

Integrate SQL solutions with Azure services

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/1-introduction>

Introduction

Completed

Modern applications rarely interact with databases through direct SQL connections. Instead, they consume data through REST and GraphQL APIs that provide flexibility, security, and standardization. Building these API layers traditionally requires significant backend development effort—writing controllers, mapping data models, handling authentication, and managing deployment infrastructure.

Data API Builder solves this challenge by automatically generating REST and GraphQL endpoints from your database schema. With a single configuration file, you can expose tables, views, and stored procedures through secure, scalable APIs without writing backend code.

In this module, you learn how to:

- Create and configure Data API Builder configuration files for SQL databases
- Define entities with field mappings, caching, pagination, and filtering
- Configure REST and GraphQL endpoints for different client needs
- Expose views, stored procedures, and define GraphQL relationships
- Explore deployment options for Data API Builder, including Azure Container Apps, App Service, and Static Web Apps
- Set up Azure Monitor with Application Insights for API observability
- Handle database changes using Change Data Capture, Azure Functions, and Change Event Streaming

Prerequisites

- Experience with SQL Server, Azure SQL Database, or SQL databases in Microsoft Fabric
- Familiarity with T-SQL for creating tables, views, and stored procedures
- Basic understanding of REST APIs and JSON
- Access to an Azure subscription for deployment exercises
- Docker Desktop installed for local development (optional)

2. Create configuration files for Data API Builder

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/2-create-configuration-files-data-api-builder>

Create configuration files for Data API Builder

Completed

Data API Builder (DAB) is a cross-platform, open-source engine that creates modern REST and GraphQL endpoints for your database without requiring you to write custom code. With a single configuration file, you can expose your database objects through secure, scalable APIs that support authentication, authorization, and caching.

If you work with Azure SQL or SQL databases in Microsoft Fabric, you've likely faced requests to make your data accessible to application developers. Building custom API layers takes time and adds maintenance overhead. Data API Builder eliminates that work by generating APIs directly from your database schema.

Understand the Data API Builder configuration file

Everything in Data API Builder starts with a configuration file. This JSON file tells DAB how to connect to your database, which objects to expose, how to handle authentication, and how to behave at runtime. When DAB starts, it reads this configuration and generates your REST and GraphQL endpoints automatically.

A typical configuration file has four main sections:

- **\$schema** - References the JSON schema for validation and IntelliSense support
- **data-source** - Defines the database connection string and type
- **runtime** - Controls runtime behavior like host settings, authentication, and caching
- **entities** - Specifies which database objects to expose and how to configure them

Here's a basic configuration file structure:

```
{
  "$schema": "https://github.com/Azure/data-api-builder/releases/download/v1.1.7/dab-config-schema.json",
  "data-source": {
    "database-type": "mssql",
```

```

    "connection-string": "@env('DATABASE_CONNECTION_STRING')"
  },
  "runtime": {
    "rest": {
      "enabled": true,
      "path": "/api"
    },
    "graphql": {
      "enabled": true,
      "path": "/graphql"
    },
    "host": {
      "cors": {
        "origins": ["http://localhost:3000"],
        "allow-credentials": false
      },
      "authentication": {
        "provider": "StaticWebApps"
      }
    }
  },
  "entities": {}
}

```

Notice the `$schema` property at the top? That gives your editor autocompletion and validation as you type. Always use the schema version that matches your DAB installation.

Configure the data source connection

The `data-source` section tells DAB how to connect to your database. For SQL Server, Azure SQL Database, and SQL databases in Microsoft Fabric, you use `mssql` as the database type.

```

"data-source": {
  "database-type": "mssql",
  "connection-string": "@env('DATABASE_CONNECTION_STRING')",
  "options": {
    "set-session-context": true
  }
}

```

You could put your connection string directly in the configuration file, but don't. Use the `@env()` syntax instead to reference environment variables. This keeps credentials out of your configuration files and source control.

The `set-session-context` option is worth noting. When enabled, DAB passes user identity information to SQL Server through session context. Your stored procedures and functions can then read this information using `SESSION_CONTEXT()`, which is helpful for implementing row-level security or audit logging.

Tip

For local development, you can use the `dab init` command with the `--connection-string` parameter to set up your initial configuration file quickly.

Set up runtime options

The `runtime` section controls how DAB operates when serving requests. Here you configure REST and GraphQL endpoints separately, along with global host settings like CORS and caching.

```
"runtime": {
  "rest": {
    "enabled": true,
    "path": "/api",
    "request-body-strict": true
  },
  "graphql": {
    "enabled": true,
    "path": "/graphql",
    "allow-introspection": true
  },
  "host": {
    "cors": {
      "origins": ["https://myapp.azurewebsites.net"],
      "allow-credentials": true
    },
    "mode": "production"
  },
  "cache": {
    "enabled": true,
    "ttl-seconds": 5
  }
}
```

The `path` properties define the base URL path for each API type. REST endpoints appear under `/api/EntityName`, while GraphQL queries go to `/graphql`. The `allow-introspection` setting for GraphQL determines whether clients can query the schema itself, which is helpful during development but should often be disabled in production for security.

CORS (Cross-Origin Resource Sharing) settings in the `host` section specify which domains can call your API from browser-based applications. The `mode` property switches between `development` and `production` behavior, affecting features like detailed error messages.

Use the DAB CLI for configuration management

The [Data API Builder CLI](#) provides commands for creating and modifying configuration files without manual JSON editing. This approach reduces errors and ensures valid configuration syntax.

To create a new configuration file:

```
dab init --database-type mssql --connection-string "@env('DATABASE_CONNECTION_STRING')"
```

This command creates a `dab-config.json` file in your current directory with baseline settings. From there, you can add entities using CLI commands or edit the file directly.

