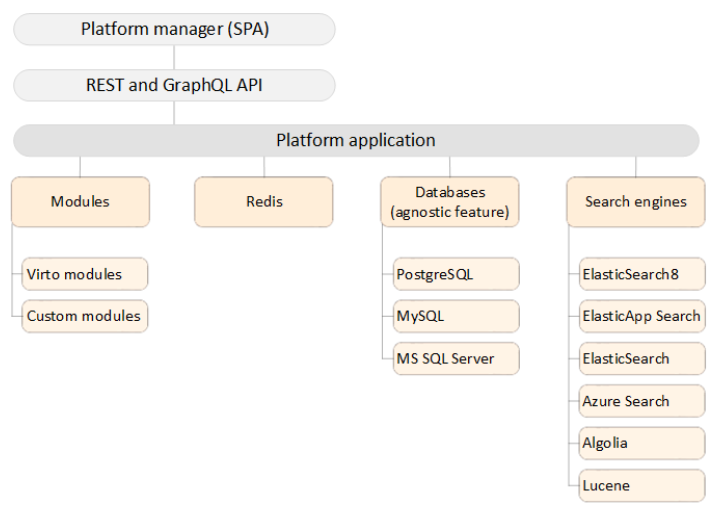
**Virto Commerce Back-End Architecture**

Virto Commerce is a B2B-focused, cloud-native e-commerce platform built on .NET Core. Its back-end architecture is designed for extensibility, scalability, and performance, making it suitable for complex enterprise scenarios. This document provides a detailed understanding of the Virto Commerce back-end architecture.

### Technology Stack

* + - C#
    - ASP.NET Core 8
    - EF Core 8
    - ASP.NET Core Identity
    - **OpenIddict**
    - **WebPack**
    - **SignalR Core**
    - **AngularJS**
    - **HangFire**
    - **StackExchange.Redis**
    - Serilog
    - **xUnit**
    - **MS SQL Server**
    - **MySQL**
    - **PostgreSQL**

### Conceptual Overview



### Folder Structure

* + src
    - **Module.Core**
      * + Events
        + Services
        + Notifications
        + Models
        + ModuleConstants.cs
    - **Module.Data**
      * + Caching
        + ExportImport
        + Handlers
        + Models
        + Repositories
        + Services

Crud

Search

* + - **Module.Data.MySQL**
      * + Migrations
        + DesignTimeDbContextFactory.cs
        + MySqlDataAssemblyMarker.cs
        + MySqlDbContextFactory.cs
    - **Module.Data.PostgreSQL**
      * + Migrations
        + DesignTimeDbContextFactory.cs
        + PostgreSqlDataAssemblyMarker.cs
        + PostgreSqlDbContextFactory.cs
    - **Module.Data.SQL**
      * + Migrations
        + DesignTimeDbContextFactory.cs
        + SqlServerDataAssemblyMarker.cs
        + SqlServerDbContextFactory.cs
    - **Module.Web**
      * + Content
        + Controllers

Api

* + - * + Localizations
        + Scripts
* Blades
* Resources
* module.js
  + - * + Module.cs
        + module.manifest
        + package.json
        + webpack.config.js
  + tests
* **Module.Test**
  + - * + Test.cs

### Core Principles and Design Patterns

* **Headless Architecture**: Virto Commerce adopts a headless approach, separating the front-end presentation layer from the back-end business logic. This allows for flexible front-end development using various technologies e.g., React, Angular, Vue.js, jQuery, JavaScript.
* **Microservices Architecture**: Virto Commerce is follows purely micro service approach which each and every module is independent from each other. We can plugin or plug out any module any time which has less impact on running application.
* **API-First Approach**: All functionalities are exposed through well-defined RESTful APIs, enabling seamless integration with other systems.
* **Event-Driven Architecture**: Asynchronous communication through events is used to decouple services and improve responsiveness.
* **Cloud-Native Design**: Virto Commerce is designed for cloud deployment, leveraging containerization (Docker) and orchestration (Kubernetes) for scalability and resilience.

