Johnny Mullaney

Solutions Architect & Platinum Optimizely MVP – First Three Things Ltd (FTT). Dublin, Ireland. E-Commerce, Technical and Product Strategy, .Net, Azure.

How Optimizely MCP Learns Your CMS (and Remembers It)

In Part 1, I introduced the "discovery-first" idea—an MCP that can plug into any SaaS CMS and learn how it's structured on its own.

This post gets into the details: how the MCP discovers your schema, builds a usable map of it, and remembers what it learns so that subsequent requests feel instant.

Discovery – asking the CMS about itself

When the MCP connects to a CMS for the first time, it doesn't guess what types exist. It introspects the CMS's GraphQL API using the standard GraphQL introspection query.

In code, that looks like this:

```
// src/clients/graph-client.ts
import { getIntrospectionQuery, IntrospectionQuery } from 'graphql';

async introspect(): Promise<IntrospectionQuery> {
  return await this.query<IntrospectionQuery>(
    getIntrospectionQuery(),
    undefined,
    {
      cacheKey: 'graphql:introspection',
      cacheTtl: 3600 // 1 hour cache
    }
  );
}
```

getIntrospectionQuery() is the full schema introspection spec from the GraphQL reference implementation.

It doesn't just fetch types and fields — it returns *everything*: object types, interfaces, enums, input objects, directives, and even nested type references up to nine levels deep.

That richness matters, because MCP needs to reason about structures like [BlockData!]! or detect which types implement _IContent or _IComponent.

Building the type map — turning raw introspection into something usable

The introspection response can be huge. It's effectively the entire CMS schema in JSON form, describing hundreds of types and relationships. By itself, that's unwieldy. MCP needs a faster way to navigate it.

That's where the **type map** comes in.

The type map is a simple but powerful lookup table that indexes every discovered type by name.
It lets MCP jump directly to any type's details without re-parsing the whole schema, which is what enables features like:

- Identifying which types represent content or components.
- Traversing relationships between nested objects.
- Dynamically generating valid GraphQL queries without hardcoding templates.

Here's the code that builds it:

```
// src/logic/graph/schema-introspector.ts
async initialize(): Promise<void> {
  if (this.schema) return;

  // Fetch and cache the full schema
  this.schema = await withCache(
    'graphql:schema:full',
    () => this.client.introspect(),
    3600
  );

  // Index all types for fast lookup
  this.schema.__schema.types.forEach(type => {
    this.typeMap.set(type.name, type);
  });

  // Identify root query type for future lookups
  const queryType = this.typeMap.get(this.schema.__schema.queryType.name);
  if (queryType && queryType.kind === 'OBJECT') {
    this.queryTypeInfo = this.extractTypeInfo(queryType);
  }

  this.logger.info('Schema introspection completed', {
    typeCount: this.schema.__schema.types.length,
    queryFields: this.queryTypeInfo?.fields?.length || 0
  });
}
```

By the end of this step, MCP knows how your CMS is shaped: which types exist, how they connect, and what each field exposes.
That's the knowledge it uses to generate safe queries on the fly.

Caching — learning once, remembering fast

Once the MCP discovers your schema, it doesn't need to do it again. Full GraphQL introspection can take a second or more, so the server layers its caches to keep things instant after the first call.

At a high level, there are three cache tiers:

1. **Base Cache** – a simple in-memory key/value store with a 5-minute TTL used across tools and logic.
2. **Discovery Cache** – holds schema introspection and type maps (TTL 5–60 min) and automatically invalidates when the schema version changes.
3. **Fragment Cache** – stores generated GraphQL fragments in memory and on disk, surviving restarts and invalidating on schema change.

```
// Simplified Base Cache
const cache = new Map();
export function get(key) {
  const e = cache.get(key);
  return e && Date.now() - e.ts < e.ttl ? e.val : null;
}
export function set(key, val, ttl) {
  cache.set(key, { val, ts: Date.now(), ttl });
}
```

Example — Getting an article page

This example shows how MCP retrieves a standard content page like `ArticlePage` or `StandardPage` — not a Visual Builder page which we'll cover in the next part of the series.

```
// User perspective – Claude or an AI client calls:
await get({ identifier: "/" });                          // homepage
await get({ identifier: "/articles/my-article/" });    // by path
await get({ identifier: "Getting Started Guide" });    // by search
```

Here's what actually happens behind the scenes

```
// 1. Initialize schema (cached after first call)
const introspector = new SchemaIntrospector(graphClient);
await introspector.initialize(); // runs getIntrospectionQuery()
```

```
// 2. Detect identifier type
const strategy = this.detectIdentifierType(identifier); // e.g. "path"

// 3. Find the content
const foundContent = await this.findContent(identifier, strategy, locale);
const contentType = foundContent.contentType; // e.g. "ArticlePage"

// 4. Generate or reuse a fragment
let fragment = await fragmentCache.getCachedFragment(contentType);
if (!fragment) {
  fragment = await fragmentGenerator.generateFragment(contentType, {
    maxDepth: 2,
    includeBlocks: true
  });
  await fragmentCache.setCachedFragment(contentType, fragment);
}

// 5. Build query and execute
const fullQuery = `
  ${fragment}
  query GetFullContent($key: String!) {
    _Content(where: { _metadata: { key: { eq: $key } } }) {
      items {
        _metadata {
          key displayName types url { default hierarchical }
          published lastModified status
        }
        ...${contentType}Fragment
      }
    }
  }
`;
const data = await graphClient.query(fullQuery, { key: foundContent.key });
```

Cold Start:

```
[INFO] Schema introspection completed (203 types, 1247 ms)
[DEBUG] Cache miss: fragment:ArticlePage
[INFO] Generating fragment for ArticlePage (124 ms)
[INFO] Query executed successfully (287 ms)
```

Warm cache:

```
[DEBUG] Cache hit: graphql:schema:full
[DEBUG] Cache hit: fragment:ArticlePage
[INFO] Query executed successfully (78 ms)
```

Example Response:

```
{
  "_metadata": {
    "key": "f3e8ef7f63ac45758a1dca8fbbde8d82",
    "displayName": "Getting Started with MCP",
    "types": ["ArticlePage", "_Page", "_Content"],
    "url": {
      "default": "/articles/getting-started/",
      "hierarchical": "/articles/getting-started/"
    },
    "published": "2024-01-15T10:30:00Z",
    "lastModified": "2024-01-20T14:22:00Z",
    "status": "Published"
  },
  "Title": "Getting Started with MCP",
  "Heading": "Your Guide to Model Context Protocol",
  "Body": { "html": "<p>This guide will help you…</p>" },
  "PromoImage": { "url": { "default": "https://cdn.example.com/mcp.jpg" } },
  "PublishDate": "2024-01-15T00:00:00Z",
  "SeoSettings": {
    "MetaTitle": "Getting Started with MCP | Developer Guide",
    "MetaDescription": "Learn how to integrate Model Context Protocol…"
  }
}
```

Next up

In **Part 3**, we'll dig into **Visual Builder** pages — the nested composition layer that makes discovery more challenging and more interesting.

That's where our Optimizely MCP shifts from "understanding structure" to "understanding composition."