Entity Anywhere
Allowing Entities to Exist Generically in Disparate Systems

Jared A. Barneck

***Abstract*—Entity Anywhere is a new architecture that extends data querying out to the service endpoint, often over HTTP. It allows for data, often represented as an entity, to exist in any data store or repository. Similarly, a set of entities could exist in a set of repositories. Yet these entities could still share relationships despite existing in disparate systems. Once a platform such as Entity Anywhere is in place, other systems can be created to use it as its data communication layer. However, the backend must be able to communicate with the various repositories that may host the entity data. Once code is written to communicate to a common repository, that code should be reusable. The final result of Entity Anywhere is that we have systems running on top of other systems to reduce data duplication, data syncing, and simplify business processes.**

***Index Terms*—Entity, Repository Pattern, OData, Commodity, ETL, Entity Anywhere, REST, RESTful, Plugin, Custom or Common.**

I. Problem

MANY ORGANIZATIONS have disparate systems that all work together, such as a Customer Relationship Manager (CRM), a Finance system, a Human Resources (HR) system, a User Management system, and more. The data and processes in these systems are often duplicated. Middleware usually must be built to keep the data in these systems in sync. Adding a new system could mean adding another place to duplicate data, more duplicate processes, and more middleware. This creates overly complex business systems and processes.

Different systems are the data master for different pieces of data. A representation of a piece of data is called an *entity*. The data that makes up these entities are often duplicated across disparate systems. Duplication of data causes various problems in any organization.

What if you could add a system without duplicate entity data? What if you could add a shopping cart that ran on top of your existing systems regardless of what underlying systems they were? This could be done by coding your own backend system to talk to these various repositories, then coding your own frontend system to use your custom backend system. However, the cost of custom coding a system, such as a shopping cart, for an enterprise would be a multimillion-dollar project that would take years. This issue affects many organizations.

Prebuilt systems, either open source or commercial, often solve a single problem but introduce more problems than they solve. They do not support custom backend repositories, but what if they did? What if you could implement a system on top your existing infrastructure with little to no coding, but still have the ability to customize the code as needed for your business?

If the repository pattern were mature and most open source and commercial system implemented it, then the cost of adding a new system would drop to only include the cost of developing a custom backend repository. Now, what if the backend repositories were commoditized? The repository pattern has also not yet matured to the point of being a commodity. The consumer cannot buy or download code that already contains all implementations necessary to communicate with a common system, such as Salesforce or SAP, and easily query an entity.

Entity Anywhere architecture is a proposed method for maturing the repository pattern from requiring customized code for each repository implementation, to instead using a commoditized implementation. Adding a new system, such as a shopping cart, could be reduced down to only setup time. 1) install the Entity Anywhere system; 2) install the desired new system, such as a shopping cart; 3) install the commoditized repositories that already know how to query or write entity data from or to your existing data stores.

The result is Zero data migration. Zero new processes. Zero middleware updates.

The goal of the Entity Anywhere architecture is to make this solution a possibility.

II. Introduction

Entity Anywhere is the combination of existing development architectures, architectural styles, and protocols with the goal of writing entity data access code once, from the web server to the repository, and then commoditizing that code.

The name is self-explanatory. Entity Anywhere:

- **Entity** is a data representation of an object such as User, Organization, Order, Product, Sku, etc.)
- **Anywhere** means that the entity can be stored anywhere, on any system.

Entity Anywhere has a simple goal encompassed in its name.

Write code without worrying about where the data representing an entity resides. The data could be in an existing system, a new custom system, a text file such as a csv, spreadsheet, xml, or again, anywhere.

The repository pattern makes this possible but is not enough by itself to sustain this architecture. Other layers and well-known software design patterns are needed.

Entity Anywhere consists of three primary layers.

1. The Entity Endpoint Layer
2. The Entity Service Layer
3. The Entity Repository Layer

Figure 1 - Entity Anywhere Layers

Each layer has a specific responsibility. Each layer has sublayers. These layers work together to allow for a system that works the same for a given entity regardless of where that entity is stored.

Along with architectural layers, there are other technologies that are critical to Entity Anywhere such as:

1. OData
2. REST
3. Generics
4. Entity Decoupling
5. Interface-based Design
6. Plugin Loading
7. Common or Custom pattern

It is like playing building blocks as a child, only the blocks are well-known pieces in the development world that just need to be stacked together.

