Service-Oriented Architecture

RESTful Web Service

Lecture 04

Henry Novianus Palit

hnpalit@petra.ac.id

REST

- REST (REpresentational State Transfer) has gained widespread acceptance across the Web as a simpler alternative to SOAP
- It was first introduced in 2000 by Roy Fielding (UC Irvine) in his dissertation
- It defines a set of architectural principles by which you can design Web services that focus on a system's resources (resource-centric rather than message-centric), including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages
- Four basic design principles:
  - Use HTTP methods explicitly
  - Be stateless
  - Expose directory structure-like URIs
  - Transfer XML, JavaScript Object Notation (JSON), or both

Explicit HTTP Methods *(1)*

- One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol definition in RFC 2616
- The basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods
  - To create a resource on the server → POST
  - To retrieve a resource → GET
  - To change the state of a resource or to update it → PUT
  - To remove or delete a resource → DELETE
- An unfortunate design flaw inherent in many Web APIs is in the use of HTTP methods for unintended purposes, e.g., the use of HTTP GET to trigger something transactional on the server (adding records to a database)

Explicit HTTP Methods *(2)*

- Old way of invoking a remote procedure:

```
GET /adduser?name=Robert HTTP/1.1
```

- RESTful way of invoking the remote procedure:

```
POST /users HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0">
<user>
    <name>Robert</name>
</user>
```

- It shows a proper use of HTTP POST and inclusion of the payload in the body of the request
- A client application may then get a representation of the resource using the new URI

```
GET /users/Robert HTTP/1.1
Host: myserver
Accept: application/xml
```

Explicit HTTP Methods *(3)*

- Using GET in this way is explicit because GET is for data retrieval only; GET's operation should be free of side effects, a property also known as *idempotence*
- Old way of updating data:

```
GET /updateuser?name=Robert&newname=Bob HTTP/1.1
```

- RESTful way of updating the resource:

```
PUT /users/Robert HTTP/1.1
Host: myserver
Content-Type: application/xml
<?xml version="1.0">
<user>
    <name>Bob</name>
</user>
```

- Using PUT to replace the original resource provides a much cleaner interface that is consistent with REST's principles and with the definition of HTTP methods

Explicit HTTP Methods *(4)*

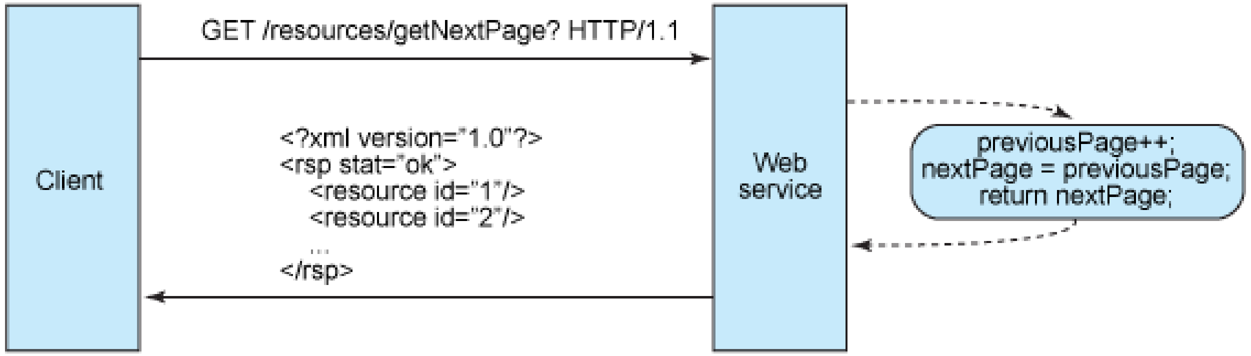
- The PUT request <u>points at the resource to be updated</u> by identifying it in the request URI, and <u>transfers a new representation of the resource</u> in the body of a PUT request instead of transferring the resource attributes as a loose set of parameter names & values on the request URI
- The example PUT request <u>has the effect of renaming the resource from Robert to Bob</u>:
  - The new resource URI is **`/users/Bob`**
  - Subsequent requests for the old resource URI (**`/users/Robert`**) would generate a standard 404 Not Found error
- As a general design principle, REST suggests for using HTTP methods explicitly by <u>using nouns in URIs instead of verbs</u>; the verbs – POST, GET, PUT, DELETE – are defined by protocol
- <u>The body</u> of an HTTP request should be used to <u>transfer resource state</u>, <u>not to carry the name of a remote method</u> to be invoked