### Key Architectural Components

* **Caching**:
  + - * Distributed Caching: Employs distributed caching solutions (e.g., Redis) to store frequently accessed data, reducing database load and improving performance.
* Cache Invalidation: Implements strategies for cache invalidation, ensuring data consistency and freshness.
  + - * Caching Layers: Caching is applied at various layers, including data access, application logic, and API responses.
      * Configuration: Caching behavior is configurable, allowing customization based on specific requirements.
* **Data Import**:
* Bulk Import: Enables efficient bulk import of large datasets from various sources (e.g., CSV, XML, and JSON).
* Data Mapping: Provides tools for mapping data from source files to Virto Commerce entities.
* Import Jobs: Supports scheduled import jobs for automated data updates.
* Error Handling: Implements robust error handling and logging mechanisms for data import processes.
* Data Validation: Includes data validation to ensure data integrity during import.
* **Persistence**:
* Data Storage: Virto Commerce utilizes a flexible data persistence layer, supporting various database systems (e.g., SQL Server, PostgreSQL, and MySQL).
* Entity Framework Core: Leverages Entity Framework Core for object-relational mapping (ORM), simplifying data access and manipulation.
* Repositories: Implements the repository pattern to abstract data access logic, promoting code reusability and testability.
* Data Migration: Provides mechanisms for managing database schema changes and data migrations.
* **Modularity**:
* Functionality is broken down into independent modules, promoting maintainability, reusability, and extensibility.
* API-First Approach: All functionalities are exposed through well-defined APIs, enabling seamless integration with other systems.
* Cloud-Native Design: Designed for cloud deployment, leveraging containerization and orchestration technologies.
* Scalability & Performance: Built to handle high traffic and large datasets, ensuring optimal performance.
* **Indexing and Search**:
* Elasticsearch: Integrates with Elasticsearch for powerful full-text search capabilities.
* Indexing: Provides mechanisms for indexing product catalogs, customer data, and other relevant information.
* Search Queries: Supports complex search queries, including faceted search, filtering, and sorting.
* Search Optimization: Allows for search optimization through relevance tuning, synonym management, and other techniques.
* Real time indexing: Supports near real time indexing of data changes.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **Lucene** | **Elasticsearch** | **Elastic App Search** | **Azure Cognitive Search** | **Algolia** |
| **Type** | Library | Search Engine | Managed Service | Managed Service | Managed Service |
| **Deployment** | Self-hosted | Self-hosted/Cloud | Elastic Cloud | Azure Cloud | Algolia Cloud |
| **Focus** | Core Search | Scalable Search & Analytics | Application Search | AI-Powered Search | Fast, Relevant Search |
| **Ease of Use** | Complex | Moderate | Simple | Moderate | Simple |
| **Scalability** | Requires Customization | Built-in | Managed | Managed | Managed |
| **AI/ML** | Requires Customization | Built-in/Plugins | Limited | Built-in | Limited |
| **Use Cases** | Custom Apps | Log Analysis, Enterprise Search | App Search, E-commerce | Enterprise Search, Knowledge Mining | E-commerce, Mobile Search |
| **Cost** | Free, development cost | Variable, based on resources | Based on usage, managed service | Based on usage, azure resources | Based on usage, tiered pricing |

* **Scalability**:
* Virto Commerce is a powerful e-commerce solution designed to handle projects of various sizes and complexities. Whether for B2B, B2C, or B2B2C enterprises, marketplaces, or SaaS commerce platforms, Virto Commerce satisfies diverse business needs.
* Its scalability options, ranging from Small to Extra Large configurations. It helps you choose the right Azure services and pricing plans for your specific project needs.
* Additionally, it explains how to scale solutions built on the Virto Commerce Platform and Frontend Application Kit.
* **Domain Event**:
* A domain event is something that happened in a particular domain, and something you want other parts of the same domain to be aware of and potentially react to the in-process principle.
* An important benefit of domain events is that any side effects of something happening in a domain can be expressed explicitly and not implicitly.
* We can create register an event handler and subscribe to a domain event.
* Also we can override existing event handler with new derived type.
* **Notification**:
* Event-Driven Architecture: Employs an event-driven architecture to trigger notifications based on specific events.
* Notification Channels: Supports various notification channels, including email, SMS, and push notifications.
* Notification Templates: Allows for the creation of customizable notification templates.
* Notification Queues: Uses message queues to ensure reliable delivery of notifications.
* Subscription Management: Provides mechanisms for managing notification subscriptions.
* **Dynamic Properties:**
* Custom Attributes: Allows for the creation of custom attributes and properties for entities, enabling flexible data modeling.
* Metadata Management: Provides tools for managing metadata, including attribute definitions, data types, and validation rules.
* Dynamic Forms: Enables the creation of dynamic forms based on custom attributes, allowing for flexible data entry and management.
* Extensible Data: Provides the ability to extend the data model without requiring code changes.
* **Security**:
  + - Uses industry standard protocols like OAuth2, and OpenID Connect.
    - Role based access control is implemented.
    - Encrypts sensitive data at rest and in transit.
    - Provides logging and auditing capabilities for security-related events.
* **Logging**:
* Structured Logging: Uses structured logging to capture detailed information about application events.
* Log Aggregation: Supports integration with log aggregation tools e.g., Serilog and Azure Application Insights for centralized log management.
* Log Levels: Allows for configurable log levels to control the verbosity of logging.
* Exception Handling: Implements robust exception handling and logging mechanisms.
* Performance Monitoring: Logs performance metrics to identify and address bottlenecks.