To add an entity for a table:

```
dab add Product --source dbo.Products --permissions "anonymous:read"
```

The CLI handles JSON formatting and validates your configuration against the schema automatically. If you work in a team, using CLI commands also makes pull request reviews easier since the formatting stays consistent.

Important

Always store your configuration files in source control, but never commit actual connection strings. Use environment variable references and manage credentials through secure mechanisms like Azure Key Vault or deployment pipeline secrets.

Organize configuration for multiple environments

Most projects need different settings for development, staging, and production. You might want introspection enabled during development but disabled in production, or different CORS origins for each environment.

DAB supports this through configuration merging. Start with a base `dab-config.json` file containing shared settings, then create environment-specific overrides:

- `dab-config.development.json` - Development-specific settings
- `dab-config.staging.json` - Staging environment settings
- `dab-config.production.json` - Production settings

When you start DAB with the `--config` parameter, it merges that file's settings with your base configuration:

```
dab start --config dab-config.production.json
```

Environment-specific files only need to contain settings that differ from the base. DAB handles the merge automatically, with environment-specific values taking precedence over base settings.

3. Define entities for REST and GraphQL

Define entities for REST and GraphQL

Completed

Entities are the core building blocks of Data API Builder. Each entity maps to a database object and defines how that object appears through your REST and GraphQL APIs. When you define entities properly, you control what data clients can access, what operations they can perform, and how the data appears in API responses.

Building on the configuration file you created, the `entities` section is where you specify the tables, views, and other database objects that DAB exposes. You configure each entity with source mappings, field customizations, and relationship definitions that shape your API surface.

The examples throughout this unit use the following sample database tables. Keep these definitions in mind as you work through each configuration:

```
CREATE TABLE dbo.Categories (  
    CategoryID INT PRIMARY KEY,  
    CategoryName NVARCHAR(50) NOT NULL  
);  
  
CREATE TABLE dbo.Products (  
    ProductID INT PRIMARY KEY,  
    ProductName NVARCHAR(100) NOT NULL,  
    UnitPrice DECIMAL(10, 2) NOT NULL,  
    UnitsInStock INT NOT NULL,  
    CategoryID INT FOREIGN KEY REFERENCES dbo.Categories(CategoryID)  
);
```

Map database objects to entities

An entity definition starts with a name and a source reference. The entity name becomes the identifier used in API endpoints and GraphQL queries, while the source points to the actual database object.

```
"entities": {  
  "Product": {  
    "source": {  
      "object": "dbo.Products",  
      "type": "table"  
    },  
    "permissions": [  
      {
```

```

        "role": "anonymous",
        "actions": ["read"]
      }
    ]
  }
}

```

This configuration exposes the `dbo.Products` table as a `Product` entity. REST clients access it at `/api/Product`, while GraphQL clients query it as `product` (singular) or `products` (plural). DAB automatically generates these naming conventions based on your entity name.

The `type` property specifies the database object type: `table`, `view`, or `stored-procedure`. Each type has different capabilities. Tables support full CRUD operations (create, read, update, delete), views typically support read operations, and stored procedures execute custom logic.

Configure field mappings and aliases

Sometimes your database column names don't match the naming conventions you want in your API. Field mappings let you rename columns and control which fields appear in API responses.

```

"Product": {
  "source": {
    "object": "dbo.Products",
    "type": "table"
  },
  "mappings": {
    "ProductID": "id",
    "ProductName": "name",
    "UnitPrice": "price",
    "UnitsInStock": "stockQuantity"
  },
  "permissions": [
    {
      "role": "anonymous",
      "actions": ["read"]
    }
  ]
}

```

With these mappings, clients see `id`, `name`, `price`, and `stockQuantity` in their API responses instead of the original database column names. This approach lets you present a clean, consistent API while keeping your existing database schema unchanged.

Note

When you use field mappings, clients must use the mapped names in their queries and mutations. The original database column names become internal implementation details.

Enable caching for improved performance

[Data API Builder caching](#) reduces database load by storing query results temporarily. You can configure caching at both the global level (in the runtime section) and per-entity for fine-grained control.

```
"Product": {
  "source": {
    "object": "dbo.Products",
    "type": "table"
  },
  "rest": {
    "enabled": true
  },
  "graphql": {
    "enabled": true,
    "cache": {
      "enabled": true,
      "ttl-seconds": 60
    }
  },
  "permissions": [...]
}
```

The `ttl-seconds` property (time-to-live) determines how long cached data remains valid. For product catalogs that change infrequently, longer cache durations improve performance. For rapidly changing data like inventory counts, use shorter durations or disable caching.

Caching works differently for REST and GraphQL. REST caching considers the complete URL including query parameters, while GraphQL caching examines the query structure. Both respect the TTL you configure.

Implement pagination for large datasets

When entities contain many records, returning all of them in a single response creates performance and usability problems. Data API Builder provides built-in pagination support that you can customize per entity.

```
"Product": {
  "source": {
    "object": "dbo.Products",
    "type": "table"
  },
  "rest": {
    "enabled": true,
    "path": "/products"
  },
  "graphql": {
    "enabled": true,
```

```

    "type": {
      "singular": "product",
      "plural": "products"
    }
  },
  "permissions": [...]
}

```

REST clients use query parameters like `$first` and `$after` to navigate through pages:

```

GET /api/products?$first=10
GET /api/products?$first=10&$after=eyJQcm9kdWN0SUQiOjEwfQ==

```

The `$after` parameter contains a cursor that points to the last item from the previous page. DAB generates these cursors automatically based on your entity's primary key.

GraphQL clients use similar pagination through the `first` and `after` arguments in their queries:

```

query {
  products(first: 10, after: "eyJQcm9kdWN0SUQiOjEwfQ==") {
    items {
      id
      name
      price
    }
    hasNextPage
    endCursor
  }
}

```

The response includes `hasNextPage` and `endCursor` fields that clients use to determine if more data exists and how to fetch it.