Stateless *(1)*

- REST Web services <u>need to scale to meet increasingly high performance demands</u>; clusters of servers with load-balancing & failover capabilities, proxies, and gateways are typically arranged in a way that forms a service topology
- <u>Requests are forwarded from one server to another</u> as needed to decrease the overall response time of a Web service call
- Using intermediary servers to improve scalability requires REST Web service clients to <u>send complete, independent requests that include all data needed</u> so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests

Stateless *(2)*

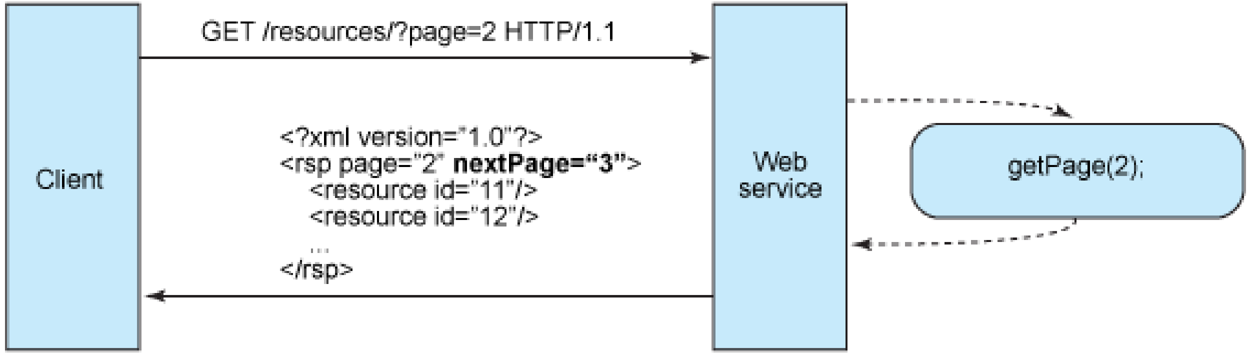
- Statelessness in this sense improves Web service performance and simplifies the design & implementation of server-side components, because the absence of state on the server removes the need to synchronize session data with an external application
- The figure below illustrates a stateful service, from which an application may request the next page in a multipage result set, assuming that the service keeps track of where the application leaves off while navigating the set



```
<?xml version="1.0"?>
<rsp stat="ok">
    <resource id="1"/>
    <resource id="2"/>
    ...
</rsp>
```

Stateless *(3)*

- Stateful services like this require a lot of upfront consideration to efficiently store and enable the synchronization of session data; in addition, session synchronization adds overhead, which impacts server performance
- Stateless server-side components (in the figure below), on the other hand, are less complicated to design, write, and distribute across load-balanced servers



```
<?xml version="1.0"?>
<rsp page="2" nextPage="3">
   <resource id="11"/>
   <resource id="12"/>
   ...
</rsp>
```

- A stateless service not only performs better, it shifts most of the responsibility of maintaining state to the client application

Directory Structure-Like URIs *(1)*

- The URIs determine how intuitive the REST Web service is going to be and whether the service is going to be used in ways that the designers can anticipate
- The structure of a URI should be straightforward, predictable, and easily understood
- One way to achieve this level of usability is to define directory structure-like URIs; it is hierarchical, rooted at a single path, and branching from it are subpaths that expose the service's main areas
- According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and superordinate branches connected at nodes

