tab": {
                  "description": "Tab in UI to which input belongs",
                  "type": "string"
                },
                "name": {
                  "description": "Input name",
                  "type": "string"
                },
                "label": {
                  "description": "Input label",
                  "type": "string"
                },
                "type": {
                  "enum" : ["text","password","textarea","file",
                  "files","folder","list","preset",
                  "checkbox","buttonbar","conditions"]
                },
                "extension": {
                  "description": "Input extension",
                  "type": "string"
                },
                "value": {
                  "description": "Input value",
                  "type": "string"
                },
                "required": {
                  "description": "True if input is required",
                  "type": "boolean"
                },
                "output": {
                  "description": "True if input represents
                  an output of the action",
                  "type": "boolean"
                },
                "placeholder": {
                  "description": "Input placeholder",
                  "type": "string"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Action schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "description": "Typefi action",
  "type": "object",
  "required":
["id","type","uiVersion","disabled","displayname","plugin","inputs"],
  "properties": {
    "id": {
      "description": "Unique identifier",
      "type": "integer"
```

```
      },
    "type": {
      "description": "Action type",
      "type": "string"
    },
    "uiVersion": {
      "description": "User interface version",
      "type": "integer"
    },
    "disabled": {
      "description": "True if action displayed within user interface is disabled",
      "type": "boolean"
    },
    "displayname": {
      "description": "Name of action displayed within user interface",
      "type": "string"
    },
    "plugin": {
      "description": "Plugin containing the action",
      "type": "string"
    },
    "inputs": {
      "type": "array",
      "items": {
        "description": "Input within the action",
        "type": "object",
        "required": ["name", "value"],
        "properties" : {
          "name": {
            "description": "Input name",
            "type": "string"
          },
          "value": {
            "description": "Input value",
            "type": "string"
          }
        }
      }
    }
  }
}
```

GsonEnvironment schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "description": "Typefi environment",
  "type": "object",
  "required":
["jobId","jobFolder","systemType","locale","actionId","actionLogFile",
  "engineVersion", "host","port","inDesignFilestorePath","tpsFilestoreLocation"],
  "properties": {
    "jobId": {
      "description": "Unique identifier for job to which action belongs,
      eg 2147399157",
      "type": "integer"
      },
    "jobFolder": {
      "description": "Folder in which action is run, eg /users/typefi/indd
      /2015-08-05 09.25.40",
      "type": "string"
    },
    "systemType" : {
      "enum" : ["SERVER", "DESKTOP"]
    },
    "locale": {
      "description": "Locale for the system, eg en-US",
```

```
      "type": "string"
    },
    "actionId": {
      "description": "Unique identifier for action, eg 2147399157-1",
      "type": "string"
    },
    "actionLogFile": {
      "description": "File to which action log should be written, eg
      /users/typefi/indd/2015-08-05 09.25.40/logs
      /01 Create InDesign Document.log",
      "type": "string"
    },
    "workflowLogFile": {
      "description": "File to which workflow log should be written, eg
      /users/typefi/indd/2015-08-05 09.25.40/workflow.log",
      "type": "string"
    },
    "serverLogFile": {
      "description": "File to which global server log should be written, eg
      /Library/Application Support/Typefi/tpss.log",
      "type": "string"
    },
    "engineVersion": {
      "description": "InDesign engine version, eg CC2014 (10.2.0.69)",
      "type": "string"
    },
    "host": {
      "description": "InDesign host, eg localhost",
      "type": "string"
    },
    "port": {
      "description": "InDesign port, eg 8470",
      "type": "string"
    },
    "inDesignFilestorePath": {
      "description": "InDesign path used to store files such as templates;
      this may optionally be an empty string",
      "type": "string"
    },
    "tpsFilestoreLocation": {
      "description": "Root of the Typefi Server filestore, eg /users/typefi",
      "type": "string"
    }
  }
}
```

Action result schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema",
  "description": "Typefi action",
  "type": "object",
  "required": ["statusCode","percentComplete","pageNumber","message"],
  "properties" : {
    "statusCode" : {
      "enum" : ["DONE", "CANCELLED", "DELETED", "FAILED", "INITIALIZED",
      "INPROGRESS", "PENDING" ]
    },
    "percentComplete" : {
      "description": "Percentage complete",
      "type": "integer"
    },
    "pageNumber" : {
      "description": "Page number for licensing",
      "type": "integer"
    },
    "message" : {
```

```
      "description": "Error message",
      "type": "integer"
    }
  }
}
```