**Registry API: Developer Manual**

Welcome! This manual is designed to help you understand the architecture, concepts, and technologies used in the Registry API project. The goal is to get you comfortable navigating the codebase and contributing effectively.

**1. Introduction**

* Purpose: The Registry API provides a way for client applications (like a web frontend) to interact with the RegistryDatabase. It allows users to Create, Read, Update, and Delete (CRUD) information about data specifications, including their core elements and extension components.
* Goal of this Manual: To explain the fundamental building blocks of this API, how the code is organized, and the patterns used.

**2. Core Technologies**

* .NET 8: The latest Long-Term Support (LTS) version (at the time of writing) of the .NET platform. We utilize its features for building modern, performant applications.
* ASP.NET Core: The framework used to build the web API. It handles incoming HTTP requests, routing, model binding, and sending responses.
* Entity Framework Core (EF Core): The Object-Relational Mapper (ORM) used to interact with the SQL Server database. It allows us to work with database tables using C# objects.
* SQL Server: The relational database management system where the application's data is stored.
* C#: The primary programming language used throughout the application.

**3. Architecture Overview: The Layered Approach**

This application follows a common layered architecture pattern to separate concerns and make the code more organized, testable, and maintainable.

+-------------------+ +----------------+ +-----------------+ +----------------+

| Client (e.g., | ---> | Controller | ---> | Service | ---> | Repository | ---> Database (SQL Server)

| Web App/Mobile)| | (ASP.NET Core) | | (Business Logic)| | (Data Access) | +----------------+

+-------------------+ +----------------+ +-----------------+ +----------------+

| | |

V V V

Handles HTTP Orchestrates Interacts with

Requests/Responses Business Rules EF Core/DbContext

* Controllers (/Controllers): Handle incoming HTTP requests (e.g., GET /api/specifications/5). They parse input, call the appropriate service method, and format the response (e.g., JSON). They *don't* contain business logic or direct database access.
  + *Example:* SpecificationsController.cs
* Services (/Services): Contain the core business logic. They orchestrate operations, validate data according to business rules, coordinate calls to one or more repositories, and map data between DTOs and database models.
  + *Example:* SpecificationService.cs, ISpecificationService.cs
* Repositories (/Repositories): Responsible *only* for data access. They abstract the underlying data store (EF Core in this case). They contain methods for querying and saving data (e.g., GetByIdAsync, AddAsync). They work directly with the DbContext and DbSet.
  + *Example:* SpecificationIdentifyingInformationRepository.cs, ISpecificationIdentifyingInformationRepository.cs
* Database: The SQL Server instance holding the actual data.

**4. Project Structure (**RegistryApi/**)**

The code is organized into folders based on its responsibility, aligning with the layered architecture:

* /Controllers: API endpoint handlers.
* /Data: Contains the RegistryDbContext.
* /DTOs: Data Transfer Objects used for API communication.
* /Helpers: Utility classes like PagedList.
* /Mappings: AutoMapper profiles for object mapping.
* /Models: C# classes representing database tables (Entities).
* /Repositories: Data access logic implementation and interfaces.
* /Services: Business logic implementation and interfaces.
* Program.cs: Application startup, configuration, and dependency injection setup.
* appsettings.json: Configuration settings (like connection strings).

**5. Key Concepts & Patterns**

**a. REST API Principles**

The API follows REST (Representational State Transfer) principles:

* Endpoints: URLs represent resources (e.g., /api/specifications, /api/specifications/5/coreElements).
* HTTP Verbs: Standard HTTP methods map to CRUD operations:
  + GET: Retrieve data.
  + POST: Create new data.
  + PUT: Update existing data (replace).
  + DELETE: Remove data.
* Status Codes: Standard HTTP status codes indicate the outcome:
  + 200 OK: Successful GET.
  + 201 Created: Successful POST.
  + 204 No Content: Successful PUT or DELETE (no body returned).
  + 400 Bad Request: Invalid input from the client.
  + 404 Not Found: Resource doesn't exist.
  + 409 Conflict: Action cannot be completed due to current state (e.g., deleting a specification with children).
* Statelessness: Each request contains all information needed to process it.

*Example (SpecificationsController.cs):*

[HttpGet("{id:int}")] // GET /api/specifications/5

public async Task<Results<Ok<SpecificationIdentifyingInformationDetailDto>, NotFound>> GetSpecification(...)