Directory Structure-Like URIs *(2)*

- For example, in a discussion threading service that gathers a variety of topics, the <u>structured set of URIs</u> is like this:
  `http://www.myservice.org/discussion/topics/{topic}`
  - The root, **`/discussion`**, has a **`/topics`** node beneath it
  - Underneath that, there are a series of topic names, such as gossip, technology, and so on, each of which points to a discussion thread
  - Within this structure, it is easy to pull up discussion threads just by typing something after **`/topics/`**
- <u>Another intuitive structure</u> based on rules:
  `http://www.myservice.org/discussion/{year}/{month}/{day}/{topic}`
  - The first path fragment is a four-digit year
  - The second path fragment is a two-digit month
  - The third path fragment is a two-digit day

Directory Structure-Like URIs *(3)*

- Guidelines to the URI structure for a RESTful Web service:
  - Hide the server-side scripting technology file extensions (e.g., .jsp, .php, .asp), if any, so you can port to something else without changing the URIs
  - Keep everything lowercase
  - Substitute spaces with hyphens or underscores
  - Avoid query strings as much as you can
  - Instead of using the 404 Not Found code if the request URI is for a partial path, always provide a default page or resource as a response
- URIs should also be static so that when the resource changes or the implementation of the service changes, the link stays the same

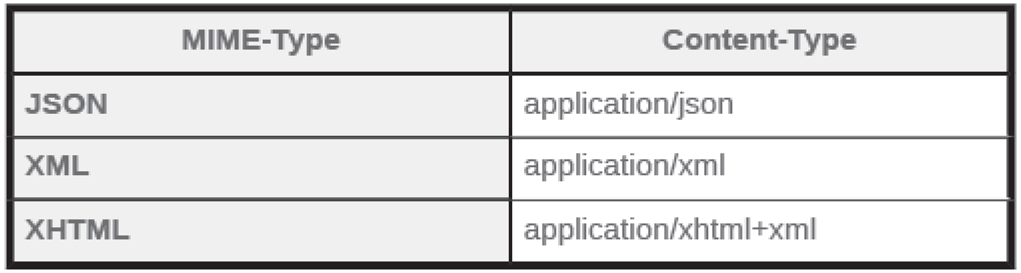
Transfer XML, JSON, or Both *(1)*

- A <u>resource representation</u> typically <u>reflects the current state of a resource, and its attributes</u>, at the time a client application requests it; in the sense, it is a snapshot in time
- The objects in your data model are usually related in some way, and the <u>relationships between data model objects (resources) should be reflected in the way they are represented</u> for transfer to a client application
- For example, in the discussion threading service, connected resource representations might <u>include a root discussion topic & its attributes</u> and <u>embed links to the responses</u> given to that topic

```xml
<?xml version="1.0"?>
<discussion date="{date}" topic="{topic}">
  <comment>{comment}</comment>
  <replies>
    <reply from="joe@mail.com" href="/discussion/topics/{topic}/joe"/>
    <reply from="bob@mail.com" href="/discussion/topics/{topic}/bob"/>
  </replies>
</discussion>
```

Transfer XML, JSON, or Both *(2)*

- To give client applications the ability to request a specific content-type that is best suited for them, construct your service so that it makes use of the built-in HTTP Accept header, where the value of the header is a MIME type
- Some common MIME types used by RESTful services



MIME-Type	Content-Type
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

- This allows the service to be used by a variety of clients, written in different languages, running on different platforms & devices
- Using MIME types and the HTTP Accept header is a mechanism known as *content negotiation*

Remarks

- REST has caught on as a way to design Web services with less dependence on proprietary middleware than the SOAP- and WSDL-based kind
- XML over HTTP is a powerful interface that allows internal applications, such as Asynchronous JavaScript + XML (Ajax)-based custom user interfaces, to easily connect, address, and consume resources; in fact, the great fit between Ajax and REST has increased the amount of attention REST is getting
- Exposing a system's resources through a RESTful API is a flexible way to provide different kind of applications with data formatted in a standard way (i.e., mashups)

References

- Alex Rodriguez, “RESTful Web Services”, 6 Nov 2008, URI=https://developer.ibm.com/articles/ws-restful/
- WebConcepts, “REST API concepts and examples”, URI=https://www.youtube.com/watch?v=7YcW25PHnAA

