Verify and convert JSON payloads dynamically for cloud-based applications

Bharat Purohit
Sarang Ratnalikar
October 02, 2013

This article presents a solution architecture that facilitates JavaScript Object Notation (JSON) validation and transformation for the benefit of cloud consumers and cloud service providers. A JVAX (JSON Verification and Conversion/Transformation) system handles all incoming requests before the cloud service receives them. JVAX is designed to address several common problems that can occur when JSON is the payload for an API request.

Basic terms

- **JSON**: A text-based open standard that is designed for transmitting structured data. JSON is most commonly used for transferring data between web applications and web servers.
- **JSON payload**: The JSON-formatted body for a REST API's POST or PUT request.
- **JSON schema**: Describes the validation and transformation rules to be applied to a JSON payload.

This article presents a solution architecture that addresses problems that can occur when interactions between a cloud service and its consumers require verification and conversion of a JavaScript Object Notation (JSON) payload. The architecture — which we call JVAX (JSON Verification and Conversion/Transformation) — uses a configurable JSON schema for verification and conversion.

The development lifecycle of cloud applications with JSON payloads typically involves continual upgrading and integration. Each use case’s additions or modifications can require changes to the application’s JSON verification and conversion logic. As a result, the application can become increasingly fragile and error-prone, and maintaining it is cumbersome.

Generally, cloud services that provide Representational State Transfer (REST) APIs do the considerable work of validating and transforming JSON payloads. The JVAX architecture is designed to verify and transform JSON payloads independently of the cloud service. As a result, a JVAX system reduces the amount of request-validation processing that service providers must do. At the same time, JVAX also helps to make the applications that use it more reliable and maintainable.
JVAX uses a sophisticated mechanism to describe the payload rules. To introduce you to JVAX constructs, let's start with an example.

**JVAX schema definition**

Listing 1 shows a simple JSON payload: A `Person` object.

**Listing 1. Sample input data for a `Person` object**

```json
"Person": {
    "FirstName": "Krishna",
    "LastName": "Yadav",
    "Age": 25,
    "PhoneNumber": "91-012-3456789"
}
```

Listing 2 shows a basic example of a JSON schema for Listing 1's `Person` object:

**Listing 2. Schema definition in JVAX for the `Person` object**

```
{
    "Person": {
        "type": "object",
        "properties": {
            "FirstName": {
                "type": "string",
                "category": "MANDATORY"
            },
            "LastName": {
                "type": "string",
                "category": "MANDATORY"
            },
            "Age": {
                "type": "number",
                "category": "MANDATORY"
            },
            "PhoneNumber": {
                "type": "string",
                "category": "OPTIONAL"
            }
        }
    }
}
```

The JSON schema in Listing 2 describes the `Person` object's fields in terms of various data types. Table 1 lists the types that JVAX supports:

**Table 1. Supported JVAX field types**

<table>
<thead>
<tr>
<th>Supported types</th>
<th>Input values</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Double-quoted string values as supported by <code>java.lang.String</code></td>
<td>&quot;type&quot;: &quot;string&quot;</td>
</tr>
<tr>
<td>boolean</td>
<td>true or false</td>
<td>&quot;type&quot;: &quot;boolean&quot;</td>
</tr>
<tr>
<td>object</td>
<td>An ordered collection of key/value pairs. The <code>properties</code> specifier holds the description of these key/value pairs.</td>
<td>&quot;type&quot;: &quot;object&quot;, &quot;properties&quot;: { /* description of the collection */ }</td>
</tr>
<tr>
<td>array</td>
<td>An ordered sequence of values (that can be of basic types or other object types).</td>
<td>&quot;type&quot;: &quot;array&quot;, &quot;items&quot;: { /* description of array elements */ }</td>
</tr>
</tbody>
</table>

Verify and convert JSON payloads dynamically for cloud-based applications
The elements are assumed to be all of the same type. The items specifier holds the description of the array.

| number | A number value as supported by java.lang.number | "type": "number" |

A value is required for every field in the payload. For each field, the schema applies a category. The categories use internal rules — implemented in the application code's business logic — that determine how JVAX handles the field's values. The rules support a way to override values, which is especially important for preventing the caller from setting invalid values. Table 1 shows the JVAX categories and their associated processing rules:

**Table 2. JVAX categories and associated processing rules**

<table>
<thead>
<tr>
<th>Category</th>
<th>Description (for the API consumer)</th>
<th>Internal processing rules (yes = provided in the input) (no = not provided in the input)</th>
</tr>
</thead>
</table>
| MANDATORY  | API request must provide the value.                                                               | • If yes, use the value.  
• If no, then an error occurs.                                                            |
| OPTIONAL   | API request can optionally provide the value. If a value is not provided, the default value is used. | • If yes, use the value.  
• If no and a default value is provided in the schema, then use it. 
• If no and a default value is not provided in schema, then no operation is performed. |
| RESERVED   | The fields are for internal purposes only. Any API request input values are ignored and replaced with the internal default values in the schema. | • If yes, override with default value.  
• If no and a default value is not provided in schema, then no operation is performed. (No error occurs in either case.) |
| SUPPRESSED | Values are ignored if supplied.                                                                    | • If yes, then delete.  
• If no, then no operation is performed.                                                     |

The schema also supports the use of enumerations, regular expressions, and custom validation mechanisms. And because schemas can refer to shared object definitions, JVAX promotes code reuse.

**The JVAX design**

The key concept in the JVAX design is that requests pass through a JVAX system before the service provider handles them. Figure 1 shows an overview of JVAX components and how they interrelate:

**Figure 1. JVAX overview**
The basic steps that are reflected in Figure 1 are:

1. On the JVAX system, the service provider defines a JSON schema that serves as a template for the API payload.
2. The consumer uses the API to send the JSON payload to the JVAX system.
3. JVAX verifies and converts the input payload in accordance with the schema.
4. When error conditions occur, JVAX responds to the consumer on behalf of the provider.
5. When verification succeeds, JVAX redirects the request to the service provider.

Designing the JSON schema

The process for defining the JSON schema consists of:

1. The service provider defines the API and the associated payload fields.
2. The service provider defines a JSON schema that reflects the API payload's fields.
3. The service provider defines the data types of the fields using the following rules:
   • If a field is required for the API, the service provider flags it as MANDATORY.
   • If a field is not required then it's flagged as OPTIONAL.
   • Fields that should have always a particular set of values are flagged as RESERVED.
   • If a field's value should not be passed to the API, the field is flagged as SUPPRESSED.

Figure 2 shows the overall steps involved in JSON schema design:

**Figure 2. Steps for designing a JSON schema**

![Diagram of JSON schema design steps]

JVAX onboarding process

Figure 3 shows the onboarding process for a JVAX system:
In the process shown in Figure 3:

1. The JVAX system reads the JSON payload that the service consumer provides.
2. JVAX reads the JSON schema that the service provider has provided for the specific API.
3. JVAX tries to validates the JSON schema. If it encounters an error, it returns an error condition and stops.
4. If the schema is valid, JVAX validates the JSON payload against the defined JSON schema. It returns an error condition and stops if the payload doesn't match the signature of the JSON schema.
5. If no errors occur in the payload validation, JVAX (optionally) transforms the input JSON payload in accordance with the API requirements.
6. The service provider base framework calls the API and uses it to send the response to the service consumer.

**Configuration options**

An overall JVAX system can be configured either stand-alone or embedded. In a stand-alone configuration, JVAX is installed separately from the cloud service and runs as a separate process. In an embedded configuration, JVAX is embedded in an existing application and thus shares the same process space. An embedded configuration makes JVAX easy to deploy and configure; it also avoids unnecessary hardware configuration costs.

**JVAX advantages**

The advantages of the JVAX solution architecture are:
• Ease of configuration: JVAX’s rule-based approach makes it easier to impose business rules that are likely to change frequently.
• Pluggability and extensibility: JVAX’s rule-based approach can be embedded inside the actual service or any other existing proxy. Or it can be added as stand-alone proxy where the payload is intercepted and inspected.
• Ease of maintenance: Only configuration changes are needed, rather than redeployment of the service itself.
• Efficiency: Requests don't reach the API unless they are valid, thereby reducing unnecessary processing overhead for the service provider.
• Versioning support: JVAX provides an easy way to maintain external interface compatibility even though the system's internals could be changed by the service provider.
• Documentation support: The schema information can be directly transformed into API documentation.

Conclusion

JSON is widely used for transmitting data between client applications and web servers, making JSON validation and transformation increasingly important in the cloud era. A JVAX system can be a boon to both service providers and service consumers. Its configurability helps service providers to reduce the maintenance effort involved in dealing with the variety of payloads in JSON format. Service consumers can easily see the response from the JVAX system and modify their JSON payloads accordingly.
Related topics

- **JSON**: Read about JSON at Wikipedia.
- **JSON Schema**: The JSON Schema specification, which is in IETF draft status, defines a JSON-based format for defining the structure of JSON data.
- **json-schema.org**: Visit this site for more information about the JSON Schema specification.