{ ... }

[HttpPost] // POST /api/specifications

public async Task<Results<Created<SpecificationIdentifyingInformationHeaderDto>, BadRequest<string>>> PostSpecification(...)

{ ... }

[HttpDelete("{id:int}")] // DELETE /api/specifications/5

public async Task<Results<NoContent, NotFound, Conflict<string>>> DeleteSpecification(...)

{ ... }

**b. Dependency Injection (DI)**

Instead of classes creating their own dependencies (like new SpecificationService()), dependencies are "injected" (usually through constructors). ASP.NET Core has built-in support for DI. This makes code more modular, testable, and flexible.

*Example (Program.cs):* Registering services and repositories tells the DI container how to create them when needed.

// Registering the service and its dependencies

builder.Services.AddScoped<ISpecificationIdentifyingInformationRepository, SpecificationIdentifyingInformationRepository>();

// ... other repositories

builder.Services.AddScoped<ISpecificationService, SpecificationService>();

*Example (SpecificationService.cs using primary constructor):* The service *receives* its dependencies.

public class SpecificationService(

ISpecificationIdentifyingInformationRepository specInfoRepo,

ISpecificationCoreRepository specCoreRepo,

ISpecificationExtensionComponentRepository specExtRepo,

IMapper mapper // AutoMapper is also injected

) : ISpecificationService

{

// ... methods use specInfoRepo, specCoreRepo etc. ...

}

**c. Repository Pattern**

This pattern isolates the data access logic. Services depend on repository *interfaces* (e.g., ISpecificationCoreRepository), not concrete implementations. This allows:

* Decoupling: Services don't know *how* data is stored/retrieved (EF Core, Dapper, etc.).
* Testability: You can easily "mock" the repository interface when testing the service logic without needing a real database.

*Example (ISpecificationCoreRepository.cs):* The contract.

public interface ISpecificationCoreRepository : IGenericRepository<SpecificationCore>

{

Task<PagedList<SpecificationCore>> GetBySpecificationIdPaginatedAsync(int specificationId, PaginationParams paginationParams);

// ... other methods

}

*Example (SpecificationCoreRepository.cs):* The EF Core implementation.

public class SpecificationCoreRepository(RegistryDbContext context)

: GenericRepository<SpecificationCore>(context), ISpecificationCoreRepository

{

public async Task<PagedList<SpecificationCore>> GetBySpecificationIdPaginatedAsync(...)

{

var query = \_dbSet.Where(sc => sc.IdentityID == specificationId)...;

// ... uses \_dbSet (from DbContext) ...

}

}

*Example (SpecificationService.cs):* The service uses the interface.

public class SpecificationService(

// ... other dependencies

ISpecificationCoreRepository specCoreRepo, // Depends on the interface

// ...

) : ISpecificationService

{

public async Task<PaginatedSpecificationCoreResponse?> GetSpecificationCoresAsync(...)

{

var pagedEntities = await specCoreRepo.GetBySpecificationIdPaginatedAsync(...); // Calls the interface method

// ...

}

}

**d. Unit of Work (Implicit via DbContext)**

Ensures that a series of operations (potentially across multiple repositories using the *same* DbContext instance) are treated as a single transaction. They either all succeed or all fail. EF Core's DbContext provides this implicitly. When you call SaveChangesAsync() on the DbContext (often via a repository method), EF Core wraps all tracked changes (adds, updates, deletes) since the last save within a database transaction.

*Example (GenericRepository.cs):* The central save method.

public async Task<bool> SaveChangesAsync()

{

// All changes tracked by this \_context instance will be saved in one transaction

return (await \_context.SaveChangesAsync()) > 0;

}

*Example (SpecificationService.cs):* Calling SaveChangesAsync after adding an entity.

public async Task<(ServiceResult Status, SpecificationCoreDto? Dto)> AddCoreElementAsync(...)

{

// ... validation ...

var entity = mapper.Map<SpecificationCore>(createDto);

await specCoreRepo.AddAsync(entity); // Tracks the addition

bool saved = await specCoreRepo.SaveChangesAsync(); // Commits the transaction

// ...

}

**e. Object-Relational Mapping (ORM) with EF Core**

EF Core maps database tables to C# classes (Models/Entities) and allows querying the database using LINQ (Language Integrated Query) instead of writing raw SQL.

*Example (RegistryDbContext.cs):* Defining the mapping.

public DbSet<SpecificationIdentifyingInformation> SpecificationIdentifyingInformations => Set<SpecificationIdentifyingInformation>();

// ... other DbSets ...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// Configures table names, keys, relationships etc.

modelBuilder.Entity<SpecificationCore>()

.HasOne(sc => sc.CoreInvoiceModel)

.WithMany(cim => cim.SpecificationCores)

.HasForeignKey(sc => sc.BusinessTermID);

// ...

}

*Example (SpecificationCoreRepository.cs):* Using LINQ to query.

public async Task<SpecificationCore?> GetByIdAndSpecificationIdAsync(int coreElementId, int specificationId)

{

// LINQ query translated by EF Core into SQL

return await \_dbSet.FirstOrDefaultAsync(sc => sc.EntityID == coreElementId && sc.IdentityID == specificationId);

}

**f. Data Transfer Objects (DTOs)**

Plain classes or records used to transfer data between layers, especially between the Service and the Controller (and thus, the API client). They help:

* Decouple: Prevent exposing internal database models directly via the API.
* Shape Data: Provide only the necessary data for a specific API endpoint.
* Prevent Over/Under-Posting: Define exactly what data can be sent/received.
* Versioning: Allow API DTOs to evolve independently of database models.

*Example (DTOs/SpecificationCoreDtos.cs):* Defines different shapes for creating, updating, and retrieving core specification data.

// For creating (doesn't include IDs generated by DB)

public record SpecificationCoreCreateDto(...);

// For retrieving (includes IDs)

public record SpecificationCoreDto(int EntityID, int IdentityID, ...);

// For paginated responses

public record PaginatedSpecificationCoreResponse(PaginationMetadata Metadata, List<SpecificationCoreDto> Items);

*Example (SpecificationService.cs):* Mapping between Model and DTO using AutoMapper.

public async Task<PaginatedSpecificationCoreResponse?> GetSpecificationCoresAsync(...)

{

var pagedEntities = await specCoreRepo.GetBySpecificationIdPaginatedAsync(...); // Gets Model objects

var dtos = mapper.Map<List<SpecificationCoreDto>>(pagedEntities.Items); // Maps Models to DTOs

// ... return PaginatedSpecificationCoreResponse containing DTOs ...

}

**g. Asynchronous Programming (**async**/**await**)**

Used extensively for I/O-bound operations (like database calls, network requests). async and await prevent blocking threads while waiting for these operations to complete, improving application scalability and responsiveness.

*Example (SpecificationService.cs):*

// Method marked async, returns a Task

public async Task<bool> UpdateSpecificationAsync(int id, SpecificationIdentifyingInformationUpdateDto updateDto)

{

// await pauses execution here until GetByIdAsync completes, without blocking the thread

var entity = await specInfoRepo.GetByIdAsync(id);

if (entity == null) return false;

// ... mapping ...

specInfoRepo.Update(entity);

// await pauses again for the database save operation

return await specInfoRepo.SaveChangesAsync();

}

**h. Pagination**

Breaking down large lists of data into smaller "pages" for efficient retrieval and display.

*Example (Helpers/PagedList.cs):* A helper class to hold paged data and metadata.

public class PagedList<T>(List<T> items, int count, int pageNumber, int pageSize) { ... }

*Example (Helpers/PaginationParams.cs):* Class to receive pagination parameters from the API query string.

public class PaginationParams { public int PageNumber { get; set; } = 1; public int PageSize { get; set; } = 10; }

*Example (SpecificationsController.cs):* Receiving parameters and returning paginated results.