Service-Oriented Architecture

Overview of JSON

Lecture 04

Henry Novianus Palit
hnpalit@petra.ac.id

JSON: Definition *(1)*

- JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent syntax for defining data interchange formats
- It is a syntax of braces, brackets, colons, and commas useful in many contexts, profiles, and applications
- It was inspired by ECMAScript, the object literals of JavaScript
  - It shares a small subset of ECMAScript's syntax with all other programming languages
  - It does not attempt to impose ECMAScript's internal data representations on other programming languages

JSON: Definition *(2)*

- JSON syntax is <u>not a specification of a complete data interchange</u>; meaningful data interchange requires agreement among the involved parties on the specific semantics to be applied
- It <u>describes a sequence of Unicode code points</u>; it also <u>depends on Unicode in the hex numbers</u> (used in the `\u` excapement notation)
- It is agnostic about the semantics of numbers; it <u>offers only the representation of numbers that humans use, i.e., a sequence of digits</u>

JSON: Definition *(3)*

- Complex data structures can be easily interchanged between incompatible programming languages
  - JSON provides a simple notation for expressing collections of name/value pairs (to represent **`record`**, **`struct`**, **`dict`**, **`map`**, **`hash`**, or **`object`**)
  - JSON provides support for ordered lists of values (to represent **`array`**, **`vector`**, or **`list`**)
- JSON does not support cyclic graphs
- JSON is not indicated for applications requiring binary data
- Because it is so simple, it is not expected that the JSON grammar will ever change

JSON Text *(1)*

- JSON text is <u>a sequence of tokens formed from Unicode code points</u> that conforms to the JSON *value* grammar
- The set of tokens includes six structural tokens, *strings*, *numbers*, and three literal name tokens

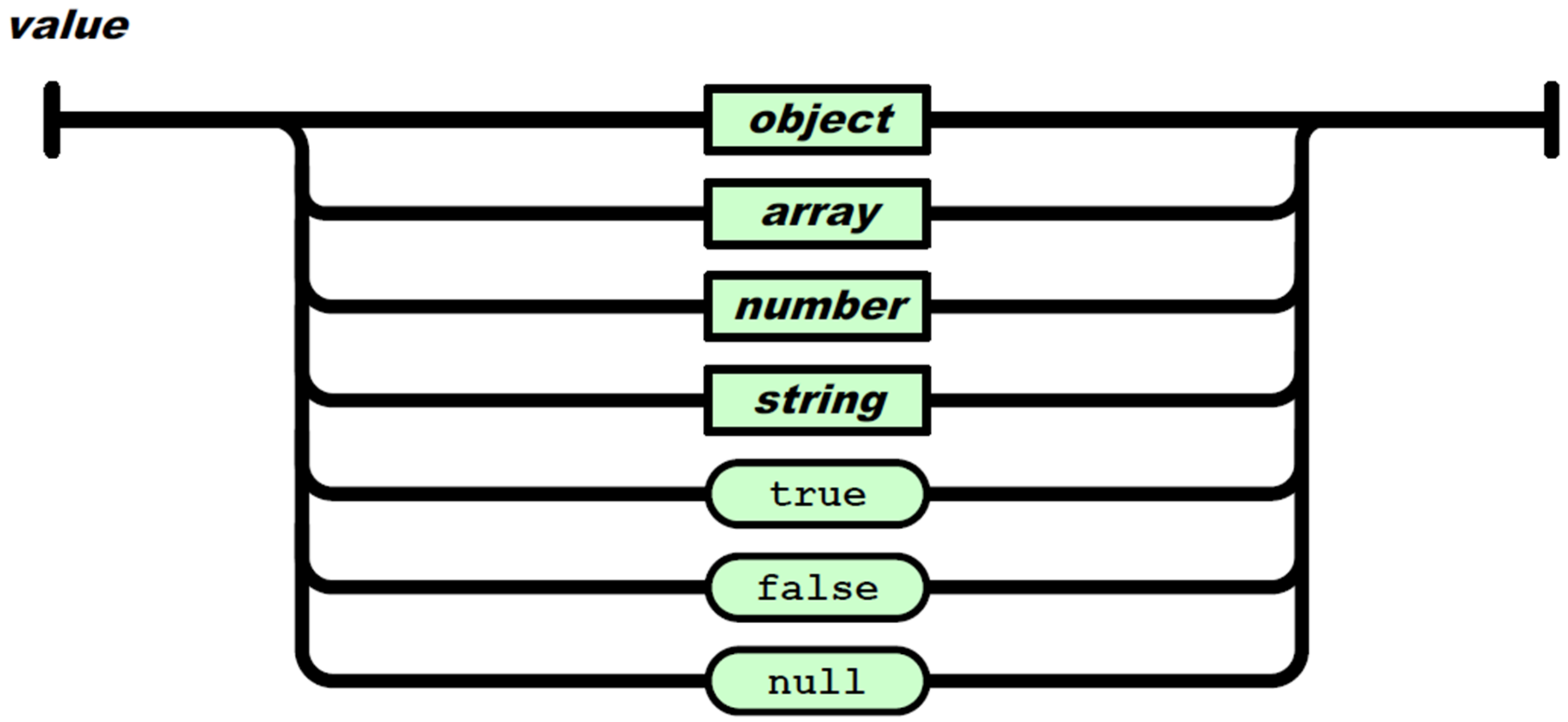
Token	Unicode	Description
[	U+005B	left square bracket
{	U+007B	left curly bracket
]	U+005D	right square bracket
}	U+007D	right curly bracket
:	U+003A	colon
,	U+002C	comma