A. Generics – Big O Notation Applied to Development Time

Big O notation is often used for both processing time of algorithms and storage needed to complete such processes. However, when applied to software development, not the time to run the code but the time to write code, we can see that Big O can be crucial to helping architectural designs minimize code.

If we have *n* entities and we must write code once for all n entities, then our writing time is O(*n*).

Entity Anywhere needs a lot of code. It needs an OData implementation on top of REST, on top of a service layer on top of a repository layer, and it needs all of that to work for each entity. Writing code for each entity O(*n*) is not sustainable. The linear time trickles downstream to unit tests and maintenance and the entire lifecycle of the code.

Generics is the ability to write code once but use it for many different types, or this case, entities. Generics allows the coding process to occur in constant time O(1). We code one time for all entities.[1]

As is often the case with performance, algorithms that run in constant time often have more preprocessing time and usually somewhat more complexity.

Generics has always been the idea of moving development time from O(n), linear time, to O(1), constant time. Almost all documentation for any language, C#, Java, PHP, C++, etc., discuss generics as allows a user to write one piece of code and reuse it. But not one of them has discussed applying Big O notation to this principle. The term *code reuse* is used as a phrase that developers honor with their blog posts but their implementations are far from it.

The O(n) to O(1) benefits not only decrease the time to write code, but this trickles down to the time to test the code, the maintenance, and the entire lifecycle of the code.

B. The Entity Endpoint Layer

The specific responsibility of this layer is to expose an endpoint for all entities. An *endpoint* is defined as a network service, such as a web service, but it could be any messaging service and is not limited to the web. An *entity endpoint* is an endpoint specific to any given entity where the implementation is not specific, it is generic.

The entity endpoint must specify the data formatting for both input and output of an entity.

Endpoints are already a well-known and established idea. Many of the features of endpoints, such as messaging, serialization, and web protocols are assumed to be known and will not be covered in this document. Entity Anywhere matures the endpoint not only by implementing it with generics but also by implementing OData, which is a protocol designed to be implemented on top of *REST*.

1) The Generic Endpoint

This key feature matures this architecture is the use of generics. As discussed, generics allow for changing the time to write code from linear to constant time. While generics oftentimes increase complexity, that small increase in complexity can lead to a huge reduction in lines of code.

Every entity can be handled by one codebase. The Web service layer behaves the same for every entity. Testing it for one entity tests it for another entity.

2) REST

In the Entity Anywhere architecture, an entity endpoint should be *RESTful*. *REST* (Representational State Transfer) is an architectural style that defines how to perform CRUD actions (create, read, update and delete) on data. This data can be thought of as nothing more than entities.

REST is not limited to HTTP. However, when discussing web services over HTTP, REST suggests the use of HTTP verbs such as GET, POST, PUT, DELETE. These verbs paired with

[1] Attempts to search for academic papers or articles discussing how to decrease development time using Big O have returned no results. This concept may be a new area of research that academia or bloggers could begin to tackle. More importantly, the cost of software development is high. There is a developer shortage. Using Big O to decrease the cost of Software Development and the amount of code needed could benefit the entire industry.

URLs and URL parameters simplify entity data management over a web service. It is not this document's intention to explain REST in detail. Many articles on REST exist already. Instead, this document expresses that REST is one of the building blocks that makes Entity Anywhere possible.

3) OData

OData (Open Data Protocol) is an ISO/IEC approved, OASIS standard that defines a set of best practices for building and consuming RESTful APIs.[2] OData is a specification on how an entity can be queried using URLs and URL parameters as well as a specification for defining entity metadata and relationships between entities.

OData describes many important features. The protocol documentation on OData is quite large and so only highlights can be included in this document.

The first important feature of OData is URL conventions.[3] Some of these important conventions are defined URL parameters for common entity querying. Some of these include:

- *$filter* – The ability to filter a given entity.
- *$expand* – The ability to query not only an entity but a related entity as well.
- *$orderby* – The ability to order the queried entity result set.
- *$top* – the ability to limit the number of results to return.
- *$skip* – the ability to skip a number of elements in the result set.
- *$select* – the ability to select a custom data set from the result set.
- *$count* – the ability to get the count of the result set.
- *$search* – the ability to search an entity in order to get a result set.

The protocol describes the ability to combine the use of these URL parameters into a single call. For example, when $skip is combined with *$top*, all the needs for paging capabilities are met.

Unfortunately, existing implementations of the OData protocol are mostly focused on entities that reside in a single database and if all the entities must be in a single database, an *entity* cannot be *anywhere.*

C. The Entity Service Layer

Every architecture needs a layer where the logic can reside. The service layer's specific goal is to provide a place for logic to go. With the endpoint implementing OData and REST, a lot of supporting logic must exist. The endpoint avoids being cluttered by this logic by forwarding data to this layer is where implementation details reside.

This layer may have many components. For OData, it needs to implement a lot of logic. Examples of this include 1) converting URL parameters to expressions that the repository layer can understand; 2) Expanding the response data to include data from related entities.

This layer is where other layers external to this architecture would connect in.

D. The Repository Layer

The specific responsibility of the repository layer is to abstract the communication with the data store, where ever that may be. One of the earliest references to the repository pattern came in 1993.[4] It was later popularized by the Gang of Four.[1] Unfortunately, the maturing of this pattern has been limited to Object-relational mapping (ORM) frameworks.

Entity Anywhere looks to mature the repository pattern. The key designs of the repository in the Entity Anywhere architecture are:

1. Generic Entity Repository
2. Allowing the Entity data store to be anywhere
3. Commoditizing common data stores

The data store is not limited to a single relational database management system (RDBMS). A repository could also be stored anywhere such as 1) a file, csv, spreadsheet, xml, or other; 2) web services, that may implement its own architecture, 3) static code, 4) web feed, such as RSS or Atom, 5) NoSQL, and many more. It is not the intention of this document to list all possible data stores but to impress upon the reader that a repository is not limited to any one of them.

1) Generic Entity Repository[5]

Combining generics with a repository implementation focusing on entities provides unprecedented code reuse. As stated earlier the code changes from an O(n) linear, implementation to an O(1) constant implementation. One single repository implementation, generically implemented, can serve any number of entities.

There has been pushback against the generic repository pattern. One Software Architect and blogger called generic repositories a lazy antipattern.[6] Many forum posts argue against the idea. However, despite this pushback, efforts to genericize the repository pattern have begun to increase over the past few years.

The primary resistance to generic repositories has come due to RDBMS features. A set of entities represented by tables in an RDBMS are often tightly coupled with foreign key constraints, cascading updates and deletes. These features are critical to self-contained database systems. They are also very mature. The fear of reimplementing an already mature feature

[2] "OData - the best way to REST", HTTP://www.odata.org [Home page]

[3] "OData Version 4.01. Part 2: URL Conventions", HTTP://docs.oasis-open.org/odata/odata/v4.01/cs01/part2-url-conventions/odata-v4.01-cs01-part2-url-conventions.html

[4] [1] David Garlan,Mary Shaw, "An Introduction to Software Architecture," Advances in Software Engineering and Knowledge Engineering, Volume I, edited by V.Ambriola and G.Tortora, World Scientific

[5] Generic repository implementations are rare and difficult to find, as are papers or articles that discus the concept. One of the earliest blog articles my research uncovered was posted in 2011.[5] Since then, articles discussing generic repository implementations have increased, but are still sparse.

[6] Ben Morris, "The generic repository is just a lazy anti-pattern", HTTP://www.ben-morris.com/why-the-generic-repository-is-just-a-lazy-anti-pattern

set has prevented the repository pattern and its implementations from maturing.

III. Plugin Design Patterns

Entity Anywhere uses various design patterns, not just the repository pattern. Other patterns critical for the Entity Anywhere architecture are plugin-related. These include:

1. The Plugin or Module Pattern
2. The Custom or Common Pattern

These patterns provide core features that make it possible to have generic and common code for all entities but still allow for customizations for a specific entity.

A. The Plugin or Module Pattern

The plugin pattern is the ability of a system to load an external module and run that module as part of the system.

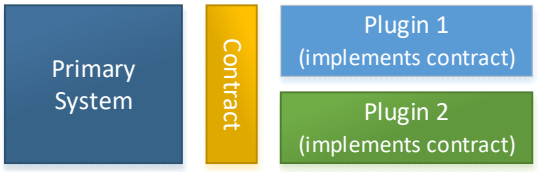

Figure 2 - The Plugin or Module Pattern

The plugin or module pattern allows for each part of this architecture to be modular. Any piece can be replaced. A plugin or module usually implements a contract or an interface. The primary system loads an external module but knows how to work that module because the module implements either a firm contract or less firm documented contract. If the contract is implemented correctly, the module works. Otherwise, the module either fails to load or fails at runtime.

Plugins are usually designed using a *convention*. A convention is a requirement of a program that relies on some external truth. For example, a common plugin convention is that the plugin module must be placed in a specific directory. That convention requires a file system, which means the code relies existence of external systems.

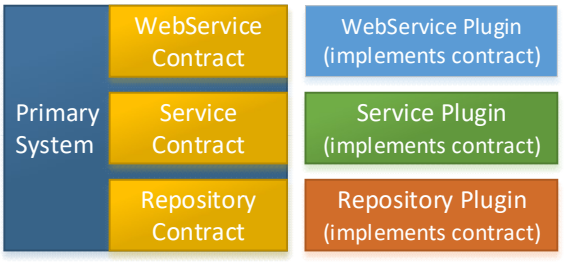

Figure 3 - Entity Anywhere Plugin Architecture

Entity Anywhere uses the plugin pattern to load each of the three entity layers.

B. The Custom or Common Pattern

The custom or common pattern is one of the most commonly used patterns in software engineering, yet is one of the least talked about as an actual pattern.[7]

One common example is a switch/case statement. It has a default case for common code, but based on any given case, the code can be customized.

```
switch(i)
{
    case 1:
    // Custom
    case 2:
    // Custom
    default:
    // Common
}
```

Figure 4 - Switch Statement

Another common example of the custom or common pattern can be found with configuration settings of various systems. These systems are implemented with configuration settings stored outside the code, often in a database or a configuration file. These systems are designed to use common configuration or settings, referred to as *default settings*. When a user changes settings, the default settings are not changed, but instead, the changed settings are stored separately from the common configuration. The code responsible for reading the settings first reads the custom values and if they are set. It then reads the common settings. Alternately, it may load the default settings and then only change the customized values.

Each layer of Entity Anywhere has a common or default implementation. No additional code is needed to implement the three entity layers. However, the code that loads each layer first looks for a custom implementation, and if there is none, falls back to the common implementation.

Figure 5 - Custom or Common Pattern

Like the plugin pattern, the custom or common pattern is often implemented by convention. For the configuration file example described earlier, the convention is often done by a file. One file is a default configuration file. Custom settings are stored in a separate file and the default file is used for any setting not configured in the custom file.

The customization doesn't have to be limited to a single customization. Like a case statement, there could be ordered list of customizations that are tried before settling on common.

There may be customizations that are common to multiple entities or a customization specific to a single entity.

C. Combining Patterns

Both the plugin pattern and the custom or common pattern are critical to Entity Anywhere's layers. The generic, common

[7] Attempts to find a definition for the Custom or Common pattern amongst lists comment patterns, academic papers, or blog articles have not returned results. Perhaps this pattern is listed under an alternate name, though other names, such as fallback pattern, were used in research, and likewise returned no results. One similar pattern is the chain of responsibility pattern. Although, the chain of responsibility is not an exact fit. It may be that this pattern is so basic that defining it has been overlooked.

implementation for each layer would not be enough for some entities that require customization. The inability to customize would be crippling. Without these two patterns working together, customization would be limited.

This limitation is solved by combining the custom or common pattern with both the plugin pattern and inheritance. The system will look for a plugin for a given layer that inherits the common code for that layer. This allows for all the well-known features that come with inheritance. New custom code can be added, existing code can be overridden.

If a custom verb needs to be added to an endpoint, this can be done quite easily by inheriting the common web service. Similarly, the service and repository layers can be customized.

IV. Architecture

The Entity Anywhere architecture can be looked at from two perspectives. A single entity, or all entities.

A. *Single Entity Architecture*

When looking at the single entity architecture, you will see each entity appears to have the following:

1. Entity
2. Entity Interface
3. OData Wrapper
4. Endpoint layer
5. Service layer
6. Repository layer
7. Data storage location

This can be seen in *Figure 6 - Entity Anywhere Architecture*.

