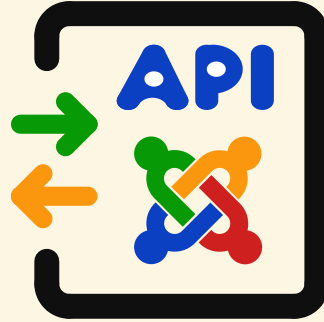


Web Services API:

Implementation for components



Dipping the toe into the sea of joomla webservice API coding for components

- Minimum component API code
- Standard use of the component controller/models
- Patch the data before reaching component controller/models
- Prepare the JSON response

The presentation is aimed at component developers who have no knowledge of Joomla's API functions.

Thomas Finner

InsideTheMachine.de

2008: First Joomla! web page

vollmondlauf-hechingen.de

Full moon fun run, no race, once per month, running up the hill



Using **RSGallery2** as my gallery for images



2013: RSGallery2 Missing support

- Founder with breakdown,
- Rsgallery2.net domain was captured and offered for money (1000\$),
- One person maintaining forum-server
- One person upgrading from J! 2.5 to 3.x

2014 Start of coding

- Start of transfer code from 3x to 4x to 5x
- Learning the ropes, first code in PHP
- First joomla! Day / JCamp Essen



2026 Actual state of oldest living gallery

- Baseline operational
- Needs development
- Some big ideas failed due to J! code expectations
- Backend Ok, Site needs help

2024++ support for JoomGallery Friends



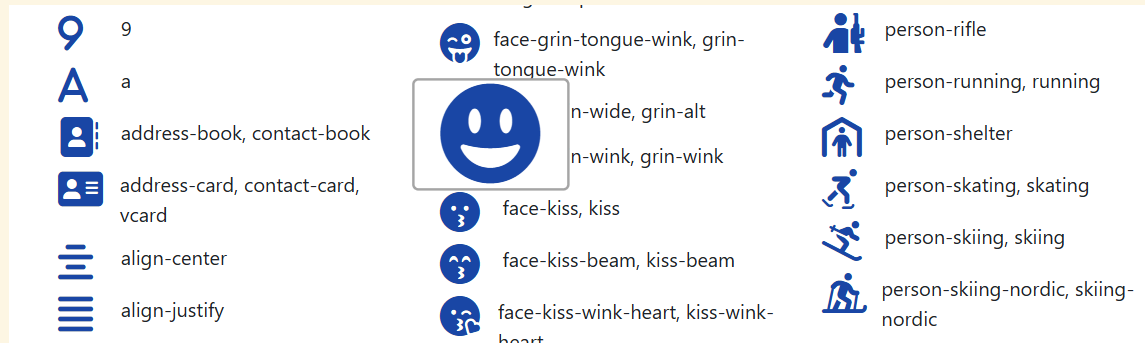
- Code for site upload
- Code for Web Services API

Where my knowledge of Joomla! API started
(the hard way by checking the code)

2024++ Module to display standard J! icons

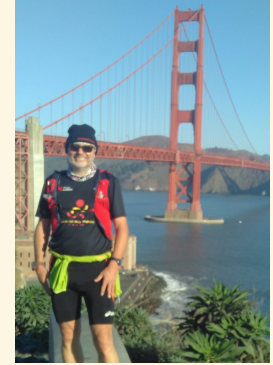
(j4x/J5x/Jx) mod_jx_std_icons

- List Joomla! standard icons on your website. (localhost | web)
- Part of guide.joomla.org: [Icons in standard templates](#)



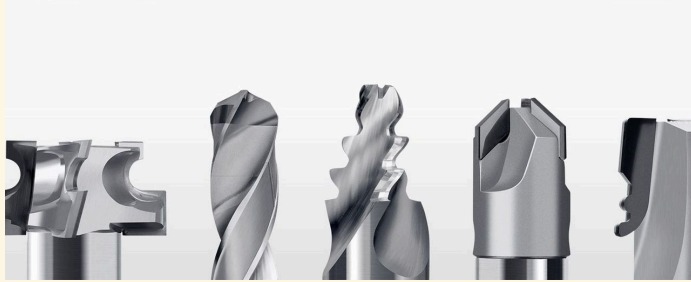
JED: <https://extensions.joomla.org/extension/mod-jx-std-icons>

Hobby:



Profession:

1999 - 2027 creating software at Walter Machines Tübingen



Location of code and a more detailed description

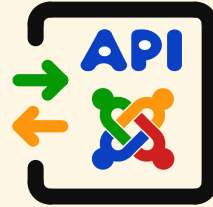
In this presentation, the code is shown in abbreviated excerpts.

A complete version can be viewed at on my Github account:
https://github.com/ThomasFinnern/J_api_demo/tree/main/code

Parts of this presentation will later be transferred to articles in
<https://guide.joomla.org>

The slides will be present as a pdf version on
<http://www.insidethemachine.de/index.php/joomla-projects/presentations>

First part



- Minimum component com_secondhand
- API: From calling to JSON result

Introduction component "com_secondhand"

Imagine a second hand books component with just the one books table and matching view.

1. Scan an book ISBN
2. Fetch book data by other web API
3. Use a J! web API to fill this table.

Table items

- id
- name
- author
- isbn
- publish, created, ...>

Which API routes do we want to expose ?

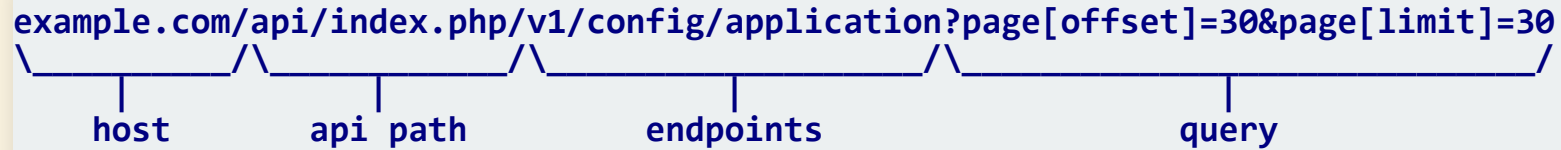
CRUD definitions (GET/POST/PATCH/DELETE)
Which would look like following

Command	Endpoint	Comment
GET	v1/secondhand/books	lists all books with variables
GET	v1/secondhand/books/:id	lists variables of book
POST	v1/secondhand/books	creates new book with data
PATCH	v1/secondhand/books/:id	writes parameters into selected book
DELETE	v1/secondhand/books/:id	deletes selected book

URL structure for Joomla API calls

Joomla API form:

```
example.com/api/index.php/v1/config/application?page[offset]=30&page[limit]=30
```



The diagram illustrates the structure of a Joomla API URL. The URL is broken down into four main components: 'host' (example.com), 'api path' (/api/index.php/), 'endpoints' (v1/config/application), and 'query' (page[offset]=30&page[limit]=30). Brackets and vertical lines connect these labels to their respective parts in the URL.