Token	Unicodes
`true`	U+0074 U+0072 U+0075 U+0065
`false`	U+0066 U+0061 U+006C U+0073 U+0065
`null`	U+006E U+0075 U+006C U+006C

JSON Text *(2)*

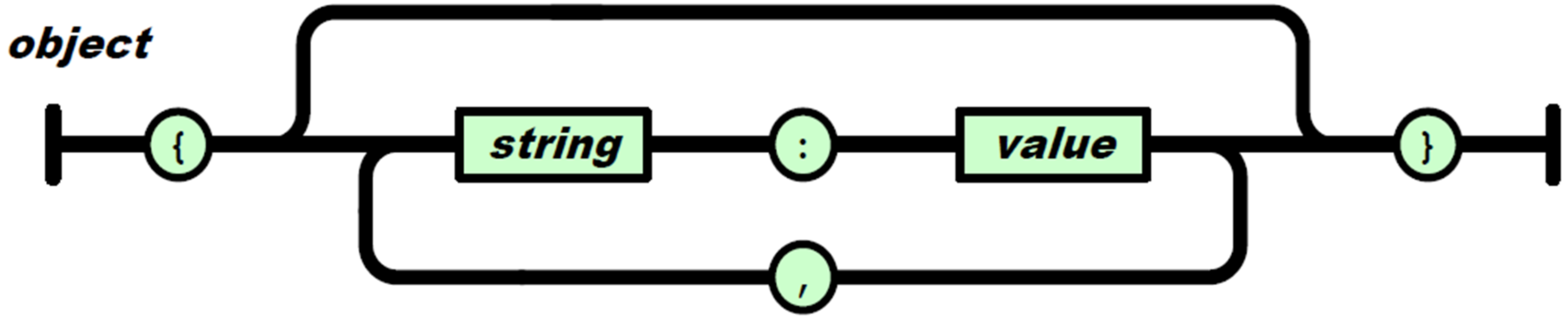
- Whitespace is any sequence of one or more of the following code points:
  - character tabulation (U+0009),
  - line feed (U+000A),
  - carriage return (U+000D), and
  - space (U+0020)
- <u>Insignificant whitespace is allowed before or after any token</u>
- <u>Whitespace is not allowed within any token, except</u> that space is allowed <u>in *strings*</u>

JSON Values



JSON Objects *(1)*

- An *object* is represented as a pair of curly bracket tokens surrounding zero or more name/value pairs
  - A name is a *string*
  - A single colon token follows each name, separating the name from the *value*
  - A single comma token separates a *value* from a following name