Looking at the single-entity architecture shows how a single entity moves up and down the three layers. It shows how each entity is wrapped in an OData wrapper for communication with the endpoint.

However, the single-entity architecture does not demonstrate the generic capabilities. For that, we need to see the multi-entity architecture.

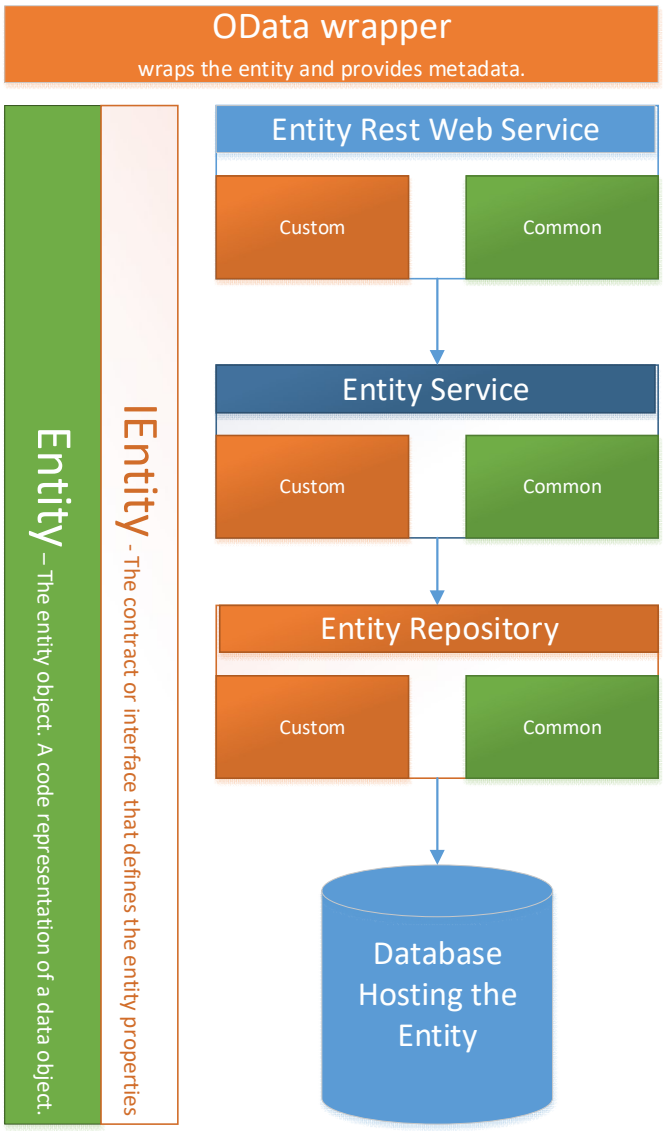

Figure 6 - Entity Anywhere Architecture

B. *Multi-Entity Architecture*

The multi-entity architecture shows that there is a sharing of code for the three layers. The entities pass up and down the three layers.

The intent is to have as few customizations as possible. However, as seen in *Figure 7 - Multi-Entity Architecture*, the repository layer is intended to have the most customizations. Often, these customizations can be shared by many entities.

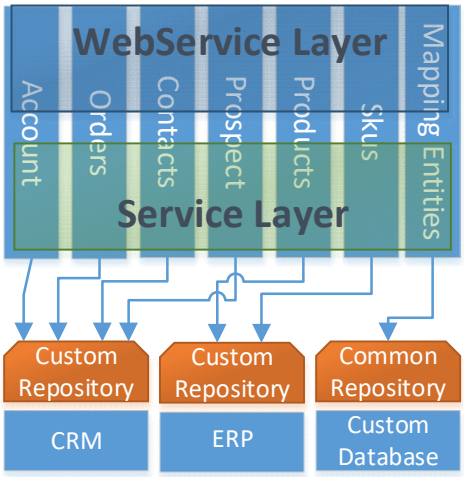

Figure 7 - Multi-Entity Architecture

V. Commodiizing Repositories

The actual repository can be existing systems that are common in the industry, such as Salesforce, SAP, NetSuite or others. The systems that are common depend on the entity in question. For a business-to-business or business-to-consumer entity, the top customer relationship managers (CRM) in the industry would become the top repositories.

Depending on the entity topic, the common industry repositories can be vastly different. A social media platform would have different entities than a shopping cart. A fiction publisher would have different entities than a hospital.

How many times does the repository layer need to be written for any given CRM? The answer is once.

Once repositories have been created for the top common systems for an entity topic, it becomes a configuration action to replace a repository.

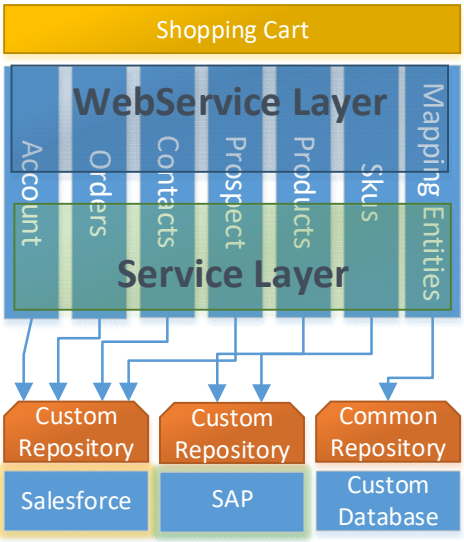

Figure 8 - Shopping Cart on Entity Anywhere #1

The above image shows how a shopping cart could be written to run on Entity Anywhere. This shopping cart has a list of entities and uses repositories that are common to the industry.

Another company could implement the same shopping cart but simply have different entity repositories.

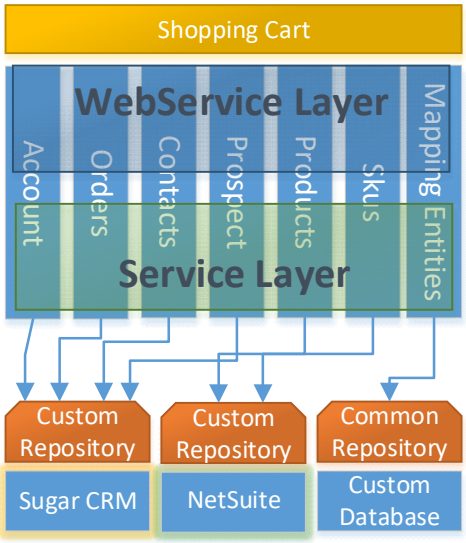

Figure 9 - Shopping Cart on Entity Anywhere #2

The above solutions require:

1. That Entity Anywhere is fully implemented
2. That a shopping cart is built on top of Entity Anywhere.
3. That commoditized repositories exist for entities.

Systems such as a CRM allow for customizations, so the commoditized repository must allow for customizations as well, which enforces the need for source code.

VI. Math

There exist various components to developing this solution and each one can be considered with Big O notation.

n	number of entities
k	number of entities that need a customization
k_s	number of entities that share the same customization
s	The number of customizations that are shared.

Work to write code	Generic code	Code per entity
Entity Definition	O(n)	O(n)
Endpoint Layer	O(1)	O(n)
Service Layer	O(1)	O(n)
Repository Layer	O(1)	O(n)
Customizations	O(k)	O(k)
Shared customizations	O(1 + s)	O($k_s * s$)

Figure 10 – Time to Write Code in Big O

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley Professional; 1 edition (November 10, 1994)

[2] C. Pratt, A Truly Generic Repository, HTTPs://cpratt.co/truly-generic-repository

[3] T. Ugurlu, Generic Repository Pattern - Entity Framework, ASP.NET MVC and Unit Testing Triangle, HTTP://www.tugberkugurlu.com/archive/generic-repository-pattern-entity-framework-asp-net-mvc-and-unit-testing-triangle

[4] J. A. Barneck, “Entity Anywhere Architecture.vsdx”, HTTPs://github.com/rhyous/EntityAnywhere [Documentation in Source Repository]

Jared A. Barneck is a Principal Software Architect at Ivanti in South Jordan, Utah. His undergraduate degree is in English, creative writing, from Brigham Young University in Provo, Utah, Unitest States. He is mostly a self-taught software developer and architect and has been a technical blogger since 2009. Along with his self-study, he did take various undergraduate computer science courses. He is currently studying for a Master of Science in computer science from Utah State University in Salt Lake City, Utah, United States.