Neulinx Web Platform Data Engine REST API Reference

概述

当前版本的 **Neulinx Data Engine** 底层采用 **arangodb** 作为 Graph Database。Neulinx Data Engine REST API 采用arangodb提供的 **Foxx Microservice** 作为框架进行开发。DataEngine API 当前版本没有提供鉴权认证功能，默认 DataEngine 运行在内网环境中。后续会加入对 API 调用的鉴权管理，将会采用 Foxx 提供的 API KEY 和 Session 机制。

当前的版本的 Data Engine 主要实现两个功能：

1. 采用 Graph 方式组织访问数据。
2. 采用模板方式进行数据渲染。(当前版本支持Handlebars模板）。

通用部分

URL模板

DE REST API 中 URL 模板如下：

```
http://主机信息/_db/数据库名/*挂载路径/*API信息
```

- 在默认安装的开发环境下，主机信息通常是: `localhost:8529`
- 默认数据库为: `_system`
- 其中`*挂载路径`也就是虚拟路径，安装*DataEngine Application*时的*mount path*。
- 在使用`_system`默认库的情况下，可以省略掉`/_db/_system`。

其中 API 信息部分又采用如下格式模板：

```
/版本号/API大类/*调用信息
```

对于当前版本的Graph大类 API，具体的 URL 模板为：

```
/v1/g/stub/key/path?参数表
```

- `v1` 是当前API版本号：1.0
- `g` 是当前API的分类：Graph

- `stub/key` 用于在 Graph 结构中定位一个**锚点**，从而把网状结构转化为树状结构。然后可以以此**锚点**为根，进行树状遍历，找到最后的叶节点。
- stub = `_uuid`，key 的值为 nodes 中的 `uuid` 值。所有资源的全局唯一标示。
- stub = `_key`, key 的值就是 nodes collection 中的 `_key` 值，在该 collection 中唯一。
- stub = 外部 collection 的名称，`stub/key` 就是当前数据库中其他 collection 中某个 document 的 `_id`。DataEngine 通过 nodes 中的 `ref` 字段索引到其在 Graph 中的位置。
- `path` 部分是以 Graph Edge Name 作为路径的寻址方式。比如：`/stub/key/trunk/branch/leaf`。
- 参数表目前版本就提供了一个s参数，具体取值如下：
- `s=..`，当前操作是对path指向的Graph Edge数据，即DataEngine中的link数据。
- `s=.`，当前操作的是graph中指向原始数据的node，而不是原始数据本身。
- `s=collectionName`，指定数据源名称是某个collection。通常用于在指定数据源中创建新的数据。
- `s=selection`，用于HTTP GET中选择性的获取数据而不是对象全部的数据。例如某个数据对象有大量的字段，我们只要求返回name, age, sex三个字段信息，参数为：`s=name,age,sex`。注意字段间用“,”分隔。
    - `s=*`，在s没有指定的默认情况，是不返回数据对象中以下划线’_’开头的属性字段，只有显示设置`s=*`时，才会返回全部属性字段。

HTTP Body数据格式

格式统一起见，DE REST API中body统一采用JSON数据格式，即HTTP `Content-Type: application/json`。API中需要body内容的HTTP Method有两个：POST和PUT。

arangodb foxx框架中主要采用 joi做数据验证。当前版本开发中对body字段没有做太多验证，只要是json格式即可。

HTTP GET: 获取数据

Header格式

```
GET /v1/g/:root/:key/*path?s=selection&r=render
```

- `v1`：版本号 version 1.0
- `g`： 类别 Graph
- `:root`：Graph中绝对寻址某个节点用于以此为Root的相对位置寻址。
- `_uuid`：用资源的全局唯一ID定位root。对应Graph.nodes.uuid值。
- `_key`：用Graph.nodes中的唯一标示`_key`寻址。
- `collectionName`：用原始数据寻址，`:root/:key`就是原始数据的`document-handle`
- `:key`：参考`:root`说明，此参数对应不同`:root`参数，有不同的取值。