Api path:

- /api/index.php/ -> webservice starting in web site '/api' folder

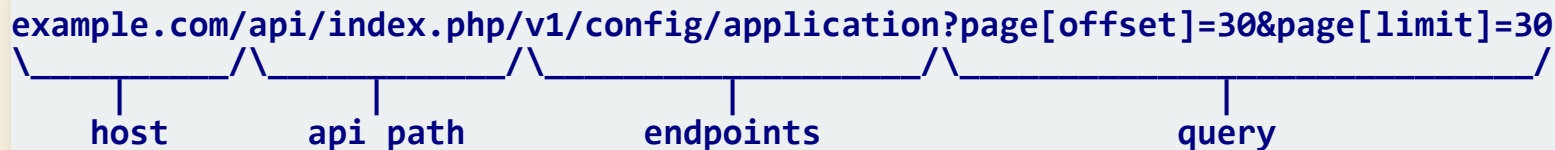
Endpoints:

- v1/config/application -> routes to address functionality

URL structure for Joomla API calls

Joomla API form:

```
example.com/api/index.php/v1/config/application?page[offset]=30&page[limit]=30
```



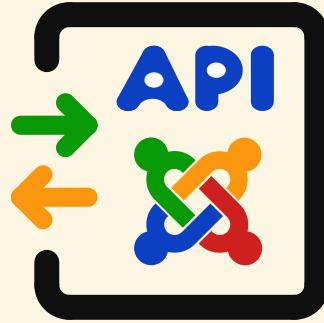
The diagram illustrates the structure of a Joomla API URL. The URL is broken down into four main components: 'host' (example.com), 'api path' (api/index.php), 'endpoints' (v1/config/application), and 'query' (page[offset]=30&page[limit]=30). Brackets and vertical lines connect these labels to their respective parts of the URL.

Query:

- pagination: `page[offset]=30&page[limit]=30`
- filter: `filter[catid]=2&filter[state]=1`
- sorting: `list[ordering]=created&list[direction]=desc`
- fields: `fields[articles]=title,catid,state`

Special characters like '[' and ']' must be converted to '%5B' and '%5D' ...
like in other HTML queries

Bare minimum component



- `plg_webservices_secondhand`
- `com_secondhand`

This example is valid only if the API is intended for accessing data in database tables.

webservice API plugin

The general base component folder structure

```
1 plg_secondhand
2   |
3   | language
4   |   |
5   |   | en-GB
6   |   |   |
7   |   |   | plg_content_secondhand.ini
8   |   |   | plg_content_secondhand.sys.ini
9   |   |
10  |   | services
11  |   |   |
12  |   |   | provider.php
13  |   |
14  |   | src
15  |   |   |
16  |   |   | Extension
17  |   |   |   |
18  |   |   |   | Secondhand.php
19  |   |
20  |   | secondhand.xml
```

Structure similar to other plugins

- provider.php Registers the service provider with a DI container.
- Secondhand.php Endpoint definitions

>

Manifest file Plugin 'secondhand.xml'

Following is an excerpt as the content is known

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <extension type="plugin" group="webservices" method="upgrade">
3   <name>plg_webservices_secondhand</name>
4   ...
5   <description>PLG WEBSERVICES SECONDHAND XML DESCRIPTION</description>
6   <namespace path="src">Bluebox\Plugin\WebServices\Secondhand</namespace>
7   <files>
8     <folder plugin="secondhand">services</folder>
9     <folder>language</folder>
10    <folder>services</folder>
11    <folder>src</folder>
12    <file>secondhand.xml</file>
13  </files>
14  ...
15 </extension>
```

Attention: The namespace contains 'WebSpaces' with an uppercase 'S' in the middle

Service provider 'services/provider.php'

Here the 'secondhand' API definition class is instantiated and listed as a service provider.

```
1 use Joomla\...\n2\n3 use Bluebox\\Plugin\\WebServices\\Secondhand\\Extension\\Secondhand;\n4\n5 return new class () implements ServiceProviderInterface{\n6     /* Registers the service provider with a DI container.\n7      * @param Container $container The DI container.\n8      * @return void */\n9     public function register(Container $container): void\n10    {\n11        $container->set(\n12            PluginInterface::class,\n13            $container->lazy(Secondhand::class, function (Container $container){\n14                $plugin = new Secondhand(\n15                    (array) PluginHelper::getPlugin('webservices', 'secondhand')\n16                );\n17                $plugin->setApplication(Factory::getApplication());\n18\n19                return $plugin;\n20            })\n21        );\n22    }\n23 };
```

Route extension class 'src\Extension\Secondhand.php'

part one

function getSubscribedEvents: Register event

```
1  /* Joomla! Webservices Plugin webservice secondhand. */
2  final class Secondhand extends CMSPlugin implements SubscriberInterface
3  {
4      /* Returns an array of events this subscriber will listen to */
5      public static function getSubscribedEvents(): array
6      {
7          return ['onBeforeApiRoute' => 'onBeforeApiRoute',,];
8      }
9
10     ....
```

Route extension class 'src\Extension\Secondhand.php'

part two

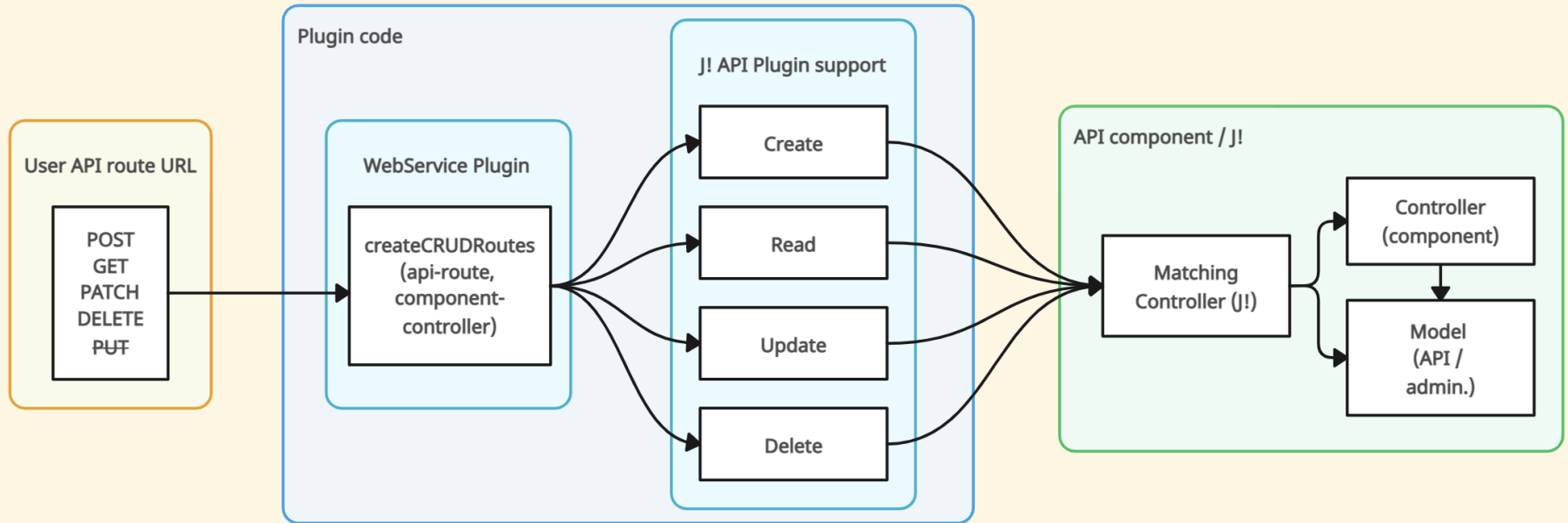
function onBeforeApiRoute

