模式

目录

- 概述
- JSON 模式
- Resource Schema
- Argument Schema
- 概述

概述

模式是一种数据，它告诉我们其他数据应如何组织。大多数数据库都实现了某种形式的模式，使我们能够以更结构化的方式处理数据。 WordPress REST API 使用 JSON Schema 来处理其数据的结构。您可以在不使用模式的情况下实现端点，但您将错过许多功能。这取决于您决定什么最适合您。

JSON 模式

首先，我们来简单谈谈 JSON。JSON 是一种人类可读的数据格式，类似于 JavaScript 对象。 JSON 代表 JavaScript 对象表示法。 JSON 正迅速普及，似乎正席卷数据结构领域。 WordPress REST API 使用一种称为 JSON 模式的特殊规范来处理 JSON。要了解更多关于 JSON 模式的信息，请查看 JSON 模式网站 和这篇更易于理解的 JSON 模式介绍 。模式为我们带来了许多好处：改进的测试、可发现性以及整体更好的结构。让我们看一个 JSON 数据块。

```
{
    "shouldBeArray": 'LOL definitely not an array',
    "shouldBeInteger": ['lolz', 'you', 'need', 'schema'],
    "shouldBeString": 123456789
}
```

JSON 解析器会毫无问题地处理这些数据，并且不会抱怨任何问题，因为它是有效的 JSON 。客户端和服务器对数据一无所知，也不知道预期什么，它们只看到 JSON 。通过实现模式，我们实际上可以简化代码库。模式将有助于更好地构建我们的数据，使我们的应用程序能更轻松地理解与 WordPress REST API 的交互。 WordPress REST API 并不强制您使用模式，但鼓励使用。有两种方式将模式数据整合到 API 中：资源的模式和已注册参数的模式。

Resource Schema

The schema for a resource indicates what fields are present for a particular object. When we register our routes we can also specify the resource schema for the route. Let’s look at what a simple comment schema might look like in a PHP representation of JSON schema.

```
// Register our routes.
function prefix_register_my_comment_route() {
	register_rest_route( 'my-namespace/v1', '/comments', array(
		// Notice how we are registering multiple endpoints the 'schema' equates to an OPTIONS request.
		array(
			'methods'  => 'GET',
			'callback' => 'prefix_get_comment_sample',
		),
		// Register our schema callback.
		'schema' => 'prefix_get_comment_schema',
	) );
}

add_action( 'rest_api_init', 'prefix_register_my_comment_route' );

/**
 * Grabs the five most recent comments and outputs them as a rest response.
 *
 * @param WP_REST_Request $request Current request.
 */
function prefix_get_comment_sample( $request ) {
	$args = array(
		'post_per_page' => 5,
	);
	$comments = get_comments( $args );
```

```
    $data = array();

    if ( empty( $comments ) ) {
        return rest_ensure_response( $data );
    }

    foreach ( $comments as $comment ) {
        $response = prefix_rest_prepare_comment( $comment, $request );
        $data[] = prefix_prepare_for_collection( $response );
    }

    // Return all of our comment response data.
    return rest_ensure_response( $data );
}

/**
 * Matches the comment data to the schema we want.
 *
 * @param WP_Comment $comment The comment object whose response is being prepared.
 */
function prefix_rest_prepare_comment( $comment, $request ) {
    $comment_data = array();

    $schema = prefix_get_comment_schema( $request );

    // We are also renaming the fields to more understandable names.
    if ( isset( $schema['properties']['id'] ) ) {
        $comment_data['id'] = (int) $comment->comment_id;
    }

    if ( isset( $schema['properties']['author'] ) ) {
        $comment_data['author'] = (int) $comment->user_id;
    }

    if ( isset( $schema['properties']['content'] ) ) {
        $comment_data['content'] = apply_filters( 'comment_text', $comment->comment_content, $comment
    );
    }

    return rest_ensure_response( $comment_data );
```

```
}

/**
 * Prepare a response for inserting into a collection of responses.
 *
 * This is copied from WP_REST_Controller class in the WP REST API v2 plugin.
 *
 * @param WP_REST_Response $response Response object.
 * @return array Response data, ready for insertion into collection data.
 */
function prefix_prepare_for_collection( $response ) {
	if ( ! ( $response instanceof WP_REST_Response ) ) {
		return $response;
	}

	$data = (array) $response->get_data();
	$server = rest_get_server();

	if ( method_exists( $server, 'get_compact_response_links' ) ) {
		$links = call_user_func( array( $server, 'get_compact_response_links' ), $response );
	} else {
		$links = call_user_func( array( $server, 'get_response_links' ), $response );
	}

	if ( ! empty( $links ) ) {
		$data['_links'] = $links;
	}

	return $data;
}

/**
 * Get our sample schema for comments.
 *
 * @param WP_REST_Request $request Current request.
 */
function prefix_get_comment_schema( $request ) {
	$schema = array(
		// This tells the spec of JSON Schema we are using which is draft 4.
		'$schema'              => 'http://json-schema.org/draft-04/schema#',
```

```
    // The title property marks the identity of the resource.
    'title'                => 'comment',
    'type'                 => 'object',
    // In JSON Schema you can specify object properties in the properties attribute.
    'properties'           => array(
      'id' => array(
        'description'  => esc_html__( 'Unique identifier for the object.', 'my-textdomain' ),
        'type'         => 'integer',
        'context'      => array( 'view', 'edit', 'embed' ),
        'readonly'     => true,
      ),
      'author' => array(
        'description'  => esc_html__( 'The id of the user object, if author was a user.', 'my-
textdomain' ),
        'type'         => 'integer',
      ),
      'content' => array(
        'description'  => esc_html__( 'The content for the object.', 'my-textdomain' ),
        'type'         => 'string',
      ),
    ),
  );

  return $schema;
}
```

If you notice, each comment resource now matches up to our schema that we specified. We made this switch in `prefix_rest_prepare_comment()`. By creating schema for our resources, we can now view this schema by making `OPTIONS` requests. Why is this useful? If we wanted other languages, JavaScript for example, to interpret our data and validate the data from our endpoint, JavaScript would need to know how our data is structured. When we provide schema, we open the doors for other authors, and ourselves, to build on top of our endpoints in a consistent manner.

Schema provides machine readable data, so potentially anything that can read JSON can understand what kind of data it is looking at. When we look at the API index by making a `GET` request to `https://ourawesomesite.com/wp-json/`, we are returned the schema of our API, enabling others to write client libraries to interpret our data. This process of reading schema data is known as discovery. When we have provided schema for a resource we make that resource discoverable via `OPTIONS` requests to that route. Exposing resource schema is only one part of our schema puzzle. We also want to use schema for our registered arguments.

Argument Schema

When we register request arguments for an endpoint, we can also use JSON Schema to provide us data about what the arguments should be. This enables us to write validation libraries that can be reused as our endpoints expand. Schema is more work upfront, but if you are going to write a production application that will grow, you should definitely consider using schema. Let’s look at an example of using argument schema and validation.

```
// Register our routes.
function prefix_register_my_arg_route() {
	register_rest_route( 'my-namespace/v1', '/schema-arg', array(
		// Here we register our endpoint.
		array(
			'methods'  => 'GET',
			'callback' => 'prefix_get_item',
			'args' => prefix_get_endpoint_args(),
		),
	) );
}

// Hook registration into 'rest_api_init' hook.
add_action( 'rest_api_init', 'prefix_register_my_arg_route' );

/**
 * Returns the request argument `my-arg` as a rest response.
 *
 * @param WP_REST_Request $request Current request.
 */
function prefix_get_item( $request ) {
	// If we didn't use required in the schema this would throw an error when my arg is not set.
	return rest_ensure_response( $request['my-arg'] );
}

/**
 * Get the argument schema for this example endpoint.
 */
function prefix_get_endpoint_args() {
	$args = array();
```

```
    // Here we add our PHP representation of JSON Schema.
    $args['my-arg'] = array(
        'description'       => esc_html__( 'This is the argument our endpoint returns.', 'my-
textdomain' ),
        'type'              => 'string',
        'validate_callback' => 'prefix_validate_my_arg',
        'sanitize_callback' => 'prefix_sanitize_my_arg',
        'required'          => true,
    );

    return $args;
}

/**
 * Our validation callback for `my-arg` parameter.
 *
 * @param mixed           $value   Value of the my-arg parameter.
 * @param WP_REST_Request $request Current request object.
 * @param string          $param   The name of the parameter in this case, 'my-arg'.
 */
function prefix_validate_my_arg( $value, $request, $param ) {
    $attributes = $request->get_attributes();

    if ( isset( $attributes['args'][ $param ] ) ) {
        $argument = $attributes['args'][ $param ];
        // Check to make sure our argument is a string.
        if ( 'string' === $argument['type'] && ! is_string( $value ) ) {
            return new WP_Error( 'rest_invalid_param', sprintf( esc_html__( '%1$s is not of type %2$s',
'my-textdomain' ), $param, 'string' ), array( 'status' => 400 ) );
        }
    } else {
        // This code won't execute because we have specified this argument as required.
        // If we reused this validation callback and did not have required args then this would fire.
        return new WP_Error( 'rest_invalid_param', sprintf( esc_html__( '%s was not registered as a
request argument.', 'my-textdomain' ), $param ), array( 'status' => 400 ) );
    }

    // If we got this far then the data is valid.
    return true;
}
```

```
/**
 * Our santization callback for `my-arg` parameter.
 *
 * @param mixed           $value   Value of the my-arg parameter.
 * @param WP_REST_Request $request Current request object.
 * @param string          $param   The name of the parameter in this case, 'my-arg'.
 */
function prefix_sanitize_my_arg( $value, $request, $param ) {
	$attributes = $request->get_attributes();

	if ( isset( $attributes['args'][ $param ] ) ) {
		$argument = $attributes['args'][ $param ];
		// Check to make sure our argument is a string.
		if ( 'string' === $argument['type'] ) {
			return sanitize_text_field( $value );
		}
	} else {
		// This code won't execute because we have specified this argument as required.
		// If we reused this validation callback and did not have required args then this would fire.
		return new WP_Error( 'rest_invalid_param', sprintf( esc_html__( '%s was not registered as a request argument.', 'my-textdomain' ), $param ), array( 'status' => 400 ) );
	}

	// If we got this far then something went wrong don't use user input.
	return new WP_Error( 'rest_api_sad', esc_html__( 'Something went terribly wrong.', 'my-textdomain' ), array( 'status' => 500 ) );
}
```

In the example above we have abstracted away from using the `'my-arg'` name. We can use these validation and sanitizing functions for any other argument that should be a string we have specified schema for. As your codebase and endpoints grow, schema will help keep your code lightweight and maintainable. Without schema you can validate and sanitize, however it will be more difficult to keep track of which functions should be validating what. By adding schema to request arguments we can also expose our argument schema to clients, so validation libraries can be built client side which can help performance by preventing invalid requests from ever being sent to the API.

[info]If you are uncomfortable with using schema, it is still possible to have validate/sanitize callbacks for each of your arguments, and in some cases it will make the most sense to do a custom validation.[/info]

[illegible]

[illegible]

[illegible] #2

[illegible] 2026-03-26 17:08:39 UTC

[illegible] 2026-03-26 17:08:39 UTC