- `*path`：使用link name进行寻址。`*path`代表后面是一个层次序列，例如：`branch1/branch2/leaf`。

- 特殊取值“.”：当前节点只有一个out link时，可以不用指定名字，比如：`qrcode/./model`。需要注意的是，如果有多个*out link*时，`.`等同于`any`，是第一个找到的*out link*。`.`可以进行重复盲查，比如：`log/first/./././error`，可以用于单线列表式的数据结构快速寻址。

- 特殊取值“..”：当前节点的后向寻址，而不是简单的回退到上一级节点。比如：`父亲/儿子/../母亲`，`儿子`节点有两个*in link*，`父亲`和`母亲`，`..`操作就是寻找*link name*为`母亲`的*in link*。`..`也可以级联进行寻址：`张三/儿子/../../张家`，但是这种级联不是简单回溯，而是等同于`张三/儿子/.././../张家`，是对`儿子`后向寻址任意一个*in link*（可能是`父亲`也可能是`母亲`），然后再从*父母*后向寻址到`张家`。

- `s=selection`：这个参数用于指定GET想要返回的内容。

- `s`参数不存在或者为空时: 完整获取数据对象，但是不返回以’_’开头的内部数据字段。这是默认的情况。

- `s=.`：获取当前的节点数据，而不是所指向的原始数据。

- `s=..`：获取前序节点与当前节点的Link数据。

- `s=选取的属性列表`：选择性的获取数据而不是对象全部的数据。例如某个数据对象有大量的字段，我们只要求返回name, age, sex三个字段信息，参数为：`s=name,age,sex`。注意字段间用“,”分隔。
    - `s=*`，在s没有指定的默认情况，是不返回数据对象中以下划线’_’开头的属性字段，只有显示设置`s=*`时，才会返回全部属性字段。

- `r=render`：这个参数用于指定是否需要对数据进行渲染。

- `r`参数存在，但没有赋值。这是默认情况，从当前数据对象中获取`_template`字段作为模板进行渲染，然后以`_contentType`作为渲染后返回的内容类型。

- `r=path`, 指定对象作为模板进行渲染。Path有如下两种形式：
    - 绝对路径寻址：`/:root/:key/*path`
    - 当root值为`_uuid`时，采用Graph中唯一定义的UUID寻址；
    - 当root值为`_key`时，采用Vertex Key寻址；
    - 其他情况下，采用arangodb的`document handle`方式寻址，即`collection/key`方式寻址。前两种方式是在同一个Graph中的寻址，这一种是在整个数据库中的自由寻址。
    - 相对路径寻址：
    - 以根节点为相对位置的寻址方式：*Path =* `a/b/c`: 以当前*URL*的`:root/:key`为起点的相对路径寻址，等效于`/:root/:key/a/b/c`；
    - 以当前节点为相对位置的寻址方式：*Path =* `./a/b/c`: 以当前节点的`:root/:key/*path`为起点的相对路径寻址，等效于`/:root/:key/*path/a/b/c`。

- 特殊格式：当`*path`部分不存在时，表示采用绝对寻址方式直接对节点操作。与上述带有`*path`格式情况对比，不需要相对`path`寻址，不支持`s=..`获取Link数据的参数，其他都一样。

返回值

- 200：找到指定的数据并返回。
    - 请求完整原始数据时，以数据原本格式返回原始数据，可以是普通字符串格式，也可以是JSON格式。
    - 选择性返回数据：`s=_key,name`
        - `{"_key":"gary","name":"Gary"}`
    - 请求节点数据时，返回例示：
        - `{"_id":handler,"_key":key,"ref":source_id,"type":type,"data":data}`
    - 请求连接数据时，返回结果实例：

```
{
  "_id":"work_de_links/195801828559",
  "_key":"195801828559",
  "_rev":"195801828559",
  "_from":"work_de_nodes/root",
  "_to":"work_de_nodes/195801631951",
  "name":"alice"
}
```

- 404: 在此API代码中，遇到任何异常统一返回如下错误:
    - `{"error":"The route is not viable."}`
- 渲染后的数据按照指定或默认的 `Content-Type` 在Body中返回。

HTTP POST: 创建数据或渲染

Header格式

```
POST /v1/g/:root/:key/*path/leaf?s=source&r=content_type
```

- `v1`: 版本号 version 1.0
- `g`：类别 Graph
- `:root`: Graph中绝对寻址某个节点用于以此为Root的相对位置寻址。
- `_uuid`: 用资源的全局唯一ID定位root。对应Graph.nodes.uuid值。
- `_key`: 用Graph.nodes中的唯一标示_key寻址。
- `collectionName`: 用原始数据寻址，`:root/:key`就是原始数据的`document-handle`
- `:key`: 参考`:root`说明，此参数对应不同`:root`参数，有不同的取值。
- `*path`: 使用link name进行寻址。`*path`代表后面是一个层次序列，例如：`branch1/branch2/branch3`。
- `leaf`: 待新建的叶节点Link名称。
- `s=source`: 这个参数用于指定所创建数据对象的类型或位置。
- s参数不存在或者为空时: 这是默认的情况，与下述`s=.`等效。

- `s=.`: 使用body数据创建新的node，同时创建以**`leaf`**为名字的Link，从**`*path`**指向的节点连接新建的节点。注：此参数不创建外部数据，但可以创建**`type="_self"`**的内部数据：**`data=内部数据`**。
- `s=..`: 创建以当前**`*path`**为起始节点、以body参数中指向的节点为终结点、以leaf为名称的链接。这种情况下只会创建链接，而不会创建节点和原始数据。
- `s=collection`：这种情况下会使用body中的数据，在外部collection中创建数据对象，同时创建一个指向该新建数据对象的node，并创建一个从**`*path`**节点指向新的node的名称为leaf的链接。
- `r=content_type`: 对Body中的内容进行渲染，并按照指定的格式返回。
- r没有赋值时，采用默认的“Content-Type”返回数据，默认“text/html”。
- 特殊格式：下面的操作直接针对DataEngine底层数据操作，通常不推荐。执行时会严格检查数据对象是否与Links、Nodes的定义一致。
- “**`POST /v1/g/._`**”：使用body数据直接创建新的节点。新节点可以在内部存放数据，也可以指向外部数据。本操作不会创建Link和外部数据。
- “**`POST /v1/g/.._`**”：使用body数据直接创建新的链接，创建过程不产生新的节点和外部数据。

Body数据格式

- 外部数据对象：当前版本仅支持同一个数据库的其他`collection`中的`document`。body采用application/json数据格式。其中作为该数据对象键值的`_key`可以自行定义，也可以由系统自动产生。只是`_key`的定义要符合arangodb的命名规则。
- node数据对象：采用DataEngine定义的数据结构，如下所示。
- `{"_key": Key, "ref": Ref, "type": Type, "uuid": UUID, "data": Data}`
- link数据对象：这个数据对象不采用link内部的数据结构，而是采用`{type: Key}`格式，具体有如下几种形式：
- `{"_uuid": UUID}`: 被连接节点的UUID值。
- `{"_key": Key}`: 被连接节点的document key。
- `{"_ref": "Collection/Key"}`: 被连接节点的外部引用数据对象的document-handle。
- `{"_path": Path}`: 被连接节点可以是当前Graph的绝对路径或者相对路径。
    - *Path =* `/:root/:key/*path`: 绝对路径寻址，与上述 *URL* 路径寻址说明相同。
    - *Path =* `a/b/c`: 以当前*URL*的`:root/:key`为起点的相对路径寻址，等效于`/:root/:key/a/b/c`
    - *Path =* `./a/b/c`: 以当前节点的`:root/:key/*path`为起点的相对路径寻址，等效于`/:root/:key/*path/a/b/c`
- 特殊格式“**`POST /v1/g/.._`**”下link数据对象则采用内部数据结构，即：
- `{"_key": Key, "_from": From, "_to": To, "name": Name}`
- 特殊格式“**`POST /v1/g/._`**”下独立节点数据对象则采用内部数据结构，即上述node数据对象格式。

返回值

- 200：正确创建数据对象，并返回摘要信息。如：
    - `{"_id":"work_de_links/202120627671","_rev":"202120627671","_key":"202120627671"}`
- 404: 在此API代码中，遇到任何异常统一返回如下错误:
    - `{"error":"The route is not viable."}`