```
1 public function onBeforeApiRoute(BeforeApiRouteEvent $event): void
2 {
3     $router = $event->getRouter();
4     $defaults = ['component' => 'com_secondhand'];
5     $getDefaults = array_merge(['public' => false], $defaults);
6     $router->createCRUDRoutes(
7         'v1/secondhand/books',
8         'books',
9         $getDefaults
10    );
11 }
12
13 }
```

createCRUDRoutes: Here the magic happens.

createCRUDRoutes

Internal routes created



function `createCRUDRoutes` has defined all CRUD routes internally

The magic applied

instead of

```
$router->createCRUDRoutes(  
    'v1/secondhand/books',  
    'books',  
    $getDefaults  
);
```

We could have written:

```
1  $routes = [  
2      new Route(['GET'], 'v1/secondhand/books', 'books.displayList', [], $defaults)  
3      new Route(['GET'], 'v1/secondhand/books/:id', 'books.displayItem', ['id' => '(\d+)'], $defaults)  
4      new Route(['POST'], 'v1/secondhand/books', 'books.add', [], $defaults)  
5      new Route(['PATCH'], 'v1/secondhand/books/:id', 'books.edit', ['id' => '(\d+)'], $defaults)  
6      new Route(['DELETE'], 'v1/secondhand/books/:id', 'books.delete', ['id' => '(\d+)'], $defaults)  
7  ];  
8  $this->addRoutes($routes);  
9
```

'new Route(...)' will be the
syntax for all our defined API routes

createCRUDRoutes function parameters

```
$router->createCRUDRoutes(  
  'v1/secondhand/books',  
  'books',  
  $defaults  
);
```

`$route = $router->createCRUDRoutes('base endpoint', 'controller', defaults);`

	example	description
endpoint	v1/secondhand/books	route path
controller	books	path to controller and view
defaults	['component' => 'com_secondhand', 'public' => false]	array of 'configuration' variables

Route function parameters

`new Route('CRUD' command, endpoint, 'controller:function', [var.regex], defaults);`

```
$routes = [ new Route(['GET'],  
    'v1/content/articles/:id/contenthistory', 'history.displayList', ['id' => '(\d+)'], $defaults),];  
$router->addRoutes($routes);
```

	example	description
'CRUD' command	GET,POST,PATCH,DELETE	
endpoint	v1/secondhand/books v1/secondhand/books/:id	route path route path with variable
controller	books	path to controller and view
function	displayItem	any function in defined controller can be called
var.regex	['id' => '(\d+)']	regex to catch and secure the variable value
defaults	['component' => 'com_secondhand', 'public' => false]	array of 'configuration' variables

\$defaults and X-Token

The route needs to know the component name

```
$defaults = ['component' => 'com_secondhand'];
```

Joomla X-Token

The X-Token is the key to the API door. It will be sent with the curl/http data.
The X-Token can be fetched in the user admin account after save.

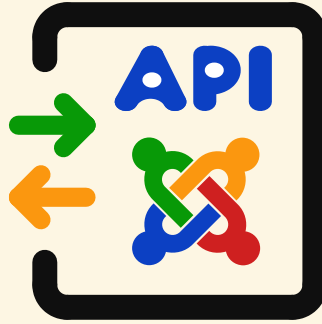
With flag **"'public' => true"**
(on local development) systems we enable API calls without X-Token.

```
1 // ToDo: Remove when tests finished, enables access without token
2 $getDefaults = array_merge(['public' => true], $defaults);
3 $getDefaults = array_merge(['public' => false], $defaults);
```

Please make sure to remove this before going online

Above \$getDefaults is passed as last parameter
to the route function definition

@



Minimum component

When the API just needs to handle table data then following minimum component structure is possible

com_secondhand

Second hand books component with just the one books table and matching view.

Table items

- id
- name
- author
- isbn
- publish, created, ...

Manifest file component 'secondhand.xml'

Add an API section as the component.
The API code will reside there in folder
"/api/component/com_secondhand"

```
1 <extension type="component" method="upgrade">
2   <name>com_secondhand</name>
3   <element>com_secondhand</element>
4   <author>Steven Smith</author>
5   ...
6   <description>COM_SECONDHAND_XML_DESCRIPTION</description>
7   <namespace path="src">Bluebox\Component\Secondhand</namespace>
8
9   <administration>
10   ...
11   </administration>
12
13   <api>
14     <files folder="api">
15       <folder>src</folder>
16     </files>
17   </api>
18   ...
19 </extension>
```

API part of the component 'com_secondhand'

- We have to insert section API into manifest
- We have to add code into the joomla root API folder.

The API sub folder structure:

From the root of the joomla web page

```
api\components\com_secondhand\  
├── src  
│   ├── Controller  
│   │   └── BooksController.php  
│   └── View  
│       ├── Books  
│       └── JsonapiView.php
```

Attention: The API folder structure is similar to other component structures with the exception that the controller folder keeps all `nnnController.php` files without further subdirectories. The view code is always kept in a `JsonapiView.php` file in a named folder

Controller in API part 'BooksController.php'

```
1 namespace Bluebox\Component\Secondhand\Api\Controller;  
2  
3 ...  
4  
5 class BooksController extends ApiController  
6 {  
7     // The content type of the item.  
8     protected $contentType = 'books';  
9  
10    // The default view for the display method.  
11    protected $default_view = 'books';  
12  
13    // Implement other methods like read, update, delete as needed  
14    ...  
15 }
```

Within the minimal controller only two variables need to be defined, as the controller class inherits from ApiController.

- The \$contentType tells which component model to use
- The \$default_view tells which view does render the collected data.

View in API part 'JsonapiView.php'

Inside the minimum controller there are only two variables to implement as the JsonapiView class inherits from BaseApiView.

- The \$fieldsToRenderItem tells which variables of the table to show for single item view
- The \$fieldsToRenderList tells which view does render the collected data.

```
protected $fieldsToRenderItem = [  
    'id',  
    'title',  
    'alias',  
    'isbn',  
    'description', 'note', 'published', 'created',  
    'created_by', 'modified', 'modified_by',  
];
```

```
protected $fieldsToRenderList = [  
    'id',  
    'title',  
    'alias',  
    'isbn',  
    'description', 'note', 'published', 'created',  
    'created_by', 'modified', 'modified_by',  
];
```

Resume:

We defined the API router interface.

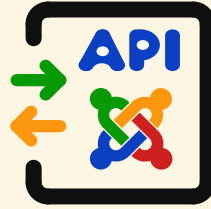
```
$router->createCRUDRoutes(  
    'v1/seconddhand/books',  
    'books',  
    $getDefaults  
);
```

We defined the table and the view

```
protected $contentType = 'books';  
protected $default_view = 'books';
```

We defined what the JSON response should contain for list and item.