### Custom App development

* Allows businesses to create their own app ecosystem and improve overall efficiency and productivity.
* Ensures consistent user interface.
* Provides an improved user experience due to seamless navigation between apps and one login.
* Enables multiple teams to work simultaneously on different aspects of the project.
* Virto Commerce allow us to develop new custom app different ways e.g., Template, Source code from scratch, or CLI.

### Deployment and Scalability

* Virto Commerce can be deployed on various cloud platform e.g., Azure, Google Could, AWS or On-Premise too.
* Microservices architecture and containerization enable horizontal scaling, allowing the platform to handle increasing traffic and load.
* Kubernetes facilitates automated deployment, scaling, and management of containerized applications.
* Cloud native design allows for automated scaling of resources.

### Customization and Extensibility

* Unlimited extensibility is an indisputable advantage of Virto Commerce.
* End-user solution developers can adapt or replace Virto Commerce modules with their own functionality.
* Virto Commerce has three types of vendor-made extensions that guarantee 'seamless delivery'.
* **No-code extensions:** Virto Commerce supports declarative extensions through the admin interface and API to extend the data model, UI, and workflow processing.
* [Dynamic properties](https://docs.virtocommerce.org/platform/developer-guide/Fundamentals/Dynamic-Properties/overview/): Extend entities, such as Cart, Order, Company, etc., with new properties.
* Statuses: Enable implementing any workflow.
* Role-based security and permissions: Restrict access for a user (API) based on their role.
* **API-based extensions:** These are traditional cloud-based extensions based on API calls and events (hooks). With such extensions, you can use any language or framework for integration and reactive programming.
* REST: Enables access to module business functionality, i.e. all CRUD operations and business logic are accessible via Rest API.
* GraphQL: A new generation of Business API for client applications, it provides you with additional capabilities.
* We also support two types of event providers:
* Webhooks: A good point for no development integration.
* Eventbus: Read for reactive programming and enterprise integrations.
* **Native extensions:** Virto Commerce creates a unique extension framework based on .NET extension and dependency injection. It enables extending default implementation for the unique needs of each business.
* Admin UI
* API
* Business Logic
* Database

### Commands

* Module Creation
* Install the Virto Commerce module template

dotnet new install VirtoCommerce.Module.Template

* Create a new module based on that template

dotnet new vc-module-dba-xapi --ModuleName CustomerReviews --Author "Jon Doe" --CompanyName VirtoCommerce

* SQL

# SQL Server connection details

$ServerName = "localhost,1433" # Replace with your server and port if needed

$SqlUser = "sa"

$SqlPassword = "Admin@123" # Replace with your SA password

$DatabaseName = "VirtoCommercedocker"

# Build the connection string

$ConnectionString = "Server=$ServerName;Database=master;User ID=$SqlUser;Password=$SqlPassword"

# Create a SqlConnection object

try {

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection($ConnectionString)

$SqlConnection.Open()

# Create the database using T-SQL

$SqlCommand = New-Object System.Data.SqlClient.SqlCommand("Use Master; Drop Database [$DatabaseName], $SqlConnection)

$SqlCommand.ExecuteNonQuery()

Write-Host "Database '$DatabaseName' created successfully."