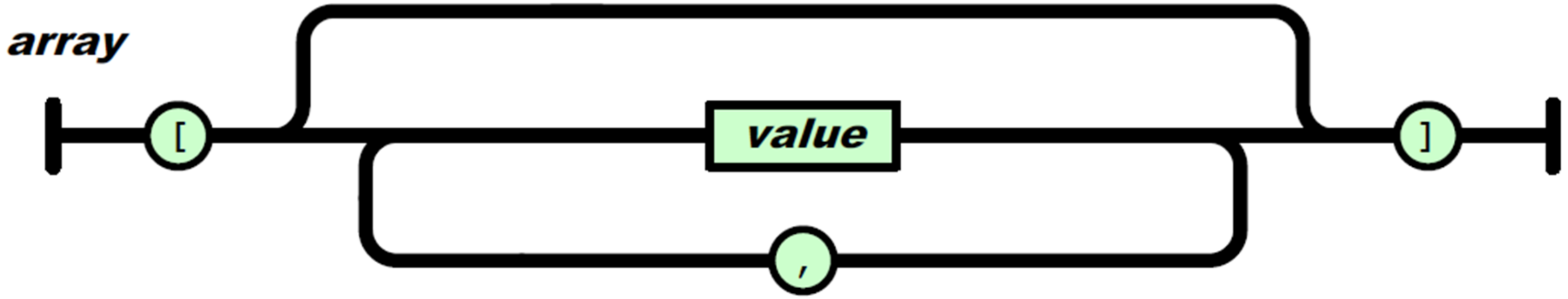
JSON Objects *(2)*

- JSON syntax DOES NOT
  - impose any restrictions on the *strings* used as names,
  - require that name *strings* be unique, and
  - assign any significance to the ordering of name/value pairs

- All semantic considerations may be defined by JSON processors or in specifications defining specific uses of JSON for data interchange

JSON Arrays

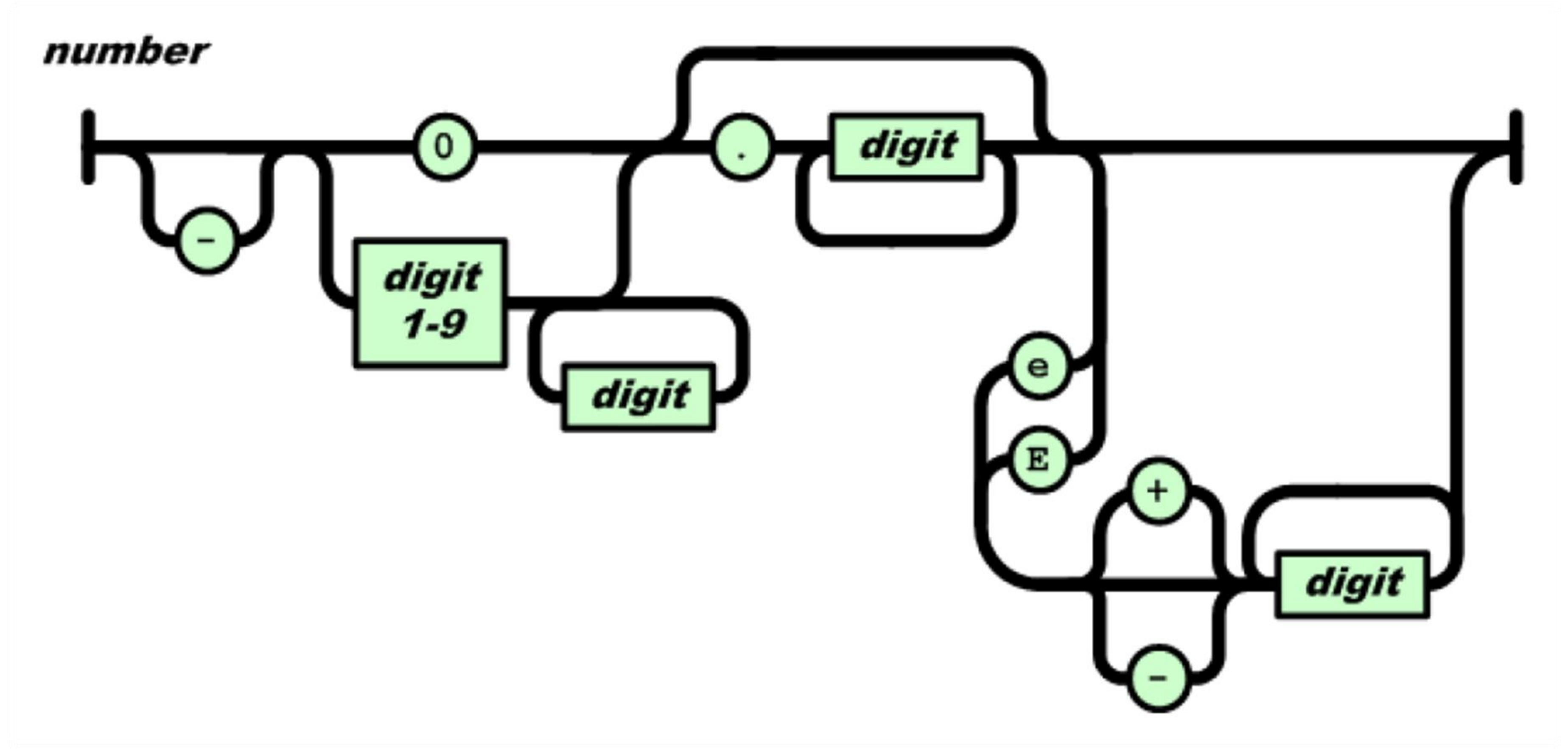
- An *array* is <u>a pair of square bracket tokens surrounding zero or more *values*</u>; the *values* are separated by commas
- JSON syntax DOES NOT define any specific meaning to the ordering of the *values*, although the *array* is often used in situations where there is some semantics to the ordering