```
protected $fieldsToRenderList = ['id', 'title', 'alias', 'isbn', 'description', ...];  
protected $fieldsToRenderItem = ['id', 'title', 'alias', 'isbn', 'description', ...];
```



That's it !

The plugin and two wo files enable
CRUD access over the API of joomla.

- BooksController.php
- JsonapiView.php

Support from the Joomla API base classes is all
that is required to process basic table data.

What API Routes do we get ?

API route definitions created

The CRUD definitions result in following API routes:

Command	endpoint	Comment
GET	v1/secondhand/books	lists all books with variables
GET	v1/secondhand/books/:id	lists variables of book
POST	v1/secondhand/books	creates new book with data
PATCH	v1/secondhand/books/:id	writes parameters into selected book
DELETE	v1/secondhand/books/:id	deletes selected book

Attention: There is no PUT support in Joomla

Calling Joomla API and json response

How can we call the routes and
how does the response look like

GET v1/secondhand/books (lists all books with variables)

Example CURL:

```
curl -s --show-error  
-X GET "http://127.0.0.1/web_page/api/index.php/v1/secondhand/books"  
-H "Content-Type: application/json" -H "X-Joomla-Token: ..."
```

Example HTTP:

```
###  
GET http://127.0.0.1/web_page/api/index.php/v1/secondhand/books  
Accept: application/vnd.api+json  
Content-Type: application/json  
X-Joomla-Token: ...
```

Response: "GET v1/secondhand/books"

http code	content-type	response
200	application/json; charset=UTF-8	json {"type": "books", "id": "2", "attributes": {"id": 2, "title": "Why We Sleep", "alias": "why-we- sleep", , ...}}

Response "GET v1/secondhand/books"

```
{
  "links": {
    "self": "http://127.0.0.1/api_6x/api/index.php/v1/secondhand/books"
  },
  "data": [
    {
      "type": "books",
      "id": "2",
      "attributes": {
        "id": 2,
        "title": "Why We Sleep",
        "alias": "why-we-sleep",
        "isbn": "da1b2048-cec2-4127-950e-35bc4b3ee181",
        "description": "Sleep is one of the most important aspects ...",
        "published": 1,
        "created": "2026-05-27 09:53:06",
        "created_by": 776,
        "modified": "2026-05-27 09:53:06",
        "modified_by": 776,
        "note": ""
      }
    },
    {
      "type": "books",
      "id": "3",
      "attributes": {
```

GET v1/secondhand/books/:id

Example CURL:

```
curl -s --show-error  
-X GET "http://127.0.0.1/web_page/api/index.php/v1/secondhand/books/1"  
-H "Content-Type: application/json" -H "X-Joomla-Token: ..."
```

Example HTTP:

```
###  
GET http://127.0.0.1/web_page/api/index.php/v1/secondhand/books/1  
Accept: application/vnd.api+json  
Content-Type: application/json  
X-Joomla-Token: ...
```

Response: "GET v1/secondhand/books/:id"

http code	content-type	response
200	application/json; charset=UTF-8	json {"type": "books", "id": "2", "attributes": {"id": 2, "title": "Why We Sleep", "alias": "why-we- sleep", , ... }

Response "GET v1/secondhand/books/1"

```
{
  "links": {
    "self": "http://127.0.0.1/api_6x/api/index.php/v1/secondhand/books/1"
  },
  "data": {
    "type": "books",
    "id": "1",
    "attributes": {
      "id": 1,
      "title": "test 01",
      "alias": "test-01",
      "isbn": "1234567890",
      "description": "",
      "published": 1,
      "created": "2026-05-24 17:23:19",
      "created_by": 775,
      "modified": "2026-05-24 17:42:09",
      "modified_by": 775,
      "note": ""
    }
  }
}
```

POST v1/secondhand/books (part 1 / curl)

Example CURL:

```
curl -s --show-error
-X POST "http://127.0.0.1/web_page/api/index.php/v1/secondhand/books"
-d "{
    \"title\": \"My book\",
    \"isbn\": \"12-3456-78-90\",
    \"author\": \"john breeze\",
    \"description\": \"The next book i write\",
    \"published\": 1
}"
-H "Content-Type: application/json" -H "X-Joomla-Token: ..."
```

POST v1/secondhand/books (part 2 / HTTP)

Example HTTP:

```
###
POST http://127.0.0.1/web_page/api/index.php/v1/secondhand/books
Accept: application/vnd.api+json
Content-Type: application/json
X-Joomla-Token: ...
{
  "title": "My book",
  "isbn": "12-3456-78-90",
  "author": "john breeze",
  "description": "The next book i write",
  "published": 1
}
```

Response: "POST v1/secondhand/books"

http code	content-type	response
200	application/json; charset=UTF-8	json {"type": "books", "id": "2", "attributes": { "id": 1, "asset_id": 105, "title": "Global Configuration", ... } }

Response "POST v1/secondhand/books"

```
{
  "links": {
    "self": "http://127.0.0.1/api_6x/api/index.php/v1/secondhand/books"
  },
  "data": {
    "type": "books",
    "id": "3",
    "attributes": {
      "id": 3,
      "title": "My book",
      "alias": "my-book",
      "isbn": "12-3456-78-90",
      "description": "The next book i write",
      "published": 1,
      "created": "2026-06-23 11:40:16",
      "created_by": 776,
      "modified": "2026-06-23 11:40:16",
      "modified_by": 776,
      "note": ""
    }
  }
}
```

PATCH v1/secondhand/books/:id

Example CURL:

```
curl -s --show-error  
-X PATCH "http://127.0.0.1/web_page/api/index.php/v1/secondhand/books/  
-d "{  
    \"published\": -2,  
    \"isbn\": \"1234-4567-8890\"  
}"  
-H "Content-Type: application/json" -H "X-Joomla-Token: ..."
```

Example HTTP:

```
###  
PATCH http://127.0.0.1/web_page/api/index.php/v1/secondhand/books/1  
Accept: application/vnd.api+json  
Content-Type: application/json  
X-Joomla-Token: ...  
{  
    "published": -2,  
    "isbn": "1234-4567-8890",  
}
```

Response: "PATCH v1/secondhand/books/:id#

http code	content-type	response
200	application/json; charset=UTF-8	json {"type": "books", "id": "1", "attributes": { "id": 1, "asset_id": 105, "title": "Global Configuration", ... } }

Response "PATCH v1/secondhand/books"

```
{
  "links": {
    "self": "http://127.0.0.1/api_6x/api/index.php/v1/secondhand/books/1"
  },
  "data": {
    "type": "books",
    "id": "1",
    "attributes": {
      "id": 1,
      "title": "My book",
      "alias": "my-book",
      "isbn": "1234-4567-8890",
      "description": "The next book i write",
      "published": -2,
      "created": "2026-06-23 11:40:16",
      "created_by": 776,
      "modified": "2026-06-23 11:40:16",
      "modified_by": 776,
      "note": ""
    }
  }
}
```

DELETE secondhand/books/:id

Reminder: Delete needs trash state before.

Please use 'PATCH' command with "published": -2, (see above) **Example CURL:**

```
curl -s --show-error  
-X DELETE "http://127.0.0.1/web_page/api/index.php/v1/secondhand/books/  
...-H "Content-Type: application/json" -H "X-Joomla-Token: ..."
```

Example HTTP:

```
###  
DELETE http://127.0.0.1/web_page/api/index.php/v1/secondhand/books/2  
Accept: application/vnd.api+json  
Content-Type: application/json  
X-Joomla-Token: ...
```

Response: None

Response after trash ("published": -2)

```
1 {
2   "links": {
3     "self": "http://127.0.0.1/api_6x/api/index.php/v1/secondhand/books/2"
4   },
5   "data": {
6     "type": "books",
7     "id": "2",
8     "attributes": {
9       "id": 4,
10      "title": "My book",
11      "alias": "my-book-7",
12      "isbn": "12-3456-78-90",
13      "description": "The next book i write",
14      "published": -2,
15      "created": "2026-05-26 16:25:09",
16      "created by": 776,
17      "modified": "2026-05-26 16:41:17",
18      "modified by": 776,
19      "note": ""
20    }
21  }
22 }
```

API routes responses (json)

The results are in JSON style string. They contain the sections

- links
- data
- meta

```
1 {  
2   "links": {  
3     "self": "http://127.0.0.1/api_6x/api/index.php/v1/secondhand/books/1"  
4     ...  
5   },  
6   "data": [ {  
7     "type": "books",  
8     "id": "1",  
9     "attributes": { ... }  
10    },  
11    ...  
12  ],  
13  "meta": {  
14    "total-pages": 4  
15  }  
16 }
```

Links / meta sections of API response

Pagination

- links
 - self
 - first
 - previous
 - next
 - last
- meta
 - total-pages

The links and the "meta:total-pages" sections support easy pagination handling by direct use of the links

Pagination request

For pagination add

```
?page[offset]=90&page[limit]=30
```

(start at 90, 30 items)

which should be written like

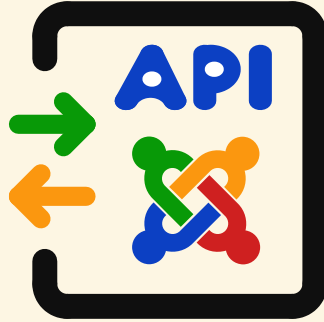
```
?page%5Boffset%5D=90&page%5Blimit%5D=30
```

Links/meta excerpt from Response

```
{
  "links": {
    "self": "http://127.0.0.1/api_6x/api/index.php/v1/config/application
             ?page%5Boffset%5D=30&page%5Blimit%5D=30",
    "first": "http://127.0.0.1/api_6x/api/index.php/v1/config/application
              ?page%5Boffset%5D=0&page%5Blimit%5D=30",
    "previous": "http://127.0.0.1/api_6x/api/index.php/v1/config/application
                 ?page%5Boffset%5D=0&page%5Blimit%5D=30",
    "next": "http://127.0.0.1/api_6x/api/index.php/v1/config/application
             ?page%5Boffset%5D=60&page%5Blimit%5D=30",
    "last": "http://127.0.0.1/api_6x/api/index.php/v1/config/application
             ?page%5Boffset%5D=90&page%5Blimit%5D=30"
  },
  "data": [ ... ],
  "meta": {
    "total-pages": 4
  }
}
```

@

What happens behind the scenes ?



Reminder:

We have defined the component's API router interface as following:

```
$router->createCRUDRoutes(  
    'v1/secondhand/books',  
    'books',  
    $defaults  
);
```

- v1/secondhand/books tells the route
 - "v1" is used as the version
 - "secondhand" is the component
 - "books" is the table.
- Parameter "books" tells the controller file to be used.
- Parameter "\$defaults" contains a config parameter array

>

Parameter "\$defaults"

The \$defaults variable contains a config array and looks like

```
$defaults = ['component' => 'com_secondhand', 'public' => false];
```

If **'public' => true** is set, no X-Token is checked and everyone can access this entry. This should only be set for tests

```
$defaults = ['component' => 'com_secondhand', 'public' => true];
```

Function createCRUDRoutes

The joomla createCRUDRoutes function organizes the code as following:

```
public function createCRUDRoutes($baseName, $controller, $defaults = [], $publicGets = false)
{
    $getDefaults = array_merge(['public' => $publicGets], $defaults);

    $routes = [
        new Route(['GET'], $baseName, $controller : '.displayList', ['id' => '(\d+)'], $defaults),
        new Route(['GET'], $baseName . '/:id', $controller : '.displayItem', ['id' => '(\d+)'], $defaults),
        new Route(['POST'], $baseName, $controller : '.add', ['id' => '(\d+)'], $defaults),
        new Route(['PATCH'], $baseName . '/:id', $controller : '.edit', ['id' => '(\d+)'], $defaults),
        new Route(['DELETE'], $baseName . '/:id', $controller : '.delete', ['id' => '(\d+)'], $defaults),
    ];

    $this->addRoutes($routes);
}
```

Function parameters

- \$baseName -> Endpoint (v1/secondhand/books/1)
- controller provided ->
"displayList", "displayItem", "add", "edit", "delete"

Route endpoint parameter

Examples:

- 'v1/secondhand/books/2'
- 'v1/secondhand/books/The hobbit'

```
new Route(['GET'], 'v1/secondhand/books/:id', 'books.displayItem', ['id' => '(\d+)'], $defaults),  
new Route(['GET'], 'v1/secondhand/books/:title', 'books.displayItem', ['title' => '([A-Z0-9 ]+)', $defaults),
```

A colon ':' indicator with 'name' defines an endpoint variable
The value is secured by regular expression ['id' => '(\d+)']

Fetch API call json variables

```
###
POST http://127.0.0.1/web_page/api/index.php/v1/secondhand/books
Accept: application/vnd.api+json
Content-Type: application/json
X-Joomla-Token: ...
{
  "title": "My book",
  "isbn": "12-3456-78-90",
  "author": "john breeze",
  "description": "The next book i write",
  "published": 1
}
```

The JSON parameters given in the request can be fetched by

```
$author = $this->input->json->getString('author');
```

or all at once

```
$data = json_decode($this->input->json->getRaw(), true);
```

Addressing the controller function

The middle part of the route definition tells about the 'controller.function' call:

```
`new Route(['GET'], 'v1/secondhand/books', 'books.displayList', [], $getDefaults)`
```

Here 'books.displayList' tells the controller name is 'books' and the function to be called is "displayList". This is the entry point in the API part of the component. The code above will lead to file BooksController.php.

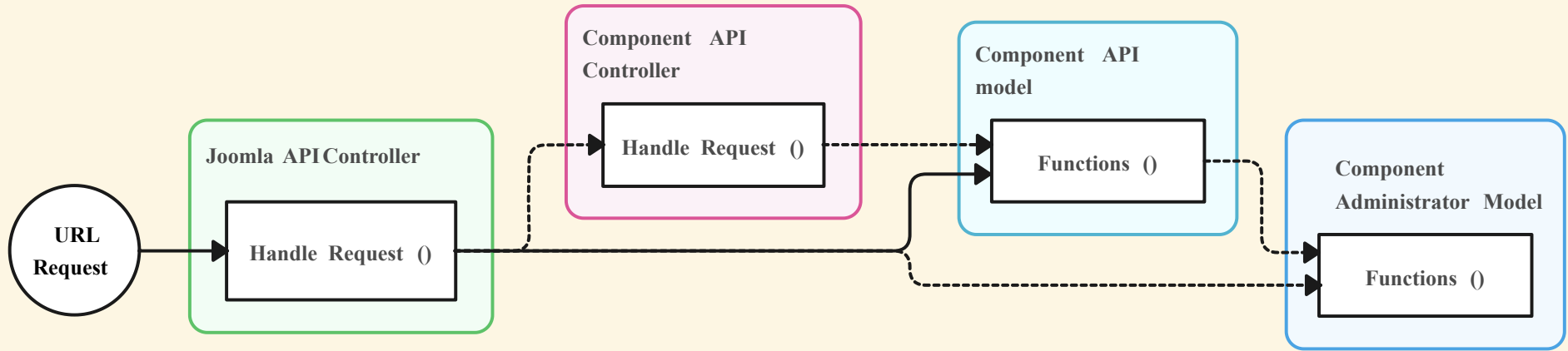
Any combination which leads to a valid php file class and function may be used.

Example: 'myControllerClass.someFunction'

The controller can leave the handling of model and view to the joomla API functions or define its own.

Code workflows

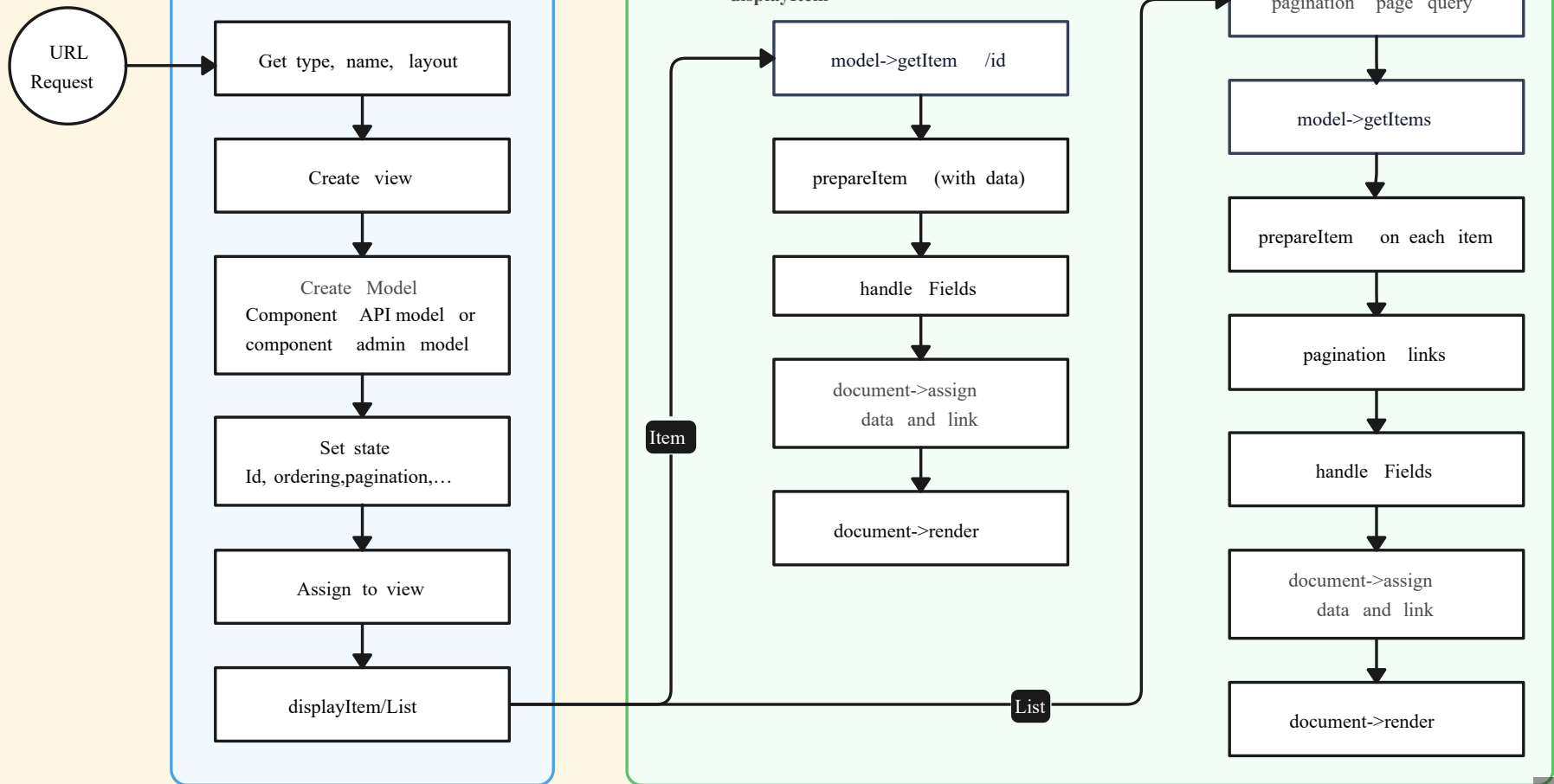
URL request to used model



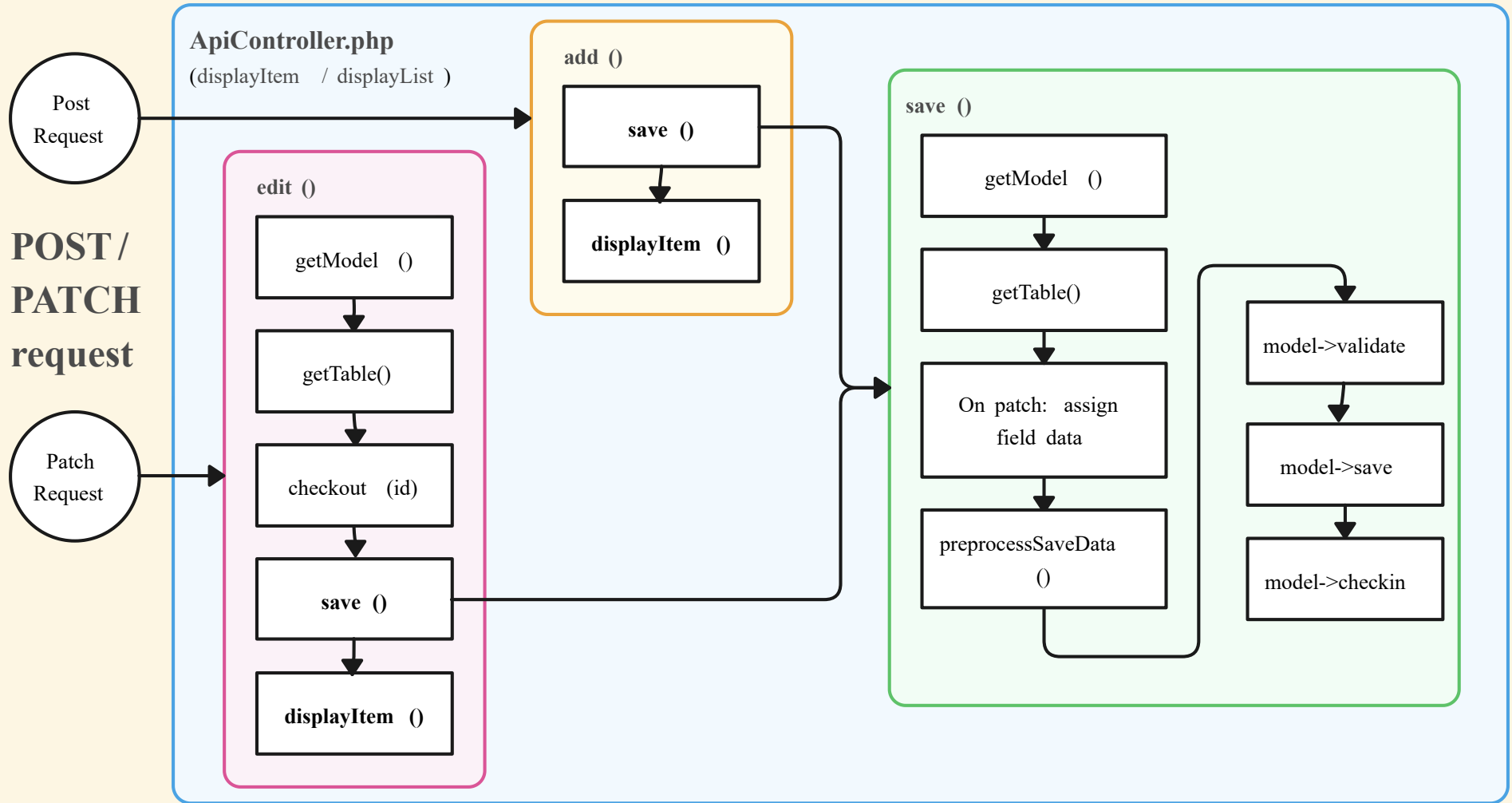
1. Controller function 'displayList' handled by component or joomla controller
2. Used model by API or administrator folder of component

Joomla API GET controller sequence

GETrequest



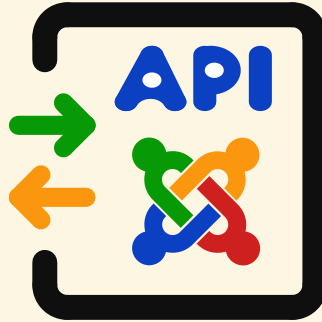
Joomla API POST/PATCH controller sequence



Extend J! API support with own code

- No standard table,
- Handle data before read/save
- Special needs ...

=>bypass the default behavior of the J! API



- Patch results by overriding base function but calling parent
- Handle parts/all on your own

Most Functions in the Joomla library API files

- **ApiController.php**
- **JsonApiView.php**
- **ApiMVCFactory.php**

can be overridden.

There is no special ApiModel.php file for API.
For 'API' models the standard joomla
administrator models are used.

ApiMVCFactory -> createTable, createModel

Functions to override

There are some empty parent functions provided to be overwritten from derived classes

Function prepareItem (jsonApi.php)

Example: The 'get' command to the component config variables has no **id** but the standard view function does expect an **id**.

Solution: In component **JsonApiView.php** override function **prepareItem(\$item)**

```
/**
 * Prepare item before render.
 * @param object $item The model item
 * @return object
 */
protected function prepareItem($item)
{
    // Config resources have no id.
    $item->id = '0';
    return $item;
}
```

Function preprocessSaveData (...controller.php)

Controller: protected function preprocessSaveData(array \$data): array

```
/**
 * Method to allow extended classes to manipulate the data to be saved for an extension.
 * @param array $data An array of input data.
 * @return array
 */
protected function preprocessSaveData(array $data): array
{
    foreach (FieldsHelper::getFields('com_contact.contact') as $field) {
        if (isset($data[$field->name])) {
            !isset($data['com_fields']) && $data['com_fields'] = [];
            $data['com_fields'][$field->name] = $data[$field->name];
            unset($data[$field->name]);
        }
    }

    if ($this->input->getMethod() === 'PATCH') {
        $data['rules'] = json_decode($data['rules'], true);
    }

    $data['client_id'] = $this->getClientIdFromInput();

    return $data;
}
```

This function is mainly used to handle fields, apply special variables handling on patch/post commands or patch data before save function is called

Expand by defining own Api model.php

Example: Route "GET component version ID"

The command will retrieve the component version ID via the API.

What do we need ?

In addition to defining the route,
only a **controller**, a **model**, and a **helper** are added.

"GET" component version ID (part 1)

Routes in webservice plugin

```
//--- version in db manifest -----  
$router->addRoutes(  
[  
    // version, createDate  
    new Route(['GET'], 'v1/secondhand/version', 'version.displayItem', [], $getDefaults),  
    new Route(['PATCH'], 'v1/secondhand/version', 'version.edit', [], $defaults),  
]);
```

"GET" component version ID (part 2)

New path structure with additional model and helper

```
1  com.secondhand\api\src
2      |
3      |--- Controller
4          |--- BooksController.php
5          |--- VersionController.php
6      |--- Helper
7          |--- ManifestHelper.php
8      |--- Model
9          |--- VersionModel.php
10     |--- View
11         |--- Books
12             |--- JsonapiView.php
13         |--- Version
14             |--- JsonapiView.php
```

New File VersionController.php

VersionController.php minimum

```
protected $contentType = 'version';  
protected $default_view = 'version';  
... edit() // see patch
```

! No additional code needed in controller for **GET**
as request data is handled in component API model

New File VersionModel.php

The model overrides getItem which is called by joomla API code

```
public function getItem()  
{  
    $oManifest = ManifestHelper::getDbManifest($this->componentName);  
    $oVersion->version      = $oManifest['version'];  
    $oVersion->creationDate = $oManifest['creationDate'];  
    return $oVersion;  
}
```

New File ManifestHelper.php

The ManifestHelper.php reads/writes
the versionId from/to the database.

Code see github repository

New File jsonApiView.php

Minimum display variable definition
and additional function prepareItem ()

```
1 class JsonapiView extends BaseApiView
2 {
3     protected $fieldsToRenderItem = [
4         'version',
5         'creationDate',
6     ];
7
8     protected function prepareItem($item)
9     {
10         $item->id = '0'; // version resources have no id.
11
12         return $item;
13     }
14 }
```

That is all that is required to create a 'getVersion' API code.

Code to write the versionID (patch)

The controller calls the edit function
of the parallel component API model

VersionController.php

```
1 public function edit() : ApiController
2 {
3     // all variables
4     $data = $this->input->json->getArray();
5
6     //--- Create the model -----
7
8     /** @var VersionModel $model */
9     $model = $this->getModel('Version', '', ['ignore_request' => true, 'state' => $this->modelState]);
10
11     $isSaved = $model->save($data);
12
13     return parent::displayItem('0');
14 }
```

The view (displayItem) is called afterwards with id=0 as it expects an table id

Save code in VersionModel.php

