# Technical Design Specification: A Modular ERP System with a.NET 9 Plugin Architecture

## Foundational Architecture

This section establishes the high-level architectural vision and the core principles that will govern all subsequent design decisions. It provides the rationale behind the technical choices, grounding the design in established patterns and business objectives to create a robust, scalable, and maintainable Enterprise Resource Planning (ERP) system.

### 1.1 The Modular Monolith Pattern: A Pragmatic Approach

The foundational architectural pattern selected for this ERP system is the **Modular Monolith**. This architecture organizes the application into a collection of distinct, independently developable modules—realized as plugins—that are ultimately deployed together as a single process. This approach provides many of the development-time benefits associated with microservices, such as team autonomy, clear domain boundaries, and strong encapsulation, but critically, it avoids the significant operational complexity inherent in distributed systems.

The structure of ERP systems naturally lends itself to this modular decomposition. An ERP is not a single, monolithic business function but rather a suite of integrated applications that handle specific domains like finance, procurement, supply chain management, and human resources.1 The Modular Monolith pattern allows for a direct mapping of these discrete business capabilities to well-defined, cohesive code modules.

The rationale for this choice is rooted in striking an optimal balance between maintainability and deployment simplicity, which is ideal for the scope of a foundational ERP. A microservices architecture would introduce substantial overhead related to network latency, distributed transactions, service discovery, and complex deployment orchestration. For this system, such complexity would be premature and counterproductive. The Modular Monolith, by contrast, allows business processes that span multiple modules to execute within a single process, leveraging shared, in-memory communication and unified, ACID-compliant database transactions.

This architectural decision has profound implications beyond the code itself; it directly influences and enables a more agile organizational structure. By aligning technical modules with business domains, it becomes possible to form autonomous, feature-aligned development teams (e.g., a "Finance Team" or an "Inventory Team"). Each team can take full ownership of its module, from the database schema to the user interface, fostering expertise and accountability. This structure significantly enhances development velocity, as the Finance team can develop and test a new feature within their module without requiring a tightly coordinated, high-risk "big bang" release with the Inventory team. The architecture, therefore, becomes a direct enabler of a scalable and efficient development process, delivering tangible business value through increased agility.

### 1.2 System Components Overview

The system is composed of three primary architectural components, working in concert to create a cohesive and extensible application.

1. **Host Application Shell:** This is the primary executable process, an ASP.NET Core 9 Web Application. Its responsibilities are strictly limited to cross-cutting concerns and platform-level services. It manages the application startup and shutdown, provides core services like security (authentication and authorization) and logging, handles configuration, and, most importantly, orchestrates the entire plugin lifecycle. It is the stable foundation upon which all business functionality is dynamically loaded.
2. **Shared Kernel:** This is a small, central class library that contains the minimal set of code shared across the Host and all plugin modules. Its scope is intentionally and aggressively restricted to prevent it from becoming a "common" library that fosters high coupling. The Shared Kernel contains only the contracts (C# interfaces) that define how plugins interact with the host and with each other, shared data transfer objects (DTOs) for events, and a few core, ubiquitous data entities such as User or Tenant. It defines the rules of engagement but contains no business logic.
3. **Plugin Modules:** These are dynamically loaded.NET assemblies, each representing a distinct business capability (e.g., Finance, Inventory, Procurement). Each plugin is a self-contained, vertical slice of functionality. It encapsulates its own data models, entity configurations, business logic, services, API endpoints, and UI components. Following the principle of strong encapsulation, plugins are strictly forbidden from having direct, compile-time references to one another.3

This structure ensures a clean separation of concerns, where the Host provides the "how" (the platform) and the plugins provide the "what" (the business features).

### 1.3 Core Design Principles

The architecture is governed by a set of inviolable principles designed to ensure long-term maintainability, scalability, and extensibility.

1. **Strong Encapsulation:** As stated, plugins must not have direct compile-time references to each other. This is the cornerstone of the architecture. A developer working on the Finance plugin should be physically unable to directly call a method in the Inventory plugin. This forces all inter-module communication to occur through well-defined, indirect mechanisms, preventing the creation of a "distributed big ball of mud." This principle is critical for allowing modules to evolve and be updated independently without causing cascading failures.3
2. **Explicit Contracts:** All interactions are mediated through clear, stable interfaces defined in the Shared Kernel. A plugin integrates with the Host by implementing interfaces like IPluginModule. It communicates with other plugins by publishing and subscribing to event DTOs defined in the Shared Kernel. This contract-based approach ensures loose coupling. As long as the contract remains stable, the internal implementation of a plugin can be completely refactored or replaced without impacting any other part of the system.
3. **Asynchronous, Event-Driven Communication:** To maintain decoupling and support complex business workflows, processes that cross module boundaries are orchestrated via asynchronous, in-process events. For example, when an invoice is paid in the Finance module, it does not directly command the Inventory module to ship a product. Instead, it publishes an InvoicePaidEvent. The Inventory module, and potentially other modules like Reporting or Analytics, can subscribe to this event and react accordingly. This "publish-subscribe" model ensures that the Finance module has no knowledge of its downstream dependents, making the system highly extensible. New functionality, such as sending a customer satisfaction survey after payment, can be added by simply creating a new plugin that subscribes to the InvoicePaidEvent, requiring zero changes to the original Finance plugin.5

## The Host Application Shell

The Host Application Shell serves as the central nervous system of the ERP. It is not responsible for any specific business logic but provides the essential runtime environment, services, and infrastructure required to discover, load, and integrate the disparate plugin modules into a single, functional application.

### 2.1 Solution and Project Structure

A well-defined project structure is essential for maintaining architectural integrity and clarity. The solution will be organized to reflect the separation of concerns between the host, the shared contracts, and the individual plugin modules. This structure is foundational for correctly implementing the plugin loading mechanism and managing dependencies.4

| Project Name | Description | Key Dependencies |
| --- | --- | --- |
| ERP.sln | The root solution file for the entire application. | N/A |
| src/ERP.Host/ERP.Host.csproj | The main ASP.NET Core 9 web application project. Acts as the executable shell, manages plugin loading, and provides the main UI layout. | ERP.SharedKernel |
| src/ERP.SharedKernel/ERP.SharedKernel.csproj | A class library containing the interfaces, event DTOs, and core entities shared by the host and all plugins. Contains no implementation logic. | None |
| src/plugins/Finance/ERP.Plugin.Finance/ERP.Plugin.Finance.csproj | A class library representing the Finance module. Contains all finance-related logic, data models, UI, and API endpoints. | ERP.SharedKernel |
| src/plugins/Inventory/ERP.Plugin.Inventory/ERP.Plugin.Inventory.csproj | A class library representing the Inventory module. Contains all inventory-related logic, data models, UI, and API endpoints. | ERP.SharedKernel |
| tests/ERP.Plugin.Finance.Tests/ERP.Plugin.Finance.Tests.csproj | Unit and integration tests for the Finance plugin. | ERP.Plugin.Finance |

This structure makes the architectural layering explicit. It is immediately clear that the Host and all plugins depend on the Shared Kernel, but crucially, there are no dependencies between the plugin projects themselves.

### 2.2 Technology Stack and Core Services

The Host will be an **ASP.NET Core 9 Blazor Web App** configured for **Interactive Auto Render Mode**. This modern hosting model, refined in.NET 9, offers an optimal user experience by combining the strengths of both server-side and client-side rendering. Initial page requests are rendered statically on the server (SSR), resulting in extremely fast load times and excellent SEO characteristics. Once the initial page is delivered, the Blazor framework establishes a real-time SignalR connection or downloads a WebAssembly runtime to enable rich, client-side interactivity for subsequent user actions. This hybrid approach provides the performance of a traditional server-rendered application with the rich user experience of a single-page application (SPA).6

The Host is responsible for bootstrapping and providing a set of core, cross-cutting services available to all plugins via dependency injection:

* **Authentication & Authorization:** The system will use ASP.NET Core Identity for robust and secure user management. The Host will configure the authentication middleware, providing a unified security context (HttpContext.User) that is accessible to all plugins for authorization checks..NET 9 introduces simplified mechanisms for serializing the authentication state, which is particularly beneficial in Blazor Web Apps for maintaining user identity across different rendering modes.7
* **Logging & Telemetry:** A centralized logging framework (e.g., Serilog or the built-in ILogger) will be configured in the Host. This allows log events from the Host and all loaded plugins to be captured, enriched with contextual information (like a correlation ID), and routed to a unified sink (e.g., console, file, or a log aggregation service like Seq or Application Insights).
* **Configuration:** The Host will provide a unified IConfiguration source, built from appsettings.json, environment variables, and other providers. This allows plugins to access their specific configuration settings in a decoupled manner using the IOptions<T> pattern.

### 2.3 The Plugin Management Subsystem

The heart of the Host application is the PluginManager service, which is responsible for the entire plugin lifecycle. This subsystem is a bespoke implementation of the patterns described in official Microsoft guidance for creating pluggable applications in.NET.4

1. **Discovery:** At application startup, the PluginManager scans a designated plugins directory within the application's root folder. It searches for assembly files (.dll) that are potential plugin entry points.
2. **Loading:** For each discovered plugin assembly, the manager creates a dedicated PluginLoadContext. This custom class, which inherits from System.Runtime.Loader.AssemblyLoadContext, is the cornerstone of the.NET plugin model. It provides a strong isolation boundary for the plugin and its dependencies. The PluginLoadContext uses an AssemblyDependencyResolver instance, which reads the plugin's .deps.json file to locate and load its specific dependencies. This mechanism is what prevents "DLL Hell," allowing two different plugins to use conflicting versions of the same third-party library without affecting each other or the host application.4
3. **Instantiation:** Once an assembly is loaded into its isolated context, the PluginManager uses reflection to scan its types. It looks for public, concrete classes that implement the core IPluginModule interface from the Shared Kernel. This discovery and instantiation process is the practical realization of the conceptually-named RuntimePluggableClassFactory. For each valid implementation found, an instance is created using Activator.CreateInstance.
4. **Integration:** With a plugin instance in hand, the PluginManager orchestrates its integration into the running application. It calls a series of methods defined on the IPluginModule interface, such as Initialize(IServiceCollection services) and Configure(IApplicationBuilder app). These methods allow the plugin to register its services with the host's dependency injection container, map its API endpoints to the ASP.NET Core routing table, and contribute its entity configurations to the data context.

The design of this loading mechanism has a critical and non-obvious requirement that dictates the entire build and deployment strategy. For the type system to work correctly across the isolation boundary of the AssemblyLoadContext, the host and the plugin must share the *exact same instance* of the Shared Kernel assembly. If the plugin were to bring its own copy of ERP.SharedKernel.dll, the.NET runtime would treat IPluginModule from the host's context and IPluginModule from the plugin's context as two completely different types, even though their code is identical. This would result in an InvalidCastException when the host attempts to use the plugin's object. To prevent this, the plugin's project file must be configured to reference the Shared Kernel with <Private>false</Private> and <ExcludeAssets>runtime</ExcludeAssets>. This seemingly minor configuration detail is the linchpin of the entire architecture, as it prevents the shared contract DLL from being copied to the plugin's output directory.4 This, in turn, necessitates a carefully designed build and packaging process to ensure this separation is strictly maintained during deployment, a topic addressed in Section 7.

### 2.4 The Shared Kernel and UI Shell

The ERP.SharedKernel.csproj project is the architectural center of the system, defining the contracts that enable modularity. Its contents are minimal by design to reduce coupling.

The UI Shell, residing within the Host application, provides the global user interface structure. Key components like App.razor and MainLayout.razor define the overall page template, including elements like the header, footer, and a primary navigation sidebar. This navigation menu is not static; it is dynamically populated at runtime. The PluginManager discovers any loaded plugins that implement an INavigationProvider interface. It then invokes a method on this interface to retrieve a list of navigation links, which are rendered into the sidebar. This allows the Finance plugin to add "Invoices" and "Ledger" links, and the Inventory plugin to add a "Products" link, all without the Host having any compile-time knowledge of these features. The main content area of the layout acts as a placeholder, rendering the specific Blazor components provided by the currently active plugin, creating a seamless user experience.10

## Plugin Development Contracts and Communication

This section defines the formal "rules of engagement" for plugins. These contracts are the API of the modular system, ensuring that modules developed by different teams can be integrated seamlessly and predictably into the host application while remaining fully decoupled from one another.

### 3.1 Core Plugin Interfaces (The Shared Kernel)

The ERP.SharedKernel project contains a set of C# interfaces that constitute the formal contract between the Host and its plugins. Implementing these interfaces is how a plugin "plugs in" to the host's lifecycle and contributes its functionality.

| Interface Name | Method/Property | Description | Implemented By |
| --- | --- | --- | --- |
| IPluginModule | void Initialize(IServiceCollection services, IConfiguration config) | The primary entry point for a plugin. Called during startup to allow the plugin to register its services (e.g., business logic, repositories) with the host's dependency injection container. | FinancePluginModule, InventoryPluginModule |
| IEndpointRegistrar | void MapEndpoints(IEndpointRouteBuilder builder) | Allows a plugin to register its web API endpoints. The host passes its IEndpointRouteBuilder, and the plugin defines its routes (e.g., builder.MapGet("/api/products",...)). | FinancePluginModule, InventoryPluginModule |
| INavigationProvider | IEnumerable<NavItem> GetNavItems() | Enables a plugin to contribute links to the main application's navigation UI. Returns a collection of objects defining the link text, URL, and icon. | FinanceNavProvider, InventoryNavProvider |
| IEntityTypeConfigurationProvider | IEnumerable<object> GetEntityTypeConfigurations() | Allows a plugin to contribute its EF Core entity configurations to the host's DbContext. Returns a collection of IEntityTypeConfiguration<T> instances. | FinancePluginModule, InventoryPluginModule |

These contracts provide a standardized, discoverable way for plugins to integrate their services, APIs, UI elements, and data models into the application runtime.

### 3.2 Inter-Plugin Communication Strategy: A Mediator-Based Event Bus

To strictly enforce the "no direct references" principle, all inter-plugin communication is handled by an in-process, mediator-based event bus. This pattern ensures that when one module needs to trigger a business process in another, it does so indirectly, without any compile-time knowledge of the receiving module. A library like MediatR is well-suited for this, but a custom implementation can also be created.

The communication flow is as follows:

1. **Event Definition:** A business event is defined as a plain DTO class in the Shared Kernel (e.g., public record InvoicePaidEvent(Guid InvoiceId, Guid CustomerId);).
2. **Publishing Events:** A service within a plugin (e.g., the InvoiceService in the Finance module) can inject an IEventPublisher service (provided by the host). After completing a unit of work, it publishes the event: await \_eventPublisher.PublishAsync(new InvoicePaidEvent(...));.
3. **Handling Events:** In another plugin (e.g., the Inventory module), a class can implement the corresponding IEventHandler<T> interface: public class ShipOrderHandler : IEventHandler<InvoicePaidEvent>.
4. **Mediation:** During startup, the Host's PluginManager uses reflection to discover all IEventHandler<T> implementations across all loaded plugins. It registers these handlers with the central mediator service. When the InvoiceService publishes the InvoicePaidEvent, the mediator looks up all registered handlers for that specific event type and invokes them.

This architecture ensures total decoupling. The Finance plugin is completely unaware of what happens after an invoice is paid. This allows for immense flexibility; a new "Reporting" plugin could be added later to also subscribe to InvoicePaidEvent to update analytics dashboards, requiring zero modifications to the Finance plugin.5

| Event Name | Payload (Data Transfer Object) | Publishing Module(s) | Subscribing Module(s) |
| --- | --- | --- | --- |
| InvoicePaidEvent | { Guid InvoiceId, Guid CustomerId, List<LineItemDto> Items } | Finance | Inventory, Reporting |
| StockLevelLowEvent | { Guid ProductId, Guid WarehouseId, int CurrentQuantity } | Inventory | Procurement |
| NewProductAddedEvent | { Guid ProductId, string Name, decimal Price } | Inventory | Finance (to set up accounting codes) |
| PurchaseOrderApprovedEvent | { Guid PurchaseOrderId, Guid VendorId } | Procurement | Finance (to create accounts payable entry) |

This catalog makes the implicit, event-driven business workflows explicit, serving as critical documentation for understanding the system's cross-module interactions.

### 3.3 Shared Data Access and Transaction Management

While modules are logically separate, a core requirement of most ERP systems is a single, transactionally consistent database.2 The architecture supports this through a shared data access strategy.

1. **Shared DbContext:** The Host application defines and owns the primary ErpDbContext class. This class is registered as a scoped service in the dependency injection container.
2. **Plugin Model Contribution:** Plugins do not define their own DbContext. Instead, they contribute their entity configurations to the host's context. Each plugin implements the IEntityTypeConfigurationProvider interface, which returns its collection of EF Core entity configuration classes (e.g., public class InvoiceConfiguration : IEntityTypeConfiguration<Invoice>).
3. **Dynamic Model Building:** During startup, the Host iterates through all loaded plugins, calls GetEntityTypeConfigurations(), and applies all returned configurations to its DbContext within the OnModelCreating method. This dynamically builds a complete database model that is a composite of the schemas from all installed plugins.
4. **Transactional Integrity:** Because all data access from all plugins is funneled through the single, host-managed ErpDbContext instance for a given HTTP request, business processes that span multiple modules can be executed within a single database transaction. For example, when an InvoicePaidEvent is handled by the Inventory plugin, any database modifications it makes (e.g., decrementing stock) and the original modification in the Finance plugin (e.g., updating invoice status) are all part of the same unit of work. A single call to \_dbContext.SaveChangesAsync() at the end of the operation will commit all changes atomically, ensuring data consistency across the entire system.

## Data Persistence Architecture

This section provides a detailed design for the database schema and the data access layer, leveraging the modern capabilities of Entity Framework Core 9 to create an efficient, maintainable, and well-modeled persistence strategy.

### 4.1 Database Strategy: Unified Relational Database

The persistence strategy for the ERP system will be a **single, unified relational database**, such as SQL Server or PostgreSQL. This approach is the most common and effective for a monolithic application that requires strong transactional consistency across different business domains, which is a hallmark of ERP systems.1

The rationale is straightforward: a single database dramatically simplifies data integrity, reporting, and operational management. Business rules that span modules (e.g., "an order cannot be shipped if its corresponding invoice is not paid") can be enforced with simple database constraints or within a single transaction. Complex reporting queries that join data from finance, inventory, and sales can be written efficiently. Backup and restore operations are atomic and uncomplicated. The alternative—distributing data across multiple databases (one per module)—would introduce significant challenges like two-phase commits, eventual consistency models, and distributed query complexity, none of which are warranted for this system's scope.

### 4.2 Entity Framework Core 9 Integration

The data access layer will be built using Entity Framework Core 9, a modern object-database mapper (ORM) for.NET.13 The design will specifically leverage new and enhanced features in the EF Core 9 release to build a richer and more performant data layer.

* **Complex Types:** EF Core 9 features enhanced support for complex types, which allows mapping.NET classes or records as value objects embedded within an entity, rather than as separate tables.14 This is a powerful domain modeling tool. For instance, an  
  Address object (composed of Street, City, State, and ZipCode) can be defined once as a record and reused as a property on both Customer and PurchaseOrder entities. EF Core will map these properties to Address\_Street, Address\_City, etc., columns in the respective tables. This improves code reuse and aligns the data model more closely with domain-driven design principles. Similarly, a Money type (Amount, Currency) can be used to ensure monetary values are always handled consistently.
* **Enhanced JSON Support:** Many ERP entities have semi-structured data, such as product specifications, user preferences, or flexible metadata fields. EF Core 9 provides robust support for mapping properties to JSON columns in the database.15 This allows for storing flexible data structures without cluttering the main table schema. Crucially, it also supports translating LINQ queries into database-specific JSON query functions. This means it is possible to write a C# query like  
  context.Products.Where(p => p.Specifications["Color"] == "Blue"), which EF Core will translate into an efficient, indexed query against the JSON data on the database server.
* **Batch Updates and Deletes:** For performance-critical bulk operations, the design will utilize the ExecuteUpdateAsync and ExecuteDeleteAsync methods. Traditional updates require loading entities into memory, modifying them, and then saving them, which is inefficient for thousands of records. These batch methods translate a LINQ query directly into a single SQL UPDATE or DELETE statement that runs on the database server.15 For example, applying a 10% discount to all products in a specific category can be accomplished with a single, highly efficient command, avoiding the overhead of change tracking and network round-trips.

### 4.3 Core and Shared Entity Models

A small, carefully selected set of entities will be defined in the ERP.SharedKernel project because they are fundamental to the entire system and are referenced across multiple domains. These include:

* User: Represents a system user, managed by ASP.NET Core Identity.
* Tenant: A foundational entity for enabling multi-tenancy, allowing the ERP to serve multiple organizations from a single deployment.
* AuditLog: A base entity or convention for tracking changes to other entities, providing a consistent audit trail across all modules.

All other entities are to be defined and owned exclusively by their respective plugins. The Product entity belongs to the Inventory plugin, and the Invoice entity belongs to the Finance plugin. This maintains the principle of strong encapsulation at the data model level.

The dynamic model-building approach, where plugins contribute their IEntityTypeConfiguration implementations to the central DbContext, establishes a "schema contract" that is as vital as the C# interface contracts. A change made within one plugin's data model can have significant, non-obvious impacts on other plugins. For example, if the Inventory team adds a new, non-nullable SerialNumber column to the Product table, any other plugin that queries the Product table (even for read-only reporting purposes) may fail if its queries are not updated to account for this new column.

This interdependency necessitates a robust governance strategy for database schema changes. The "schema contract" must be treated as a public API. Teams cannot arbitrarily alter the structure of their tables without considering the downstream impact. This leads to the critical operational requirement for comprehensive integration testing that specifically validates cross-module data interactions. Furthermore, EF Core database migrations must always be generated from the fully composed model (Host + all loaded plugins). The continuous integration process must ensure that a migration generated from an updated plugin does not break the functionality of other, unchanged plugins, preventing a subtle but severe class of runtime failures.

## Reference Module Implementation: The Finance Plugin

This section provides a concrete, end-to-end design for the Finance plugin. It serves as a canonical example and a practical template for the development of all other modules in the system, demonstrating how the architectural principles and contracts are applied in practice.

### 5.1 Module Responsibilities and Boundaries

The Finance plugin is the financial backbone of the ERP. Its responsibilities are strictly confined to managing the company's financial records and transactions. Based on standard ERP module definitions, this includes:

* **General Ledger:** Managing the Chart of Accounts and recording all financial transactions as journal entries.1
* **Accounts Receivable (AR):** Creating, sending, and tracking customer invoices, as well as recording payments received.11
* **Accounts Payable (AP):** Managing vendor bills and processing payments to suppliers.
* **Financial Reporting:** Generating core financial statements like the Balance Sheet and Profit & Loss statement.

### 5.2 Data Models (Entities)

The plugin defines its own set of EF Core entities to represent its domain. These entities will reside within the ERP.Plugin.Finance project.

* Account: Represents an account in the Chart of Accounts (e.g., Cash, Accounts Receivable, Sales Revenue). Properties include AccountNumber, Name, and Type (Asset, Liability, Equity, etc.).
* JournalEntry: Represents a balanced accounting transaction, consisting of multiple JournalEntryLine items that debit or credit various accounts.
* Invoice: Represents a bill sent to a customer. Properties include InvoiceNumber, IssueDate, DueDate, Status (e.g., Draft, Sent, Paid, Void), and a collection of InvoiceLine items. It will also feature a complex type property, BillingAddress, of type Address.
* Payment: Records a payment received from a customer, which is typically applied against one or more invoices.

### 5.3 Internal Services and Logic

The business logic is encapsulated within a set of internal services, which are registered with the dependency injection container.

* InvoiceService: Contains methods for core AR operations, such as CreateInvoiceAsync, SendInvoiceToCustomerAsync, and MarkAsPaidAsync. The MarkAsPaidAsync method is responsible for updating the invoice status and publishing the InvoicePaidEvent.
* LedgerService: Provides methods like PostTransactionAsync, which takes a business event (like a payment) and creates the corresponding balanced journal entry in the general ledger.

### 5.4 UI Components and API Endpoints

The Finance plugin contributes its own user interface and web APIs to the application.

* **UI (Blazor Components):** The plugin contains a folder of self-contained Razor components for its user-facing features. These are discovered and routed by the Host.
  + InvoicesListPage.razor: Displays a searchable, sortable grid of all customer invoices.
  + InvoiceDetailPage.razor: Shows the details of a single invoice and allows for actions like sending reminders or recording payments.
  + ChartOfAccountsPage.razor: Provides an interface for managing the Chart of Accounts.
* **API (ASP.NET Core Endpoints):** The plugin registers its own API endpoints for programmatic access or integration with external systems.16
  + POST /api/finance/invoices: Creates a new customer invoice from a JSON payload.
  + GET /api/finance/invoices/{id}: Retrieves the details of a specific invoice.

### 5.5 Contract Implementations

To integrate with the host, the Finance plugin provides concrete implementations of the core interfaces defined in the Shared Kernel.

* FinancePluginModule.cs: This class implements IPluginModule, IEndpointRegistrar, and IEntityTypeConfigurationProvider. Its Initialize method registers InvoiceService and LedgerService. Its MapEndpoints method defines the API routes. Its GetEntityTypeConfigurations method returns instances of InvoiceConfiguration, AccountConfiguration, etc.
* FinanceNavProvider.cs: This class implements INavigationProvider. Its GetNavItems() method returns a list of navigation links, such as "Finance," "Invoices," and "Chart of Accounts," which the host UI shell will render in the main sidebar.

## Reference Module Implementation: The Inventory Plugin

This section designs a second plugin, Inventory, to manage products and stock levels. More importantly, it provides a detailed workflow that demonstrates the event-driven, inter-plugin communication pattern in action, illustrating how a complex business process can be orchestrated across decoupled modules.

### 6.1 Module Responsibilities and Boundaries

The Inventory plugin is responsible for the physical (or digital) goods the company manages. Its core functions include:

* **Product Management:** Defining product catalogs, including SKUs, descriptions, pricing, and other attributes.12
* **Warehouse Management:** Tracking inventory across multiple physical locations or warehouses.
* **Stock Control:** Managing stock levels, including receiving new stock, dispatching goods for orders, and performing stock adjustments.11

### 6.2 Data Models (Entities)

The Inventory plugin defines its own entities to model its domain.

* Product: Represents an item that can be stocked and sold. Properties include SKU, Name, Description, and potentially a JSON column for flexible Specifications.
* Warehouse: Represents a physical location where stock is held.
* StockLevel: A linking entity that tracks the quantity of a specific Product at a specific Warehouse.
* StockTransaction: Records every movement of stock. Properties include ProductId, WarehouseId, QuantityChanged, and Type (e.g., GoodsReceived, OrderDispatched, Adjustment). This provides a complete audit trail of all inventory changes.

### 6.3 Demonstrating Inter-Plugin Communication: A Detailed Workflow

This scenario illustrates the end-to-end process of fulfilling a customer order after payment has been received, showcasing the collaboration between the Finance and Inventory plugins without any direct coupling.

**Scenario: Fulfilling a Paid Order**

1. **Origin (Finance Plugin):** A user in the finance department receives a payment from a customer. They open the corresponding invoice in the ERP's UI (served by InvoiceDetailPage.razor from the Finance plugin) and click a "Record Payment" button. This action calls the MarkAsPaidAsync method on the InvoiceService.
2. **State Change and Event Publication (Finance Plugin):** Inside the MarkAsPaidAsync method, the following occurs:
   * The service updates the Status property of the Invoice entity to "Paid."
   * It creates a Payment record and associates it with the invoice.
   * It constructs a new event object: new InvoicePaidEvent(invoice.Id, invoice.CustomerId, invoice.LineItems).
   * Crucially, after these changes are prepared but before they are committed, it adds this event to a list of events to be published upon successful save.
   * It calls \_dbContext.SaveChangesAsync(). This commits the changes to the Invoices and Payments tables to the database in a single transaction.
   * Upon successful completion of SaveChangesAsync, the service iterates through its list of pending events and calls \_eventPublisher.PublishAsync(invoicePaidEvent).
3. **Mediation (Host Application):** The IEventPublisher service, managed by the Host, receives the InvoicePaidEvent. It looks up its registry and finds that a handler class within the Inventory plugin, ShipOrderHandler, is subscribed to this event type. The mediator then invokes the Handle method on an instance of ShipOrderHandler, passing the invoicePaidEvent object as an argument.
4. **Event Consumption and Action (Inventory Plugin):** The ShipOrderHandler.Handle method in the Inventory plugin is now executed.
   * It receives the event payload, which contains the list of line items from the paid invoice.
   * It iterates through the line items. For each item, it calls an internal StockService.DispatchProductAsync method.
   * The StockService performs the core inventory logic: it validates that sufficient stock is available, creates a new StockTransaction record with the type "OrderDispatched," and decrements the Quantity in the corresponding StockLevel record.
   * All these database operations are tracked by the same DbContext instance that was used by the Finance plugin.
5. **Transactional Completion (Shared Context):** After the event handler completes its logic, the control returns to the ASP.NET Core middleware pipeline. The DbContext's SaveChangesAsync() is called (either explicitly by the handler or at the end of the request), committing the inventory changes (new StockTransaction, updated StockLevel) to the database.

**Result:** The entire business process—from marking an invoice as paid to updating stock levels—is completed. It flows across two distinct, decoupled modules orchestrated by an asynchronous event. Because both modules operate on the same DbContext instance within the same request scope, the entire operation is transactionally consistent. If the stock update were to fail for any reason (e.g., insufficient inventory), the entire transaction would be rolled back, and the invoice would not be marked as paid, ensuring data integrity across the system.3

## Build, Deployment, and Operations

This final section addresses the practical, real-world considerations of building, deploying, and operating the modular ERP application. The architectural choices made earlier directly inform these operational strategies.

### 7.1 Build and Packaging Strategy

A disciplined build and packaging strategy is essential to enforce the architectural principle of shared dependency management, particularly concerning the ERP.SharedKernel library.

The Continuous Integration (CI) pipeline will be structured as follows:

1. **Shared Kernel Build:** The ERP.SharedKernel project is built, versioned (using Semantic Versioning), and published as a NuGet package to a private package repository (e.g., Azure Artifacts, GitHub Packages). This is the canonical source of the system's contracts.
2. **Host Application Build:** The ERP.Host project is built. It references the specific version of the ERP.SharedKernel NuGet package. Its published output contains the host executable and all its dependencies, including ERP.SharedKernel.dll.
3. **Plugin Build:** Each plugin project (e.g., ERP.Plugin.Finance) is built independently. It also references the exact same version of the ERP.SharedKernel NuGet package. The project file for the plugin must contain the critical metadata to prevent the shared assembly from being copied into its output directory:  
   XML  
   <ItemGroup>  
    <PackageReference Include="ERP.SharedKernel" Version="1.0.0">  
    <Private>false</Private>  
    <ExcludeAssets>runtime</ExcludeAssets>  
    </PackageReference>  
   </ItemGroup>  
     
   This configuration instructs the build process to use the ERP.SharedKernel assembly for compilation but to *not* include it in the final output folder. This ensures the plugin expects the host environment to provide this dependency at runtime.4
4. **Packaging:** The build output of each plugin is a "clean" package (e.g., a .zip file or a NuGet package with a custom type). This package contains only the plugin's own assembly (e.g., ERP.Plugin.Finance.dll), its .deps.json file, and its private, third-party dependencies. It explicitly does not contain ERP.SharedKernel.dll.

### 7.2 Deployment Model

The recommended deployment model is a **containerized application using Docker**. This approach promotes consistency, reliability, and immutable infrastructure principles.

The deployment process, managed by a Continuous Deployment (CD) pipeline, involves building a single Docker image that contains the host and all its plugins:

1. **Base Image:** The Dockerfile will start from the official ASP.NET Core 9 runtime base image.
2. **Copy Host:** The published output of the ERP.Host application is copied into the image's application directory (e.g., /app).
3. **Create Plugins Directory:** A dedicated directory for plugins is created inside the image (e.g., /app/plugins).
4. **Copy Plugins:** The contents of each plugin package (from step 4 above) are unzipped and copied into the /app/plugins directory.

The resulting Docker image is a self-contained, runnable instance of the entire ERP application. To deploy an update, whether it's a change to the host or an update to a single plugin, a new image is built and deployed, replacing the old one. This "immutable infrastructure" approach is significantly simpler and more reliable than attempting to hot-swap plugin DLLs in a running process, which can lead to complex state management issues and application instability.17

### 7.3 Configuration Management

The application's configuration will be managed using the standard ASP.NET Core configuration system, which aggregates settings from multiple sources, with environment variables typically overriding appsettings.json in production environments.

To maintain modularity, the appsettings.json file will be structured with dedicated sections for each plugin:

JSON

{  
 "ConnectionStrings": {  
 "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=ErpDb;Trusted\_Connection=True;"  
 },  
 "Logging": {  
 "LogLevel": {  
 "Default": "Information",  
 "Microsoft.AspNetCore": "Warning"  
 }  
 },  
 "FinancePlugin": {  
 "DefaultCurrency": "USD",  
 "InvoiceReminderDays": 15  
 },  
 "InventoryPlugin": {  
 "DefaultWarehouseId": "WH-CENTRAL-1",  
 "LowStockThreshold": 10  
 }  
}

Plugins will not access the global IConfiguration object directly. Instead, they will define their own strongly-typed configuration options classes (e.g., FinanceOptions) and register them with the dependency injection container. They can then inject IOptions<FinanceOptions> into their services. This decouples the plugin from the specific configuration source and makes its configuration dependencies explicit and testable. The Host application is responsible for binding the configuration sections from appsettings.json to these options classes during startup.

## Conclusion

This Technical Design Specification outlines a robust and modern architecture for a modular ERP system built on the.NET 9 platform. The chosen **Modular Monolith** pattern, realized through a dynamic plugin system, provides a pragmatic and powerful solution that balances development agility with operational simplicity. It allows for the decomposition of a complex domain into discrete, manageable modules, enabling autonomous teams to build and evolve features independently.

The success of this architecture hinges on disciplined adherence to its core principles:

* **Strong Encapsulation:** Plugins must remain isolated, communicating only through indirect, contract-based mechanisms.
* **Explicit Contracts:** The Shared Kernel must be treated as a stable, versioned API, defining the rules of engagement for all modules.
* **Event-Driven Communication:** An in-process event bus is the primary mechanism for orchestrating cross-module business processes, ensuring loose coupling and high extensibility.

The technical implementation, centered around the AssemblyLoadContext, provides true dependency isolation but introduces critical nuances in the build and deployment pipeline. The strategy of excluding shared contracts from plugin packages is not merely a suggestion but a mandatory requirement for the system to function correctly.

By leveraging the latest features of.NET 9, ASP.NET Core, Blazor, and Entity Framework Core, this design provides a forward-looking foundation for building a high-performance, scalable, and maintainable enterprise application. The provided reference implementations for the Finance and Inventory plugins serve as a concrete blueprint, demonstrating not only the structure of an individual module but also the elegant, decoupled manner in which they collaborate to execute complex, end-to-end business workflows. This specification provides a comprehensive and actionable plan for developing an ERP system that is architecturally sound and poised for future growth.

#### Works cited

1. 10 Top ERP Modules and Their Features - Oracle, accessed September 2, 2025, <https://www.oracle.com/erp/erp-modules/>
2. Understanding ERP Modules - The Top ERP Systems, accessed September 2, 2025, <https://www.top10erp.org/blog/erp-modules>
3. c# - Plugin-like architecture in .NET - Stack Overflow, accessed September 2, 2025, <https://stackoverflow.com/questions/2791859/plugin-like-architecture-in-net>
4. Create a .NET Core application with plugins - .NET | Microsoft Learn, accessed September 2, 2025, <https://learn.microsoft.com/en-us/dotnet/core/tutorials/creating-app-with-plugin-support>
5. Plugin Architecture In C# For Improved Software Design, accessed September 2, 2025, <https://newsletter.csharpinsights.com/p/plugin-architecture-c-improved-software-design>
6. ASP.NET Core Blazor - Microsoft Learn, accessed September 2, 2025, <https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-9.0>
7. Is .NET 9 beneficial for Blazor? - DEV Community, accessed September 2, 2025, <https://dev.to/alikolahdoozan/is-net-9-beneficial-for-blazor-37h6>
8. ASP.NET Core Blazor 9.0 New Features Summary | ABP.IO, accessed September 2, 2025, <https://abp.io/community/articles/asp.net-core-blazor-9.0-new-features-summary--x0fovych>
9. .Net core and plugins - Stack Overflow, accessed September 2, 2025, <https://stackoverflow.com/questions/60209723/net-core-and-plugins>
10. Building Scalable Enterprise Apps with Blazor and .NET 9 - Devessence, accessed September 2, 2025, <https://devessence.com/blog/!/57/building-scalable-enterprise-apps-with-blazor-and-net-9>
11. ERP Modules and Integrations: Our Complete Field Guide - Stampli, accessed September 2, 2025, <https://www.stampli.com/blog/accounting/erp-modules-integrations/>
12. ERP Modules: Types, Features & Functions - NetSuite, accessed September 2, 2025, <https://www.netsuite.com/portal/resource/articles/erp/erp-modules.shtml>
13. Microsoft.EntityFrameworkCore 9.0.8 - NuGet, accessed September 2, 2025, <https://www.nuget.org/packages/microsoft.entityframeworkcore>
14. 10 New Features in EF 9 - Telerik.com, accessed September 2, 2025, <https://www.telerik.com/blogs/10-new-features-ef-9>
15. Entity Framework Core 9: New Features and Performance Boosts You Need to Know, accessed September 2, 2025, <https://blog.ajithjoseph.com/blog/bbcfc2a6-59f7-47a1-9419-fc43ac690698/entity-framework-core-9-new-features-and-performance-boosts-you-need-to-know>
16. Tutorial: Create a controller-based web API with ASP.NET Core | Microsoft Learn, accessed September 2, 2025, <https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-9.0>
17. Does anyone bother with plugin architecture anymore in web apis to make easier to extend : r/dotnet - Reddit, accessed September 2, 2025, <https://www.reddit.com/r/dotnet/comments/1borpd9/does_anyone_bother_with_plugin_architecture/>
18. .NET application publishing overview - .NET | Microsoft Learn, accessed September 2, 2025, <https://learn.microsoft.com/en-us/dotnet/core/deploying/>