JSON Numbers *(1)*

- A *number* is a sequence of decimal digits with no superfluous leading zero
- It MAY
  - have a preceding minus sign (U+002D)
  - have a fractional part prefixed by a decimal point (U+002E)
  - have an exponent, prefixed by **`e`** (U+0065) or **`E`** (U+0045) and optionally **+** (U+002B) or **–** (U+002D)
- The digits are the code points U+0030 (digit 0) through U+0039 (digit 9)
- Numeric values that cannot be represented as sequences of digits (such as **`Infinity`** and **`NaN`**) are not permitted

JSON Numbers *(2)*



JSON Strings *(1)*

- A *string* is <u>a sequence of Unicode code points wrapped with quotation marks</u> (U+0022)
- All code points may be placed within the quotation marks except for the code points that must be escaped:
  - quotation mark (U+0022),
  - reverse solidus (U+005C), and
  - the control characters U+0000 to U+001F

JSON Strings *(2)*

- There are two-character escape sequence representations of some characters:
  - `\"` represents the quotation mark character (U+0022)
  - `\\` represents the reverse solidus character (U+005C)
  - `\/` represents the solidus character (U+002F)
  - `\b` represents the backspace character (U+0008)
  - `\f` represents the form feed character (U+000C)
  - `\n` represents the line feed character (U+000A)
  - `\r` represents the carriage return character (U+000D)
  - `\t` represents the tabulation character (U+0009)