HTTP PUT: 更改数据

Header格式

```
PUT /v1/g/:root/:key/*path?s=source
```

- v1: 版本号 version 1.0
- g：类别 Graph
- :root: Graph中绝对寻址某个节点用于以此为Root的相对位置寻址。
- _uuid: 用资源的全局唯一ID定位root。对应Graph.nodes.uuid值。
- _key: 用Graph.nodes中的唯一标示_key寻址。
- collectionName: 用原始数据寻址，:root/:key就是原始数据的document-handle
- :key: 参考:root说明，此参数对应不同:root参数，有不同的取值。
- *path: 使用link name进行寻址。*path代表后面是一个层次序列，例如：branch1/branch2/branch3。
- s=source: 这个参数用于指定所创建数据对象的类型或位置。
- s=.: 使用body数据更新当前*path指向的node数据。此操作仅改变node内部数据，不会更改外部的原始数据和链接数据。
- s=..: 使用body中的数据更新当前*path末端的link数据。需要注意的是，由于link内部数据中只有name字段能够被修改，也就是说只有{"name": newName}才真正起作用。
- s未设置或者为空：这是默认情况，使用body中的数据更新原始数据。body中的数据会更新原始数据已有字段、增加原始数据中没有的字段。注：在node内部data字段保存的数据可以是任意形式。
- 特殊格式：当*path部分不存在时，表示采用绝对寻址方式直接对节点操作。与上述带有*path格式情况对比，不需要相对path寻址，不支持s=..操作Link数据的参数，其他都一样。即当s=.时更新Graph Node数据；当s不存在时，更新原始数据。

Body数据格式

- 外部数据对象：当前版本仅支持同一个数据库的其他collection中的document。body采用application/json数据格式。其中作为该数据对象键值的_key可以自行定义，也可以由系统自动产生。只是_key的定义要符合arangodb的命名规则。
- 内部数据对象：直接在node.data中存放数据时，body可以为任意格式，只不过当node.data和body数据都是object类型时，执行union update操作；是其他类型数据时，直接执行覆盖替换操作。
- node数据对象：采用DataEngine定义的数据结构，如下所示。不过由于系统限制，_key是无法被修改的。
    - {"_key": Key, "ref": Ref, "type": Type, "uuid": UUID, "data": Data}
- link数据对象：这个数据对象采用link内部的数据结构，但是由于系统限制，整个link内部数据中，只有name是可以被更改的。因此有效的数据格式是{"name": newName}。

返回值

- 200：正确创建数据对象，并返回摘要信息。如下所示。其中_rev与_oldRev是更新前后的revision号。

    - `{"_id":"male/jerry","_rev":"214392413514","_oldRev":"202177512919","_key":"jerry"}`
- 404: 在此API代码中，遇到任何异常统一返回如下错误:
    - `{"error":"The route is not viable."}`

HTTP DELETE: 删除数据

Header格式

```
DELETE /v1/g/:root/:key/*path?s=source
```

- `v1`: 版本号 version 1.0
- `g`：类别 Graph
- `:root`: Graph中绝对寻址某个节点用于以此为Root的相对位置寻址。
- `_uuid`: 用资源的全局唯一ID定位root。对应Graph.nodes.uuid值。
- `_key`: 用Graph.nodes中的唯一标示_key寻址。
- `collectionName`: 用原始数据寻址，`:root/:key`就是原始数据的`document-handle`
- `:key`: 参考`:root`说明，此参数对应不同`:root`参数，有不同的取值。
- `*path`: 使用link name进行寻址。`*path`代表后面是一个层次序列，例如：`branch1/branch2/branch3`。
- `s=source`: 这个参数用于指定所创建数据对象的类型或位置。
- `s=..`: 仅仅删除当前`*path`最后的叶节点与前序节点间的链接。
- `s=.`: 删除当前`*path`指向的节点，同时自动删除与该节点相连的所有链接。此操作不影响外部原始数据，但是会同时删除内部数据。
- s未设置或者为空：这是默认情况，首先会删除`*path`指向的节点及该节点关联的所有链接，然后如果该node指向外部数据，则会同时删除外部数据对象。
- 特殊格式：当`*path`部分不存在时，表示采用绝对寻址方式直接对节点操作。与上述带有`*path`格式情况对比，不需要相对`path`寻址，不支持`s=..`删除Link数据的参数，其他都一样。即当`s=.`时删除Graph Node及相关链接；当s不存在或者为空时，同时还会删除原始数据。

返回值

- 200：成功删除数据对象后，返回
    - `{"success": true}`
- 404: 在此API代码中，遇到任何异常统一返回如下错误:
    - `{"error":"The route is not viable."}`

数据渲染

当前版本的数据引擎支持基于 Handlebars模板的数据渲染，并在其中扩展了一个名为 `locate` 的块指令。

模板格式

模板中的格式参考Handlebars文档，可以按照Handlebars定义的格式以当前数据节点为context进行数据访问操作。同时数据引擎扩展了一个 `locate` 命令，该命令的参数为上述 `path` 格式的路径信息，进行绝对寻址或者相对寻址。例如，假定数据引擎中存储这样一个Graph：/_key/root/a/b/c/d, 假定每一节点中包含一个代表名称的name字段，其中d节点中包含一个模板 `_template` 字段，其内容如下。

```
<html>
  <head>
    <title> 这里是节点a的name：{{name}}，在此模板中，a节点成为根节点</title>
  </head>
  <body>
    <p>第一层，context是a，显示a的名称： {{name}}</p>
    <p>使用 locate 命令进入第二层：{{#locate "b"}} 这里的context是b，显示节点b的名称：{{name}}，
locate的参数“b”是从锚点a开始的相对路径寻址，此处等同于"./b"。{{/locate}}</p>
    <p>两级路径进入第三层：{{#locate "b/c"}}这里的context是c，显示c的名称：{{name}}。{{/locate}}
</p>
    <p>采用嵌套方式：
      {{#locate "b"}}这里是节点b的名称：{{name}}。
        {{#locate "./c"}}这里是c的名称：{{name}}。需要注意的是，这里"./c"参数跟"c"不同，前者路径为：
`/root/a/b/c`，后者的路径为`/root/a/c`。{{/locate}}
      {{/locate}}
    </p>
    <p>直接访问root名称：{{locate "/_key/root" "name"}}，直接访问d的名称：{{locate "./b/c/d",
"name"}}</p>
  </body>
</html>
```

采用 `GET /v1/g/_key/root/a?r=./b/c/d` 获取渲染后的数据（r值等效于： `r=a/b/c/d`，或者 `r=/_key/root/a/b/c/d` ）：

```
<html>
  <head>
    <title> 这里是节点a的name：a_name，在此模板中，a节点成为根节点</title>
  </head>
  <body>
    <p>第一层，context是a，显示a的名称： a_name</p>
    <p>使用 locate 命令进入第二层： 这里的context是b，显示节点b的名称：b_name，locate的参数“b”是从锚
点a开始的相对路径寻址，此处等同于"./b"。</p>
    <p>两级路径进入第三层：这里的context是c，显示c的名称：c_name。</p>
    <p>采用嵌套方式：这里是节点b的名称：b_name。这里是c的名称：c_name。需要注意的是，这里"./c"参数
跟"c"不同，前者路径为：`/root/a/b/c`，后者的路径为`/root/a/c`。</p>
    <p>直接访问root名称：root_name，直接访问d的名称：d_name</p>
  </body>
</html>
```

后端渲染

用于数据渲染的模板要求存储在同数据库中任意可被访问位置的document中。当前版本要求该document至少包含一个 `_template` 字段存储模板，还可以包含一个 `_contentType` 的字段指定渲染后数据的内容格式，用于HTTP ’content-type’字段。

参考上述HTTP GET中render参数的描述。GET获取数据时，可以在指定模板上进行数据渲染，该数据模板的context数据即为GET所指向的资源节点。当 `r` 参数没有赋值时，默认当前数据对象中包含模板相关数据字段。 `r` 参数可以携带模板存储路径信息，具体格式参考HTTP GET相关描述。

前端渲染

前端WEB应用也可以把复杂的数据访问以模板方式推送到后端，由后端进行渲染后一次性返回所有的数据访问结果。这个数据推送操作采用HTTP POST方式，详见前面的内容。以 curl 举例：

```
curl -d "root 的 名 字 ： {{name}}, c 的 名 字 ： {{locate "a/b/c" "name"}}" http://host:port/mount/v1/g/_key/root?r
```

返回：

```
root的名字：root_name, c的名字：c_name
```