[HttpGet] // GET /api/specifications?pageNumber=2&pageSize=20

public async Task<Ok<PaginatedSpecificationHeaderResponse>> GetSpecifications([FromQuery] PaginationParams paginationParams)

{

var result = await specificationService.GetSpecificationsAsync(paginationParams);

return TypedResults.Ok(result); // result includes metadata and items for the page

}

**i. .NET 8 Features Used**

* Primary Constructors: Simplify class definitions by declaring constructor parameters directly in the class declaration (used in Repositories, Services, DbContext, some DTOs/Helpers).
  + *Example:* public class SpecificationService(ISpecificationRepo repo, ...) : ISpecificationService
* required Modifier: Ensures non-nullable properties are initialized (used in Models and some DTOs).
  + *Example:* public required string SpecificationIdentifier { get; set; }
* Collection Expressions ([]): Concise syntax for initializing collections.
  + *Example:* public virtual ICollection<SpecificationCore> SpecificationCores { get; set; } = [];
* TypedResults: Provides strongly-typed results for minimal API endpoints (used in SpecificationsController), improving code clarity and testability compared to returning IActionResult.
  + *Example:* return TypedResults.Ok(result);, return TypedResults.NotFound();

**6. Data Models & Relationships**

The core data revolves around SpecificationIdentifyingInformation. Each specification can have multiple SpecificationCore entries (linking to CoreInvoiceModel) and multiple SpecificationExtensionComponent entries (linking to ExtensionComponentModelElement, which in turn links to ExtensionComponentsModelHeader).

These relationships are defined in the Model classes using navigation properties (ICollection<>, virtual properties) and configured in RegistryDbContext.OnModelCreating using EF Core's Fluent API to set up foreign keys, primary keys, unique constraints, and delete behaviors (like cascade delete or restrict).

*Example (RegistryDbContext.OnModelCreating):*

// Composite foreign key from SpecificationExtensionComponent to ExtensionComponentModelElement

modelBuilder.Entity<SpecificationExtensionComponent>()

.HasOne(specExt => specExt.ExtensionComponentModelElement) // Navigation property in SpecExtComp

.WithMany(elem => elem.SpecificationExtensionComponents) // Inverse navigation property in ExtCompElement

.HasForeignKey(specExt => new { specExt.ExtensionComponentID, specExt.BusinessTermID }) // FK columns in SpecExtComp

.HasPrincipalKey(elem => new { elem.ExtensionComponentID, elem.BusinessTermID }) // Unique key columns in ExtCompElement

.OnDelete(DeleteBehavior.Restrict); // Don't delete element if used in a spec

**7. Workflow Example: Get Specification by ID (**GET /api/specifications/{id}**)**

1. Request: A client sends GET /api/specifications/5?corePageNumber=1&extPageSize=10.
2. Routing: ASP.NET Core routes the request to the GetSpecification method in SpecificationsController.
3. Controller:
   * Receives id = 5, coreParams, and extParams.
   * Calls \_specificationService.GetSpecificationByIdAsync(5, coreParams, extParams).
4. Service (GetSpecificationByIdAsync):
   * Calls \_specInfoRepo.GetByIdAsync(5) to get the main SpecificationIdentifyingInformation entity. Returns null if not found.
   * If found, calls GetSpecificationCoresAsync(5, coreParams) and GetSpecificationExtensionsAsync(5, extParams) (which internally call their respective repositories with pagination).
   * Maps the results (header entity and paginated child DTOs) into a single SpecificationIdentifyingInformationDetailDto.
   * Returns the DTO to the controller.
5. Repository (GetByIdAsync):
   * Uses \_context.SpecificationIdentifyingInformations.FindAsync(5) (or similar EF Core method).
   * EF Core generates SQL (SELECT \* FROM SpecificationIdentifyingInformation WHERE IdentityID = 5).
   * Returns the entity (or null) to the service.
6. Controller:
   * Receives the DTO (or null) from the service.
   * If null, returns TypedResults.NotFound() (HTTP 404).
   * If DTO exists, returns TypedResults.Ok(dto) (HTTP 200 with the DTO serialized as JSON in the response body).
7. Response: The client receives the HTTP 200 or 404 response.

**8. Setup & Running**

Refer to the separate C# Specification API: Setup and File Organization Guide (.NET 8+) document for detailed steps on setting up the project, installing packages, configuring the connection string, and running the application.

**9. Further Learning**

* Microsoft Learn: Excellent resources for .NET, ASP.NET Core, EF Core, and C#.
* ASP.NET Core Documentation: Official documentation covering controllers, middleware, DI, etc.
* Entity Framework Core Documentation: Official documentation for DbContext, LINQ queries, migrations, etc.
* Repository Pattern: Search for articles and examples online.
* REST API Design Best Practices: Understand common conventions.

Don't hesitate to ask senior developers questions as you explore the code! Good luck!