JSON Strings *(3)*

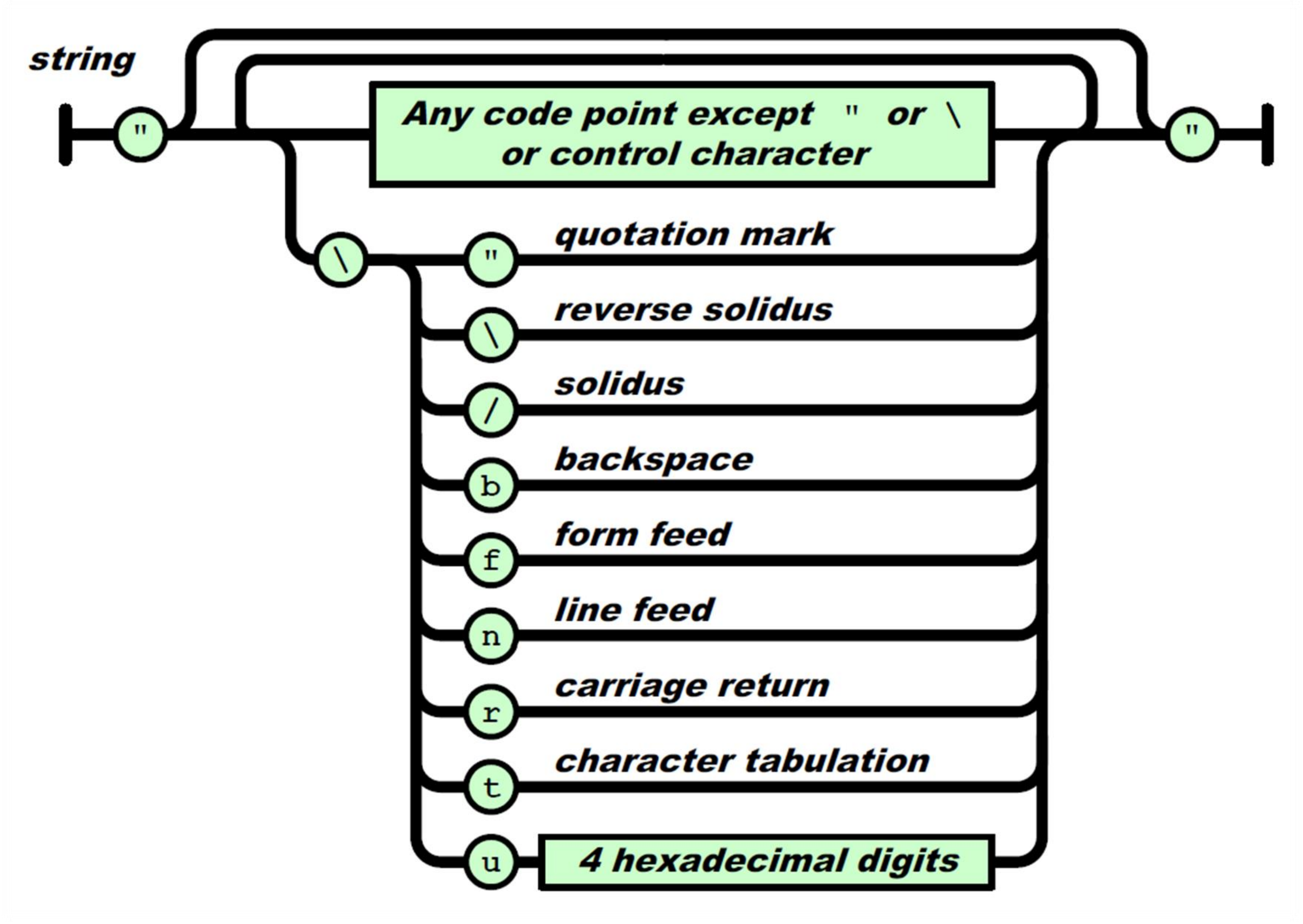
- Any code point may be represented as a hexadecimal escape sequence; the meaning is determined by ISO/IEC 10646 (UCS / Universal Coded Character Set)
- If the code point is in the Basic Multilingual Plane (U+0000 through U+FFFF), then it may be represented as a six-character sequence:
  - a reverse solidus,
  - followed by the lowercase letter **`u`**,
  - followed by four hexadecimal digits that encode the code point

E.g.: **`"\u005c"`** (a string containing only a single reverse solidus character)

JSON Strings *(4)*

- To escape a code point that is in the Supplementary Planes, the character may be represented as a twelve-character sequence, encoding the UTF-16 surrogate pair corresponding to the code point; e.g., **`"\uD834\uDD1E"`** (a string containing only the G clef character U+1D11E)
  - Whether a processor of JSON texts interprets such a surrogate pair as a single code point or as an explicit surrogate pair is a semantic decision that is determined by the specific processor

JSON Strings *(5)*



References

- ECMA International, "The JSON data interchange syntax", ECMA-404, 2nd edition, Dec 2017, URI=https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf