0019-graphql-mutations

- uri: "/docs/graphql-mutations/" title: "Mutations with GraphQL"
- Basic Mutation Structure
  - Simple Mutations
  - Named Operations
  - Using Variables
  - HTTP Method Requirements
  - Authentication Requirements
  - Common Authentication Errors
- Common Mutation Patterns
  - Post Operations
    - Creating Posts
    - Updating Posts
    - Deleting Posts
    - Managing Post Meta
  - Media Operations
    - Creating Media Items
    - Updating Media Items
    - Deleting Media Items
  - User Operations
    - Creating Users
    - Updating Users
    - Deleting Users
    - Managing User Meta
  - Comment Operations
    - Creating Comments
    - Updating Comments
    - Deleting Comments
    - Moderating Comments
- Working with Mutations

    - Understanding Input Types
    - Handling Responses
    - Error Handling
    - Optimistic Updates
- Best Practices
    - Input Validation
    - Security Considerations
    - Performance Tips
    - Testing Mutations

---

uri: "/docs/graphql-mutations/" title: "Mutations with GraphQL"

Building on the concepts from our Querying with GraphQL guide, this guide covers how to modify data using GraphQL mutations.

Basic Mutation Structure

Simple Mutations

The most basic mutation structure:

```
mutation {
  createPost(input: {
    title: "Hello World"
    content: "This is my first post"
    status: PUBLISH
  }) {
    post {
      id
```

```
      title
    }
  }
}
```

Named Operations

Like queries, it's best practice to use named operations:

```
# ✅ Better: Using the mutation keyword and operation name
mutation CreateNewPost {
  createPost(input: {
    title: "Hello World"
    content: "This is my first post"
    status: PUBLISH
  }) {
    post {
      id
      title
    }
  }
}
```

Using Variables

Variables are especially important in mutations to handle user input safely:

```
mutation CreateNewPost($input: CreatePostInput!) {
  createPost(input: $input) {
    post {
      id
      title
    }
  }
}
```

Variables would be passed like:

```
{
  "input": {
    "title": "Hello World",
    "content": "This is my first post",
    "status": "PUBLISH"
  }
}
```

HTTP Method Requirements

> Mutations can only be executed using HTTP POST requests. Attempting to execute mutations over GET requests will result in an error.

```
// □ Correct: Using POST for mutations
fetch('/graphql', {
  method: 'POST',  // Required for mutations
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    query: `mutation { ... }`,
    variables: { }
  })
})
```

Common errors when using GET requests:

```
{
  "errors": [
    {
      "message": "GET supports only query operation",
      "category": "request"
    }
  ]
}
```

> While queries can be executed over both GET and POST requests, mutations are restricted to POST requests for security reasons and to follow proper HTTP semantics.

Authentication Requirements

> Most mutations in WPGraphQL require authentication and proper user capabilities. Without proper authentication, you'll receive "User is not authorized" errors.

For example, creating a post requires a user to have the `publish_posts` capability:

```
mutation CreatePost($input: CreatePostInput!) {
  createPost(input: $input) {
    post {
      id
      title
    }
  }
}
```

Some mutations that typically don't require authentication:

- `login`: Authenticating users (provided by the WPGraphQL JWT Authentication plugin and others)
- `registerUser`: When registration is enabled
- `createComment`: When comments are open
- `submitForm`: For form submissions (when enabled - not a core mutation, but some extension plugins have form submission mutations that don't require auth)

For most other mutations:

1. Ensure you're authenticated
2. Verify the user has proper capabilities
3. Include authentication headers with your request following documentation for whatever authentication method you're using

> See our Authentication and Authorization guide for details on how to authenticate your requests.

Common Authentication Errors

```
mutation UpdatePost($input: UpdatePostInput!) {
  updatePost(input: $input) {
    post {
      id
      title
    }
  }
}
```

Response when not authenticated:

```
{
  "errors": [
    {
      "message": "Sorry, you are not allowed to update a post",
      "category": "user"
    }
  ]
}
```

Common Mutation Patterns

Post Operations

Creating Posts

```
mutation CreatePost($input: CreatePostInput!) {
  createPost(input: $input) {
    post {
      id
      title
      status
      uri
    }
```

```
  }
}
```

Variables:

```
{
  "input": {
    "title": "My New Post",
    "content": "Post content here...",
    "status": "PUBLISH",
    "categories": {
      "nodes": [
        { "id": "category-id-here" }
      ]
    },
    "tags": {
      "nodes": [
        { "id": "tag-id-here" }
      ]
    }
  }
}
```

Example successful response:

```
{
  "data": {
    "createPost": {
      "post": {
        "id": "cG9zdDo1",
        "title": "My New Post",
        "status": "PUBLISH",
        "uri": "/my-new-post"
      }
    }
  }
}
```

Example error response:

```
{
  "errors": [
    {
      "message": "Sorry, you are not allowed to create posts",
      "category": "user"
    }
  ]
}
```

Updating Posts

```
mutation UpdatePost($id: ID!, $input: UpdatePostInput!) {
  updatePost(input: {
    id: $id,
    # Only include fields you want to update
    title: $input.title
    content: $input.content
  }) {
    post {
      id
      title
      modified # When the post was last modified
    }
  }
}
```

Deleting Posts

```
mutation DeletePost($id: ID!) {
  deletePost(input: {
    id: $id,
    # Optional: force delete instead of moving to trash
    forceDelete: true
  }) {
    # Returns the deleted post
    post {
      id
      title
    }
    # Was the post deleted?
```

```
    deleted
  }
}
```

> By default, deleting a post moves it to the trash. Use `forceDelete: true` to permanently delete.

Managing Post Meta

```
mutation UpdatePostMeta($id: ID!) {
  updatePost(input: {
    id: $id
    # Update custom fields
    customFields: [
      { key: "my_field", value: "new value" }
    ]
  }) {
    post {
      id
      # Query the updated meta
      customFields {
        key
        value
      }
    }
  }
}
```

Media Operations

Creating Media Items

> File uploads are not handled directly through GraphQL mutations. The file must first be uploaded to WordPress using the REST API or other methods, then you can create a MediaItem with the uploaded file's details.

```
mutation CreateMediaItem($input: CreateMediaItemInput!) {
  createMediaItem(input: $input) {
    mediaItem {
      id
      title
      altText
      sourceUrl
    }
  }
}
```

Variables:

```
{
  "input": {
    "title": "My Image",
    "altText": "Description of image",
    "description": "Detailed description here",
    "filePath": "/2024/03/my-image.jpg",
    "status": "PUBLISH"
  }
}
```

Updating Media Items

```
mutation UpdateMediaItem($id: ID!, $title: String, $altText: String) {
  updateMediaItem(input: {
    id: $id
    title: $title
    altText: $altText
  }) {
    mediaItem {
      id
      title
      altText
      modified
    }
  }
}
```

Deleting Media Items

```
mutation DeleteMediaItem($id: ID!) {
  deleteMediaItem(input: {
    id: $id
    forceDelete: true
  }) {
    mediaItem {
      id
      title
    }
  }
}
```

> When deleting media items, consider:
>
> - Files will be deleted from the server when `forceDelete` is true
> - Referenced files in content may break if not updated
> - Ensure proper backup procedures are in place

User Operations

Creating Users

```
mutation CreateUser($input: CreateUserInput!) {
  createUser(input: $input) {
    user {
      id
      databaseId
      username
      email
      firstName
      lastName
      roles {
        nodes {
          name
        }
      }
```

```
    }
  }
}
```

Variables:

```
{
  "input": {
    "username": "newuser",
    "email": "user@example.com",
    "firstName": "John",
    "lastName": "Doe",
    "roles": ["subscriber"],
    "password": "secure_password_here"
  }
}
```

Updating Users

```
mutation UpdateUser($id: ID!, $input: UpdateUserInput!) {
  updateUser(input: {
    id: $id
    firstName: $input.firstName
    lastName: $input.lastName
    description: $input.description
  }) {
    user {
      id
      firstName
      lastName
      description
      modified
    }
  }
}
```

Deleting Users

```
mutation DeleteUser($id: ID!) {
  deleteUser(input: {
```

```
    id: $id
    reassignPosts: null  # Optional: User ID to reassign content to
  }) {
    user {
      id
      databaseId
    }
  }
}
```

> When deleting users:
>
> - Consider what happens to their content
> - Use `reassignPosts` to transfer content to another user
> - Ensure proper user capabilities (`delete_users`)
> - Cannot delete your own user account

Managing User Meta

```
mutation UpdateUserMeta($id: ID!, $input: UpdateUserInput!) {
  updateUser(input: {
    id: $id
    # Update custom fields
    customFields: [
      { key: "user_preference", value: "dark_mode" }
    ]
  }) {
    user {
      id
      customFields {
        key
        value
      }
    }
  }
}
```

Comment Operations

Creating Comments

```
mutation CreateComment($input: CreateCommentInput!) {
  createComment(input: $input) {
    comment {
      id
      content
      date
      status
      author {
        node {
          name
          email
        }
      }
    }
  }
}
```

Variables:

```
{
  "input": {
    "commentOn": 123,           # Post database ID to comment on
    "content": "Great post!",
    "author": "John Smith",     # Required if not authenticated
    "authorEmail": "john@example.com",  # Required if not authenticated
    "authorUrl": "https://example.com"  # Optional
  }
}
```

- Authenticated users don't need to provide author details
- Comments may be held for moderation based on WordPress settings
- The post must have comments open to accept new comments

Updating Comments

```
mutation UpdateComment($id: ID!, $content: String) {
  updateComment(input: {
```

```
    id: $id
    content: $content
  }) {
    comment {
      id
      content
      modified
    }
  }
}
```

Deleting Comments

```
mutation DeleteComment($id: ID!) {
  deleteComment(input: {
    id: $id
    forceDelete: true
  }) {
    comment {
      id
      databaseId
    }
  }
}
```

Moderating Comments

```
mutation UpdateCommentStatus($id: ID!, $status: CommentStatusEnum!) {
  updateComment(input: {
    id: $id
    status: $status
  }) {
    comment {
      id
      status
    }
  }
}
```

Variables for moderation:

```
{
  "id": "commentID",
  "status": "APPROVE"  # APPROVE, HOLD, SPAM, or TRASH
}
```

> Comment moderation requires proper capabilities (`moderate_comments`). Regular users can typically only:
>
> - Create new comments (when allowed)
> - Edit their own comments
> - Delete their own comments

Working with Mutations

Understanding Input Types

Each mutation accepts specific input types that define what data can be provided:

```
# Exploring input type fields
query GetInputFields {
  __type(name: "CreatePostInput") {
    inputFields {
      name
      type {
        name
        kind
      }
      description
    }
  }
}
```

Input types are strictly typed:

```
mutation CreatePost($input: CreatePostInput!) {
  createPost(input: $input) {
    post {
```

```
      id
    }
  }
}
```

Variables must match the input type:

```
{
  "input": {
    "title": "My Post",     # String
    "status": "PUBLISH",    # PostStatusEnum
    "password": null,       # Optional String
    "commentStatus": "OPEN" # Optional CommentStatusEnum
  }
}
```

Handling Responses

Mutations return specific types that include:

1. The modified object
2. Any additional fields specific to the mutation

```
mutation UpdatePost($id: ID!, $title: String) {
  updatePost(input: { id: $id, title: $title }) {
    # The modified post
    post {
      id
      title
      modified
    }
    # Check if the operation was successful
    success
  }
}
```

Error Handling

GraphQL errors can occur at different levels:

1. **Syntax Errors**

```
{
  "errors": [
    {
      "message": "Syntax Error: Expected Name, found <EOF>",
      "category": "graphql"
    }
  ]
}
```

2. **Validation Errors**

```
{
  "errors": [
    {
      "message": "Variable \"$input\" of required type \"CreatePostInput!\" was not
provided.",
      "category": "validation"
    }
  ]
}
```

3. **Authorization Errors**

```
{
  "errors": [
    {
      "message": "Sorry, you are not allowed to create posts",
      "category": "user"
    }
  ]
}
```

> Always handle errors in your application code. A successful HTTP response (200) might still contain GraphQL errors.

Optimistic Updates

When building user interfaces, you can update the UI before the mutation completes:

```
function updatePost({ id, title }) {
  // 1. Get current data
  const originalPost = cache.get(id);

  // 2. Optimistically update UI
  cache.update(id, { title });

  // 3. Perform mutation
  mutation({
    variables: { id, title }
  }).catch(error => {
    // 4. Revert on error
    cache.update(id, originalPost);
    showError(error);
  });
}
```

This provides a better user experience by:

- Showing immediate feedback
- Handling offline scenarios
- Gracefully recovering from errors

Best Practices

Input Validation

1. **Validate Before Sending**

```
function validatePostInput(input) {
  const errors = {};

  if (!input.title?.trim()) {
    errors.title = "Title is required";
  }

  if (input.title?.length > 200) {
```

```
    errors.title = "Title must be less than 200 characters";
  }

  return Object.keys(errors).length ? errors : null;
}
```

2. **Use Proper Types**

```
# ❌ Avoid: Using improper types for variables
mutation UpdatePost($id: ID!, $status: String) {
  updatePost(input: { id: $id, status: $status })
}

# ✅ Better: Use specific types defined in the schema
mutation UpdatePost($id: ID!, $status: PostStatusEnum!) {
  updatePost(input: { id: $id, status: $status })
}
```

Security Considerations

1. **Sanitize User Input**

```
// ❌ Avoid: Direct user input
mutation.updatePost({
  content: userInput
});

// ✅ Better: Sanitize input
mutation.updatePost({
  content: sanitizeHtml(userInput, allowedTags)
});
```

2. **Limit Query Depth**

```
# ❌ Avoid: Deep nested queries in mutations
mutation CreatePost($input: CreatePostInput!) {
  createPost(input: $input) {
    post {
      author {
```

```
        posts {
          nodes {
            author {
              posts {
                nodes {
                  # Too deep!
                }
              }
            }
          }
        }
      }
    }
  }
}

# ✅ Better: Request only what you need
mutation CreatePost($input: CreatePostInput!) {
  createPost(input: $input) {
    post {
      id
      title
      author {
        node {
          name
        }
      }
    }
  }
}
```

Performance Tips

1. **Batch Related Changes**

```
# ❌ Avoid: Multiple separate mutations
mutation UpdatePost($id: ID!) {
  updatePost(input: { id: $id, title: "New Title" }) {
    post { id }
```

```
  }
}
mutation UpdateMeta($id: ID!) {
  updatePost(input: { id: $id, customFields: [{ key: "meta", value: "value" }] }) {
    post { id }
  }
}

# □ Better: Single mutation with all changes
mutation UpdatePost($id: ID!) {
  updatePost(input: {
    id: $id
    title: "New Title"

    # NOTE: This is a made up field for the sake of example
    customFields: [{ key: "meta", value: "value" }]
  }) {
    post {
      id
      title

      # NOTE: This is a made up field for the sake of example
      customFields {
        key
        value
      }
    }
  }
}
```

2. **Select Specific Fields**

```
# □ Avoid: Over-fetching
mutation UpdatePost($input: UpdatePostInput!) {
  updatePost(input: $input) {
    post {
      # Don't fetch everything!
      ...AllPostFields
    }
  }
```

```
}

# ✅ Better: Request specific fields
mutation UpdatePost($input: UpdatePostInput!) {
  updatePost(input: $input) {
    post {
      id
      title
      modified
    }
  }
}
```

Testing Mutations

1. **Test Input Validation**

```
it('validates required fields', async () => {
  const { errors } = await mutate({
    mutation: CREATE_POST,
    variables: {
      input: { /* missing required fields */ }
    }
  });

  expect(errors[0].message).toContain('required');
});
```

2. **Test Authorization**

```
it('requires authentication', async () => {
  const { errors } = await mutate({
    mutation: UPDATE_POST,
    variables: {
      input: { /* ... */ }
    }
  });

  expect(errors[0].category).toBe('user');
```

```
});
```

3. **Test Success Cases**

```
it('creates post with valid input', async () => {
  const { data } = await mutate({
    mutation: CREATE_POST,
    variables: {
      input: {
        title: "Test Post",
        status: "PUBLISH"
      }
    }
  });

  expect(data.createPost.post.title).toBe("Test Post");
});
```

> Consider using a testing environment with predictable data and disabled webhooks/side effects for reliable tests.

□□□ #2
□ □□□ □□□ 2026-08-29 23:36:26 UTC
□ □□□ □□□ 2026-08-29 23:36:26 UTC