```
1 public function save(mixed $data = []) {  
2  
3     // New data  
4     $version = $data['version'];  
5     $creationDate = $data['creationDate'];  
6  
7     // Fetch old data  
8     $oManifest = ManifestHelper::getDbManifest($this->componentName);  
9  
10    // Assign data  
11    $oManifest['version'] = $version;  
12    $oManifest['creationDate'] = $creationDate;  
13  
14    // Save  
15    $isSaved = ManifestHelper::saveDbManifest($oManifest, $this->componentName);  
16  
17    return $isSaved;  
18 }
```

The ManifestHelper.php reads/writes the versionId from/to the database.

Summary

for the requests without the corresponding table:

Get item/items:

- Tell controller the name of content type and view
- Overwrite GetItem/GetItems in model

post/patch item/items:

- Overwrite edit in controller
- Create save function in model and call it in edit function

There may be similar approaches working. But this is the gist of it.

Use model from administrator part

Copy the model and helper to the component administrator path.
Change the namespace and you are ready to go ;-)

```
.../administrator/components/com_secondhand/src/Model/VersionModel.php  
.../administrator/components/com_secondhand/src/Helper/VersionModel.php
```

Possible overrides

Functions to override from ApiController.php in controller

```
public function displayItem($id = null);  
public function displayList();  
public function delete($id = null);  
public function add();  
public function edit();  
protected function save($recordKey = null);  
protected function allowEdit($data = [], $key = 'id');  
protected function allowAdd($data = []);  
protected function allowDelete(): bool;  
protected function preprocessSaveData(array $data): array;
```

Functions to override from JsonApiView.php in view

```
public function displayList(?array $items = null);  
public function displayItem($item = null);  
protected function prepareItem($item);  
protected function queryEncode($query);
```

How to manipulate the 'model' state

In the controller, the state parameters must be collected from the URL parameters or the JSON data.

```
public function displayItem($path = '')
{
    // Set the id as the parent sets it as int
    $this->modelState->set('id', $this->input->get('id', '', 'string'));
    return parent::displayItem();
}
```

When using the J! API model call the `$this->modelState` is assigned to the model.

Otherwise use

```
$this->modelState->set('data', $data);
// Create the model, ignoring request data so we can safely set the state in the request from the controller
$model = $this->getModel($modelName, '', ['ignore_request' => true, 'state' => $this->modelState]);
```

modelState collection

Examples of modelState use collected in joomla component API projects.
Seen in standard controller functions displayItems, displayList ...

```
public function displayList()
{
    $this->modelState->set('list.ordering', 'h.save_date');
    $this->modelState->set('list.direction', 'DESC');

    $this->modelState->set('filter.core', ($requestBool === 'true') ? '1' : '0');
    $this->modelState->set('filter.status', $this->input->get->get('status', null, null));
    $this->modelState->set('filter.type', $this->input->get->get('type', null, null));

    $this->modelState->set('filter.author_id', $filter->clean($apiFilterInfo['author'], 'INT'));
    $this->modelState->set('filter.category_id', $filter->clean($apiFilterInfo['category'], 'INT'));
    $this->modelState->set('filter.search', $filter->clean($apiFilterInfo['search'], 'STRING'));
    $this->modelState->set('filter.published', $filter->clean($apiFilterInfo['state'], 'INT'));
    $this->modelState->set('filter.featured', $filter->clean($apiFilterInfo['featured'], 'INT'));

    $this->modelState->set('type_alias', $this->getTypeAliasFromInput());
    $this->modelState->set('type_id', $this->getTypeIdFromInput());
    $this->modelState->set('item_id', $this->getTypeAliasFromInput() . '.' . $this->getItemIdFromInput());

    return parent::displayList();
}
```

Especially the filtering addresses the original table model in the administrator section. The model can be used like it is intended for the component (admin)

Example modelState:

URL/JSON

- \$param: param in url
- \$data: data by json object

```
1 public function displayItem($param = '')
2 {
3     //--- set state from given URL parameter -----
4     $param = $this->input->get('para', '', 'string');
5     $this->modelState->set('param', $param);
6
7     // --- set state from given json data -----
8
9     $data = $this->input->json->getArray();
10    $this->modelState->set('data', $data);
11
12    //--- display parameters -----
13
14    return parent::displayItem(0);
15 }
16
```

Example modelState:

Filter

Collect URL filter data and assign single expected variables to the modelState

```
public function displayList()
{
    $apiFilterInfo = $this->input->get('filter', [], 'array');
    $filter       = InputFilter::getInstance();

    if (\array_key_exists('author', $apiFilterInfo)) {
        $this->modelState->set('filter.author_id', $filter->clean($apiFilterInfo['author'], 'INT'));
    }

    if (\array_key_exists('category', $apiFilterInfo)) {
        $this->modelState->set('filter.category_id', $filter->clean($apiFilterInfo['category'], 'INT'));
    }

    ...

    return parent::displayList();
}
```

Example modelState:

Ordering/direction

Collect URL list data and assign single expected variables to the modelState

```
public function displayList()
{
    $apiListInfo = $this->input->get('list', [], 'array');
    if (\array_key_exists('ordering', $apiListInfo)) {
        $this->modelState->set('list.ordering', $filter->clean($apiListInfo['ordering'], 'STRING'));
    }
    if (\array_key_exists('direction', $apiListInfo)) {
        $this->modelState->set('list.direction', $filter->clean($apiListInfo['direction'], 'STRING'));
    }
    return parent::displayList();
}
```

findings

If a URL variable could not be found in input try post data

```
private function getExtensionFromInput()  
{  
    return $this->input->exists('extension') ?  
        $this->input->get('extension') : $this->input->post->get('extension');  
}  
  
private function getClientIdFromInput()  
{  
    return $this->input->exists('client_id') ?  
        $this->input->get('client_id') : $this->input->post->get('client_id');  
}  
  
private function getClientIdFromInput()  
{  
    return $this->input->exists('client_id') ? $this->input->get('client_id') : $this->input->post->get('client_id');
```

Name it "get 'something' FromInput () {...}"

Links to "Web service API"

- **Peter Martin: Using the Joomla Web Services API**

Very good overview:

<https://petermartin.nl/en/focus-on/joomla/web-services-api>

- **[manual@joomla.org/.../API](https://manual.joomla.org/.../API)**

Web Services API component code example

<https://manual.joomla.org/docs/web-services-api/json-response-format/>

- **J4.x: Adding an API to a Joomla Component:**

Plain top down showing the code needed for minimum API support in component, old J4 description

https://docs.joomla.org/J4.x:Adding_an_API_to_a_Joomla_Component

- **Communicate with the Joomla 4.x Web Services API**

External PHP scripts to access Web page per "Web service API"

<https://manual.joomla.org/docs/general-concepts/webservices>

- **Marc Dechèvre: Playing with the Joomla Web Services (API)**

(in Joomla 4 or 5 or 6)

<https://slides.woluweb.be/api>

- Checkout postman and bruno