Configure filtering and searching

Data API Builder generates filtering capabilities automatically based on your entity's fields. Clients can filter results using standard comparison operators.

For REST endpoints, filtering uses OData-style query syntax:

```

GET /api/products?$filter=price gt 100 and stockQuantity gt 0

```

For GraphQL, filtering uses typed input objects:

```

query {
  products(filter: { price: { gt: 100 }, stockQuantity: { gt: 0 } }) {
    items {
      id
      name
    }
  }
}

```

```
    price
  }
}
```

DAB generates filter types for each field based on its data type. Numeric fields support `gt`, `gte`, `lt`, `lte`, `eq`, and `neq`. String fields add `contains`, `startsWith`, and `endsWith`. You don't need to configure these capabilities explicitly; they're available automatically for all exposed fields.

Tip

For complex filtering scenarios that can't be expressed through standard operators, consider creating a view or stored procedure that encapsulates the filtering logic, then expose that as a separate entity.

Define entity relationships for GraphQL

GraphQL's strength lies in traversing relationships between entities in a single query. Data API Builder supports defining relationships that map to your database's foreign key relationships.

```
"entities": {
  "Product": {
    "source": { "object": "dbo.Products", "type": "table" },
    "relationships": {
      "category": {
        "cardinality": "one",
        "target.entity": "Category",
        "source.fields": ["CategoryID"],
        "target.fields": ["CategoryID"]
      }
    },
    "permissions": [...]
  },
  "Category": {
    "source": { "object": "dbo.Categories", "type": "table" },
    "relationships": {
      "products": {
        "cardinality": "many",
        "target.entity": "Product",
        "source.fields": ["CategoryID"],
        "target.fields": ["CategoryID"]
      }
    },
    "permissions": [...]
  }
}
```

With these relationships configured, GraphQL clients can query across entities:

```

query {
  products {
    items {
      name
      price
      category {
        name
      }
    }
  }
}

```

The `cardinality` property indicates whether the relationship returns one item or many. Use `one` for many-to-one relationships (product to category) and `many` for one-to-many relationships (category to products).

Understand REST endpoint generation

When you define an entity, DAB automatically creates REST endpoints following standard conventions. Each entity gets a base URL path, and HTTP verbs map to CRUD operations.

For an entity named `Product` with the default configuration:

HTTP Method	URL Pattern	Operation
GET	<code>/api/Product</code>	List all products (paginated)
GET	<code>/api/Product/id/123</code>	Get product with ID 123
POST	<code>/api/Product</code>	Create a new product
PUT	<code>/api/Product/id/123</code>	Update or create product 123
PATCH	<code>/api/Product/id/123</code>	Partially update product 123
DELETE	<code>/api/Product/id/123</code>	Delete product 123

The `/id/` segment in URLs indicates a primary key lookup. For composite keys, chain multiple key segments: `/api/OrderDetail/orderId/100/productId/50`.

Customize REST endpoint paths

You can modify the default URL patterns through entity configuration. The `rest` section controls REST-specific behavior:

```

"Product": {
  "source": { "object": "dbo.Products", "type": "table" },
  "rest": {
    "enabled": true,
    "path": "/products",

```

```

    "methods": {
      "get": true,
      "post": true,
      "put": false,
      "patch": true,
      "delete": true
    }
  },
  "permissions": [...]
}

```

The `path` property changes the URL from `/api/Product` to `/api/products`. This setting is useful when you want lowercase, plural endpoint names that differ from your entity names.

The `methods` object lets you enable or disable specific HTTP verbs. In this example, `PUT` is disabled, which prevents full record replacement while still allowing `PATCH` for partial updates. This configuration pattern protects against accidental data loss from clients that might send incomplete records.

Tip

Disabling methods at the endpoint level provides defense in depth. Even if permissions allow an action, disabling the method prevents it entirely.

Set up GraphQL endpoint configuration

GraphQL operates through a single endpoint, typically at `/graphql`. All queries, mutations, and subscriptions go through this endpoint, with the request body specifying the operation.

```

"runtime": {
  "graphql": {
    "enabled": true,
    "path": "/graphql",
    "allow-introspection": true,
    "depth-limit": 8,
    "multiple-mutations": {
      "create": { "enabled": true }
    }
  }
}

```

The `allow-introspection` setting controls whether clients can query the GraphQL schema itself. During development, introspection helps tools like GraphiQL and Apollo Studio understand your API. In production, consider disabling it to hide implementation details.

The `depth-limit` setting prevents overly complex nested queries that could strain your database. A depth of 8 allows reasonable relationship traversal while blocking potential abuse through deeply nested queries.

Configure GraphQL-specific entity settings

Each entity can have GraphQL-specific configuration that differs from REST:

```
"Product": {
  "source": { "object": "dbo.Products", "type": "table" },
  "graphql": {
    "enabled": true,
    "type": {
      "singular": "product",
      "plural": "products"
    },
    "operation": "query"
  },
  "rest": {
    "enabled": true
  },
  "permissions": [...]
}
```

The `type` property customizes the GraphQL type names. By default, DAB uses the entity name, but you can specify different singular and plural forms.

The `operation` property determines where mutations appear. Use `query` for read-only entities (like views) or `mutation` for entities that support write operations.

4. Expose database objects, stored procedures, and views

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/4-expose-database-objects-stored-procedures-views>

Expose database objects, stored procedures, and views

Completed

Beyond simple table mappings, Data API Builder can expose views and stored procedures as API endpoints. These database objects let you encapsulate complex business logic, aggregate data from multiple tables, or implement operations that go beyond standard CRUD patterns.

Views provide read-only access to computed or joined data, while stored procedures enable custom operations with input parameters and multiple result sets. GraphQL relationships add another dimension, allowing clients to traverse connections between entities in single queries.

Expose views as read-only entities

Database views aggregate and transform data without duplicating storage. When you expose a view through Data API Builder, clients access the computed results through familiar REST and GraphQL patterns.

Consider a view that combines product information with inventory status:

```
CREATE VIEW dbo.ProductInventory AS
SELECT
    p.ProductID,
    p.ProductName,
    p.UnitPrice,
    p.UnitsInStock,
    c.CategoryName,
    CASE
        WHEN p.UnitsInStock = 0 THEN 'Out of Stock'
        WHEN p.UnitsInStock < 10 THEN 'Low Stock'
        ELSE 'Available'
    END AS StockStatus
FROM dbo.Products p
INNER JOIN dbo.Categories c ON p.CategoryID = c.CategoryID;
```

Expose this view in your configuration:

```
"ProductInventory": {
  "source": {
    "object": "dbo.ProductInventory",
    "type": "view",
    "key-fields": ["ProductID"]
  },
  "rest": {
    "enabled": true,
    "path": "/inventory"
  },
  "graphql": {
    "enabled": true,
    "operation": "query"
  },
  "permissions": [
    {
      "role": "anonymous",
      "actions": ["read"]
    }
  ]
}
```

The `key-fields` property is required for views because DAB can't automatically detect primary keys. Specify the columns that uniquely identify each row.

Setting `operation` to `query` in the GraphQL section indicates this entity supports only read operations. DAB won't generate mutations for this entity.

Note

Views can be updatable in SQL Server if they meet certain criteria. However, exposing updatable views through DAB requires careful consideration of the underlying table permissions and constraints.

Configure stored procedures for custom operations

[Stored procedures](#) provide the most flexibility for custom database operations. They can accept parameters, perform complex logic, and return multiple result sets.

```
CREATE PROCEDURE dbo.GetProductsByPriceRange
    @MinPrice DECIMAL(10,2),
    @MaxPrice DECIMAL(10,2)
AS
BEGIN
    SELECT ProductID, ProductName, UnitPrice, UnitsInStock
    FROM dbo.Products
    WHERE UnitPrice BETWEEN @MinPrice AND @MaxPrice
    ORDER BY UnitPrice;
END;
```

Configure the stored procedure as an entity:

```
"GetProductsByPriceRange": {
  "source": {
    "object": "dbo.GetProductsByPriceRange",
    "type": "stored-procedure",
    "parameters": {
      "MinPrice": "minPrice",
      "MaxPrice": "maxPrice"
    }
  },
  "rest": {
    "enabled": true,
    "path": "/products/by-price-range",
    "methods": {
      "get": true,
      "post": true
    }
  },
  "graphql": {
    "enabled": true,
    "operation": "query"
  },
  "permissions": [
```

```
{
  "role": "authenticated",
  "actions": ["execute"]
}
```

The `parameters` object maps stored procedure parameters to API parameter names. REST clients call the procedure using query parameters or request body:

```
GET /api/products/by-price-range?minPrice=10&maxPrice=50
```

```
POST /api/products/by-price-range
Content-Type: application/json

{
  "minPrice": 10,
  "maxPrice": 50
}
```

GraphQL clients use the procedure through a generated query:

```
query {
  executeGetProductsByPriceRange(minPrice: 10, maxPrice: 50) {
    ProductID
    ProductName
    UnitPrice
  }
}
```

Expose stored procedures that modify data

For stored procedures that insert, update, or delete data, configure them with mutation operations:

```
CREATE PROCEDURE dbo.CreateOrder
  @CustomerID INT,
  @ProductID INT,
  @Quantity INT
AS
BEGIN
  DECLARE @OrderID INT;

  INSERT INTO dbo.Orders (CustomerID, OrderDate, Status)
  VALUES (@CustomerID, GETDATE(), 'Pending');

  SET @OrderID = SCOPE_IDENTITY();

  INSERT INTO dbo.OrderDetails (OrderID, ProductID, Quantity, UnitPrice)
  SELECT @OrderID, @ProductID, @Quantity, UnitPrice
  FROM dbo.Products
```

```

WHERE ProductID = @ProductID;

UPDATE dbo.Products
SET UnitsInStock = UnitsInStock - @Quantity
WHERE ProductID = @ProductID;

SELECT @OrderID AS OrderID;
END;

```

```

"CreateOrder": {
  "source": {
    "object": "dbo.CreateOrder",
    "type": "stored-procedure",
    "parameters": {
      "CustomerID": "customerId",
      "ProductID": "productId",
      "Quantity": "quantity"
    }
  },
  "rest": {
    "enabled": true,
    "methods": {
      "post": true
    }
  },
  "graphql": {
    "enabled": true,
    "operation": "mutation"
  },
  "permissions": [
    {
      "role": "authenticated",
      "actions": ["execute"]
    }
  ]
}

```

Setting `operation` to `mutation` places this procedure under GraphQL mutations rather than queries, indicating it modifies data.

Important

Stored procedures execute with the permissions of the database connection, not the calling user. Implement appropriate validation and authorization logic within the procedure or use session context for row-level security.

Define GraphQL relationships between entities

GraphQL relationships enable clients to traverse connections between entities without multiple round trips. DAB supports both one-to-one and one-to-many relationships.

```
"entities": {
  "Order": {
    "source": { "object": "dbo.Orders", "type": "table" },
    "relationships": {
      "customer": {
        "cardinality": "one",
        "target.entity": "Customer",
        "source.fields": ["CustomerID"],
        "target.fields": ["CustomerID"]
      },
      "orderDetails": {
        "cardinality": "many",
        "target.entity": "OrderDetail",
        "source.fields": ["OrderID"],
        "target.fields": ["OrderID"]
      }
    },
    "permissions": [...]
  },
  "OrderDetail": {
    "source": { "object": "dbo.OrderDetails", "type": "table" },
    "relationships": {
      "order": {
        "cardinality": "one",
        "target.entity": "Order",
        "source.fields": ["OrderID"],
        "target.fields": ["OrderID"]
      },
      "product": {
        "cardinality": "one",
        "target.entity": "Product",
        "source.fields": ["ProductID"],
        "target.fields": ["ProductID"]
      }
    },
    "permissions": [...]
  }
}
```

With these relationships, GraphQL clients can build rich queries:

```
query {
  orders(first: 10) {
    items {
      orderDate
      customer {
        companyName
        contactName
      }
      orderDetails {
```

```

    items {
      quantity
      product {
        productName
        unitPrice
      }
    }
  }
}

```

This single query retrieves orders with customer information and line items including product details. Without relationships, clients would need multiple separate requests.

Handle complex relationship scenarios

Some database designs use junction tables for many-to-many relationships. DAB handles these through explicit relationship chains.

For a products-to-suppliers many-to-many relationship through a `ProductSuppliers` junction table:

```

"Product": {
  "relationships": {
    "productSuppliers": {
      "cardinality": "many",
      "target.entity": "ProductSupplier",
      "source.fields": ["ProductID"],
      "target.fields": ["ProductID"]
    }
  }
},
"ProductSupplier": {
  "source": { "object": "dbo.ProductSuppliers", "type": "table" },
  "relationships": {
    "supplier": {
      "cardinality": "one",
      "target.entity": "Supplier",
      "source.fields": ["SupplierID"],
      "target.fields": ["SupplierID"]
    }
  }
}

```

Clients traverse the relationship chain: Product → ProductSupplier → Supplier. While this requires one extra level in queries, it accurately represents the database structure and maintains referential integrity.

5. Explore deployment options for Data API Builder

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/5-deploy-data-api-builder-azure-services>

Explore deployment options for Data API Builder

Completed

Moving Data API Builder from local development to Azure production environments requires choosing the right hosting platform and configuring deployment automation. Azure offers several options for running DAB, each with different characteristics for scalability, management overhead, and integration with other Azure services.

Your deployment strategy affects not only where DAB runs but also how you manage configuration, handle secrets, and scale to meet demand. Understanding these deployment options helps you select the approach that best fits your application architecture and operational requirements.

Deploy to Azure Container Apps

[Azure Container Apps](#) provides a serverless container platform that handles infrastructure management automatically. It's well-suited for Data API Builder because it can scale from zero when idle and automatically scale up based on request volume.

To deploy DAB to Container Apps, first create a container image with your configuration:

```
FROM mcr.microsoft.com/azure-databases/data-api-builder:latest

COPY dab-config.json /App/dab-config.json

ENV DATABASE_CONNECTION_STRING=""
ENV ASPNETCORE_URLS="http://+:5000"

EXPOSE 5000
```

This Dockerfile starts from the official DAB image and adds your configuration file. Environment variables handle the connection string and port configuration.

Deploy using the Azure CLI:

```
az containerapp create \
  --name dab-api \
  --resource-group myResourceGroup \
  --environment myContainerAppEnv \
  --image myregistry.azurecr.io/dab-api:v1 \
  --target-port 5000 \
  --ingress external \
  --secrets database-conn="<connection-string>" \
  --env-vars DATABASE_CONNECTION_STRING=secretref:database-conn
```

Container Apps integrates with Azure Key Vault for secret management. Reference secrets in your environment variables rather than storing credentials in configuration files or container images.

Tip

Enable Azure Container Apps managed identity and grant it access to your Azure SQL Database. This approach eliminates stored credentials entirely by using Microsoft Entra authentication.

Deploy to Azure App Service

[Azure App Service](#) offers a mature platform with built-in deployment slots, custom domains, and comprehensive monitoring. For teams already using App Service, adding DAB follows familiar patterns.

Configure App Service to run the DAB container:

```
az webapp create \
  --name dab-api \
  --resource-group myResourceGroup \
  --plan myAppServicePlan \
  --deployment-container-image-name mcr.microsoft.com/azure-databases/data-api-bui
```

Set environment variables through App Service configuration:

```
az webapp config appsettings set \
  --name dab-api \
  --resource-group myResourceGroup \
  --settings DATABASE_CONNECTION_STRING="@Microsoft.KeyVault(SecretUri=https://myva
```

The Key Vault reference syntax lets App Service pull secrets at runtime without exposing them in configuration. Enable system-assigned managed identity on the App Service and grant it access to Key Vault.

Deploy to Azure Static Web Apps

[Azure Static Web Apps](#) includes Data API Builder as a built-in feature called database connections. This integration provides the simplest deployment experience when your API complements a static frontend.

Link your database through the Azure portal or CLI:

```
az staticwebapp dbconnection create \  
  --name myStaticWebApp \  
  --resource-group myResourceGroup \  
  --db-type mssql \  
  --db-resource-id /subscriptions/.../servers/myserver/databases/mydb
```

The `staticwebapps.database.config.json` file in your repository defines entity configurations using the same structure as standalone DAB:

```
{  
  "data-source": {  
    "database-type": "mssql"  
  },  
  "entities": {  
    "Product": {  
      "source": "dbo.Products",  
      "permissions": [  
        { "role": "anonymous", "actions": ["read"] }  
      ]  
    }  
  }  
}
```

Static Web Apps automatically handles authentication integration with its built-in auth providers. API routes appear under `/.auth/` and `/data-api/` paths by default.

Note

Database connections in Static Web Apps have some limitations compared to standalone DAB, including restricted stored procedure support. Review the documentation for current feature availability.

Implement CI/CD deployment pipelines

Automated deployment pipelines ensure consistent configuration across environments and enable rapid iteration. GitHub Actions provides a straightforward approach for DAB deployments.

Create a workflow file at `.github/workflows/deploy-dab.yml`:

```
name: Deploy Data API Builder  
  
on:
```

```

push:
  branches: [main]
  paths:
    - 'dab-config*.json'
    - 'Dockerfile'

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Login to Azure Container Registry
        uses: azure/docker-login@v1
        with:
          login-server: ${ secrets.ACR_LOGIN_SERVER }
          username: ${ secrets.ACR_USERNAME }
          password: ${ secrets.ACR_PASSWORD }

      - name: Build and push container
        run: |
          docker build -t ${ secrets.ACR_LOGIN_SERVER }/dab-api:${ github.sha }
          docker push ${ secrets.ACR_LOGIN_SERVER }/dab-api:${ github.sha }

      - name: Deploy to Container Apps
        uses: azure/container-apps-deploy-action@v1
        with:
          resourceGroup: myResourceGroup
          containerAppName: dab-api
          imageToDeploy: ${ secrets.ACR_LOGIN_SERVER }/dab-api:${ github.sha }

```

This workflow triggers on changes to configuration files or the Dockerfile. It builds a new container image, pushes it to Azure Container Registry, and updates the Container App.

Configure environment-specific deployments

Production deployments require careful separation between environments. Use environment-specific configuration files combined with deployment variables.

Structure your repository with multiple configuration files:

```

/
├── dab-config.json           # Base configuration (shared settings)
├── dab-config.development.json
├── dab-config.staging.json
├── dab-config.production.json
└── Dockerfile

```

Modify your Dockerfile to accept a build argument:

```
FROM mcr.microsoft.com/azure-databases/data-api-builder:latest

ARG ENVIRONMENT=production
COPY dab-config.json /App/dab-config.json
COPY dab-config.${ENVIRONMENT}.json /App/dab-config.${ENVIRONMENT}.json

ENV DAB_ENVIRONMENT=${ENVIRONMENT}
```

Build environment-specific images:

```
docker build --build-arg ENVIRONMENT=production -t dab-api:production .
```

Important

Never include actual connection strings or secrets in configuration files committed to source control. Use environment variable references (`@env()`) and manage secrets through Azure Key Vault or your deployment pipeline's secret management.

Monitor deployment health

After deployment, configure health checks to verify DAB is operating correctly. Container Apps and App Service support health probes:

```
{
  "probes": [
    {
      "type": "liveness",
      "httpGet": {
        "path": "/",
        "port": 5000
      },
      "periodSeconds": 30
    },
    {
      "type": "readiness",
      "httpGet": {
        "path": "/api/Product?$first=1",
        "port": 5000
      },
      "periodSeconds": 10
    }
  ]
}
```

The liveness probe checks that DAB is responding. The readiness probe verifies database connectivity by making an actual API request. Configure your orchestration platform to restart unhealthy instances automatically.

6. Recommend Azure Monitor configurations

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/6-recommend-azure-monitor-configurations>

Recommend Azure Monitor configurations

Completed

Running Data API Builder in production requires monitoring to understand performance, diagnose issues, and optimize resource usage. [Azure Monitor](#) provides the comprehensive observability platform that integrates with DAB through Application Insights and Log Analytics.

Effective monitoring goes beyond simple uptime checks. You need visibility into request latencies, error rates, database query performance, and resource consumption patterns. This type of data helps you identify bottlenecks before they affect users and provides data for capacity planning.

Configure Application Insights integration

[Application Insights](#) captures detailed information from your Data API Builder instance, including request traces, dependency calls to your database, and exception details.

Enable Application Insights by setting the connection string as an environment variable:

```
APPLICATIONINSIGHTS_CONNECTION_STRING="InstrumentationKey=your-key;IngestionEndpoint=your-endpoint"
```

For Azure Container Apps or App Service deployments, configure this through application settings. DAB automatically detects the connection string and begins sending telemetry.

Once enabled, Application Insights tracks:

- **Request telemetry** - Every REST and GraphQL request with timing, status code, and response size
- **Dependency telemetry** - Database queries with execution time and success/failure status
- **Exception telemetry** - Unhandled errors with stack traces
- **Custom metrics** - Cache hit rates, connection pool usage

Tip

Set a sampling rate for high-volume production deployments to control costs while maintaining visibility. Application Insights adaptive sampling automatically adjusts based on traffic volume.

Create custom dashboards for API monitoring

Application Insights data powers custom Azure dashboards that display key performance indicators for your API.

Create a dashboard with these essential visualizations:

Request metrics:

```
requests
| where timestamp > ago(24h)
| summarize
    RequestCount = count(),
    AvgDuration = avg(duration),
    P95Duration = percentile(duration, 95),
    FailureRate = countif(success == false) * 100.0 / count()
| project RequestCount, AvgDuration, P95Duration, FailureRate
```

This query summarizes 24-hour request volume, average response time, 95th percentile latency, and failure rate. These metrics quickly indicate API health.

Top slow endpoints:

```
requests
| where timestamp > ago(1h)
| summarize
    AvgDuration = avg(duration),
    RequestCount = count()
    by name
| top 10 by AvgDuration desc
| project name, AvgDuration, RequestCount
```

Identifying slow endpoints helps prioritize optimization efforts. Slow queries might indicate missing indexes, inefficient entity configurations, or database contention.

Set up Log Analytics for detailed diagnostics

[Log Analytics](#) provides a central repository for logs from all Azure resources. Configure your DAB deployment to send logs to a Log Analytics workspace.

For Container Apps, enable logging through the environment configuration:

```
az containerapp env update \
  --name myContainerAppEnv \
  --resource-group myResourceGroup \
  --logs-workspace-id <workspace-id>
```

Query DAB container logs to investigate issues:

```
ContainerAppConsoleLogs_CL
| where ContainerName_s == "dab-api"
| where Log_s contains "error" or Log_s contains "exception"
| project TimeGenerated, Log_s
| order by TimeGenerated desc
```

Combine container logs with Application Insights telemetry for complete request traces. Correlate errors with specific requests using operation IDs that flow through both data sources.

Configure alert rules for proactive monitoring

Alert rules notify your team when API behavior deviates from normal patterns. Set up alerts for critical conditions that require immediate attention.

High error rate alert:

```
az monitor metrics alert create \
  --name "DAB-HighErrorRate" \
  --resource-group myResourceGroup \
  --scopes "/subscriptions/.../components/myAppInsights" \
  --condition "avg requests/failed > 5" \
  --window-size 5m \
  --evaluation-frequency 1m \
  --action-group myActionGroup
```

This alert triggers when the failure rate exceeds 5% over a 5-minute window. Action groups define notification channels like email, SMS, or webhooks to your incident management system.

Response time degradation alert:

```
requests
| where timestamp > ago(5m)
| summarize P95 = percentile(duration, 95) by bin(timestamp, 1m)
| where P95 > 2000
```

Create a log-based alert that fires when 95th percentile latency exceeds 2 seconds. This catches performance degradation even when requests technically succeed.

Important

Configure alerts thoughtfully to avoid alert fatigue. Start with high-severity conditions and refine thresholds based on normal operating patterns.

Monitor database query performance

Application Insights dependency tracking captures database queries issued by DAB. Use this telemetry to identify slow or frequently executed queries.

```
dependencies
| where timestamp > ago(1h)
| where type == "SQL"
| summarize
    CallCount = count(),
    AvgDuration = avg(duration),
    FailureCount = countif(success == false)
    by data
| top 20 by AvgDuration desc
```

The `data` field contains the SQL query text. Review slow queries to determine if indexes would help, if entity configurations need optimization, or if certain operations should use stored procedures instead.

Compare database metrics across time periods to identify trends:

```
dependencies
| where type == "SQL"
| summarize
    AvgDuration = avg(duration)
    by bin(timestamp, 1h)
| render timechart
```

Gradual increases in query duration might indicate growing data volumes, index fragmentation, or changing query patterns that need attention.

Implement distributed tracing

When DAB serves requests from multiple client applications, distributed tracing connects the complete request flow from client through API to database.

Enable correlation by including trace context headers in client requests:

```
GET /api/Product HTTP/1.1
traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01
```

Application Insights correlates these traces, allowing you to see end-to-end latency breakdown in the Transaction Search view. This visibility is valuable for diagnosing intermittent issues that span multiple services.

For GraphQL queries that resolve multiple entities, traces show the timing of each database call, helping identify which relationships or entities contribute most to response time.

Review resource utilization metrics

Beyond application telemetry, monitor the underlying compute resources running DAB. Container Apps and App Service provide resource metrics through Azure Monitor.

Key metrics to track:

- **CPU percentage** - Sustained high CPU indicates compute-bound operations or need for scaling
- **Memory percentage** - Memory pressure can cause container restarts or degraded performance
- **Network bytes in/out** - Unusual network patterns might indicate large result sets or denial-of-service attempts
- **Replica count** - For scaled deployments, verify autoscaling responds appropriately to load

Set up autoscaling rules based on these metrics to maintain responsive APIs during traffic spikes while controlling costs during quiet periods.

7. Handle changes with event-driven patterns

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/7-handle-changes-event-driven-patterns>

Handle changes with event-driven patterns

Completed

Modern applications often need to respond to database changes in real time. When a customer places an order, updates their profile, or when inventory levels change, downstream systems need notification. SQL Server and Azure SQL provide several mechanisms for capturing and streaming these changes, each with different characteristics for latency, throughput, and implementation complexity.

Data API Builder integrates with these change mechanisms through Azure Functions SQL trigger bindings. Understanding when to use each approach helps you build responsive, event-driven architectures that react to data changes efficiently.

Understand change capture mechanisms

SQL platforms offer multiple approaches for tracking data changes, each suited to different scenarios:

- **Change Data Capture (CDC)** records insert, update, and delete operations to special change tables. It captures the complete before-and-after state of changed rows, making it ideal for audit requirements and data synchronization.

- [Change Tracking](#) provides a lighter-weight alternative that records which rows changed without capturing the actual values. Applications query change information to determine what data needs synchronization.
- [Change Event Streaming \(CES\)](#) is a newer feature that pushes changes to event streams without polling. It integrates directly with Azure Event Hubs for high-throughput scenarios.

Feature	CDC	Change Tracking	CES
Captures changed values	Yes	No	Yes
Polling required	Yes	Yes	No
Historical data	Yes	Limited	Stream only
Setup complexity	Moderate	Low	Moderate

Enable Change Data Capture for audit scenarios

CDC is appropriate when you need a complete history of changes for compliance or debugging. Enable it on specific tables:

```
-- Enable CDC at the database level
EXEC sys.sp_cdc_enable_db;

-- Enable CDC on a specific table
EXEC sys.sp_cdc_enable_table
    @source_schema = 'dbo',
    @source_name = 'Orders',
    @role_name = NULL,
    @capture_instance = 'dbo_Orders',
    @supports_net_changes = 1;
```

Once enabled, SQL Server creates change tables that store historical records. Query these tables to retrieve changes within a time range:

```
DECLARE @from_lsn binary(10), @to_lsn binary(10);
SET @from_lsn = sys.fn_cdc_get_min_lsn('dbo_Orders');
SET @to_lsn = sys.fn_cdc_get_max_lsn();

SELECT *
FROM cdc.fn_cdc_get_all_changes_dbo_Orders(@from_lsn, @to_lsn, 'all');
```

This query returns all changes between the minimum and maximum log sequence numbers, including the operation type (`__$operation`) indicating insert, update, or delete.

Note

CDC requires SQL Server Agent to be running. The capture and cleanup jobs run on schedules you can configure. Monitor job performance in high-transaction environments.

Configure Azure Functions SQL trigger binding

[Azure Functions SQL trigger binding](#) provides the most direct integration for responding to database changes. The trigger monitors tables using Change Tracking and invokes your function when rows change.

First, enable Change Tracking on your database and table:

```
-- Enable Change Tracking at database level
ALTER DATABASE [YourDatabase]
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON);

-- Enable Change Tracking on the table
ALTER TABLE dbo.Orders
ENABLE CHANGE_TRACKING;
```

Create an Azure Function with the SQL trigger:

```
[FunctionName("OrderChangedTrigger")]
public static void Run(
    [SqlTrigger("[dbo].[Orders]", "SqlConnectionString")]
    IReadOnlyList<SqlChange<Order>> changes,
    ILogger log)
{
    foreach (var change in changes)
    {
        log.LogInformation($"Change operation: {change.Operation}");
        log.LogInformation($"Order ID: {change.Item.OrderId}");

        if (change.Operation == SqlChangeOperation.Update)
        {
            // Process order update
            ProcessOrderUpdate(change.Item);
        }
    }
}

public class Order
{
    public int OrderId { get; set; }
    public int CustomerId { get; set; }
    public DateTime OrderDate { get; set; }
    public string Status { get; set; }
}
```

The `SqlChange<T>` type wraps each changed row with its operation type. Your function receives batches of changes, which helps efficiency when multiple rows change simultaneously.

Configure the connection string in your function app settings:

```
{
  "Values": {
    "SqlConnectionStrings": "@Microsoft.KeyVault(SecretUri=https://myvault.vault.azure.net/secrets/secret-name/secret-value)"
  }
}
```

Important

The SQL trigger binding requires Change Tracking, not CDC. If you need CDC's historical capabilities alongside real-time triggers, enable both features on your tables.

Stream changes with Change Event Streaming

For high-throughput scenarios where polling latency is unacceptable, Change Event Streaming pushes changes directly to Azure Event Hubs. This approach eliminates polling delays and scales to millions of events per second.

Configure CES on your database:

```
-- Create an event streaming group
CREATE EVENT STREAMING GROUP MyOrderStream
WITH (
  TARGET_TYPE = 'eventhub',
  TARGET_NAME = 'orders-eventhub',
  TARGET_CONNECTION = '<event-hub-connection-string>'
);

-- Add tables to the streaming group
ALTER EVENT STREAMING GROUP MyOrderStream
ADD TABLE dbo.Orders;
```

Changes flow to Event Hubs immediately as transactions commit. Downstream consumers process events using Azure Functions Event Hubs triggers, Stream Analytics, or custom applications.

The Event Hubs message contains:

- Operation type (insert, update, delete)
- Timestamp
- Full row data for inserts and updates
- Key values for deletes

Choose the right change pattern

Selecting the appropriate change mechanism depends on your requirements:

Use **CDC** when:

- You need complete before-and-after values
- Compliance requires historical change records
- Batch synchronization is acceptable

Use **change tracking** with Functions when:

- You need real-time response to changes
- You only need current values, not history
- You're building event-driven microservices

Use **change event streaming** when:

- You need sub-second latency
- High transaction volumes require streaming
- You're building real-time analytics pipelines

Consider combining approaches. Use CDC for audit and compliance while using Functions triggers for real-time notifications. This pattern provides both historical records and immediate responses.

8. Exercise - Configure Data API Builder for a product catalog

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/8-exercise-data-api-builder>

Exercise - Configure Data API Builder for a product catalog

Completed

Now it's your chance to create a Data API Builder configuration and expose a SQL database through modern APIs.

In this exercise, you create a Data API Builder configuration for a product catalog database. You configure entities for tables and views, set up field mappings and relationships, and test REST and GraphQL endpoints locally.

Note

To complete this exercise, you'll need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

9. Module assessment

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/9-knowledge-check>

Module assessment

Completed

10. Summary

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/10-summary>

Summary

Completed

In this module, you learned how to:

- Create Data API Builder configuration files with data source connections and runtime settings
- Define entities for tables with field mappings, caching, pagination, and filtering capabilities
- Configure REST and GraphQL endpoints with customized paths and enabled operations
- Expose views as read-only entities and stored procedures for custom operations
- Explore deployment options for Data API Builder, including Azure Container Apps, App Service, and Static Web Apps
- Set up Azure Monitor with Application Insights for request telemetry and database performance tracking
- Implement change capture patterns using CDC, Change Tracking, Azure Functions triggers, and Change Event Streaming

Key takeaways

- Data API Builder eliminates custom API code by generating REST and GraphQL endpoints from a single JSON configuration file
- Entity relationships enable GraphQL clients to traverse connections between tables in single queries, reducing round trips
- Azure Container Apps provides serverless hosting with automatic scaling and managed identity support for production deployments
- Application Insights integration gives visibility into request performance, database queries, and error rates
- Change capture mechanisms enable event-driven architectures that respond to database modifications in real time

Learn more

- [Data API Builder documentation](#)
- [Data API Builder configuration reference](#)
- [Deploy Data API Builder to Azure Container Apps](#)
- [Azure Functions SQL trigger binding](#)
- [Change Data Capture in SQL Server](#)
- [Application Insights overview](#)