}

catch {

Write-Error $\_.Exception.Message

}

finally {

if ($SqlConnection -and $SqlConnection.State -eq "Open") {

$SqlConnection.Close()

}

}

------------------------------------------------------------------------------------------------------------------

# SQL Server connection details

$ServerName = "localhost,1433" # Replace with your server and port if needed

$SqlUser = "sa"

$SqlPassword = "Admin@123" # Replace with your SA password

$DatabaseName = "VirtoCommercedocker"

# Build the connection string

$ConnectionString = "Server=$ServerName;Database=master;User ID=$SqlUser;Password=$SqlPassword;"

# Create a SqlConnection object

try {

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection($ConnectionString)

$SqlConnection.Open()

# Create the database using T-SQL

$SqlCommand = New-Object System.Data.SqlClient.SqlCommand("CREATE TABLE [dbo].[Emp]([EmployeeID] [int] IDENTITY(1,1) NOT NULL, [EmployeeName] [nvarchar](50) NULL, [salary] [int] NULL,PRIMARY KEY CLUSTERED ( [EmployeeID] ASC)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON, OPTIMIZE\_FOR\_SEQUENTIAL\_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]", $SqlConnection)

$SqlCommand.ExecuteNonQuery()

Write-Host "Query executed successfully."

}

catch {

Write-Error $\_.Exception.Message

}

finally {

if ($SqlConnection -and $SqlConnection.State -eq "Open") {

$SqlConnection.Close()

}

}

------------------------------------------------------------------------------------------------------------------

# SQL Server connection details

$ServerName = "localhost,1433" # Replace with your server and port if needed

$SqlUser = "sa"

$SqlPassword = "Admin@123" # Replace with your SA password

$DatabaseName = "VirtoCommerce3docker"

$Query= "Insert into [dbo].[Emp] values('EmployeeName1',5000 ) ;Insert into [dbo].[Emp] values('EmployeeName2',5000 ) ;Insert into [dbo].[Emp] values('EmployeeName3',5000 ) ;Insert into [dbo].[Emp] values('EmployeeName4',5000 ) Select\* from [dbo].[Emp] "

# Build the connection string

$ConnectionString = "Server=$ServerName;Database=master;User ID=$SqlUser;Password=$SqlPassword;TrustServerCertificate=True;Encrypt=false;"

# Create a SqlConnection object

try {

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection($ConnectionString)

$SqlConnection.Open()

# Create the database using T-SQL

$SqlCommand = New-Object System.Data.SqlClient.SqlCommand($Query, $SqlConnection)

$Result = $SqlCommand.ExecuteNonQuery()

Write-Host $Result

#Write-Host "Database '$DatabaseName' created successfully."

}

catch {

Write-Error $\_.Exception.Message

}

finally {

if ($SqlConnection -and $SqlConnection.State -eq "Open") {

$SqlConnection.Close()

}

}

------------------------------------------------------------------------------------------------------------------

# SQL Server connection details

$ServerName = "localhost,1433" # Replace with your server and port if needed

$SqlUser = "sa"

$SqlPassword = "Admin@123" # Replace with your SA password

$DatabaseName = "VirtoCommerce3docker"

$Query= "IF EXISTS(SELECT 1 FROM sys.databases WHERE name = '$DatabaseName') SELECT 1 ELSE SELECT 0;"

# Build the connection string

$ConnectionString = "Server=$ServerName;Database=master;User ID=$SqlUser;Password=$SqlPassword;TrustServerCertificate=True;Encrypt=false;"

# Create a SqlConnection object

try {

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection($ConnectionString)

$SqlConnection.Open()

# Create the database using T-SQL

$SqlCommand = New-Object System.Data.SqlClient.SqlCommand($Query, $SqlConnection)

$Result = $SqlCommand.ExecuteNonQuery()

Write-Host $Result

#Write-Host "Database '$DatabaseName' created successfully."

}

catch {

Write-Error $\_.Exception.Message

}

finally {

if ($SqlConnection -and $SqlConnection.State -eq "Open") {

$SqlConnection.Close()

}

}

--------------------------------------------------------------------------------------------------------------

# Requires the Docker module for PowerShell: Install-Module DockerMsftProvider -Force; Install-Module Docker -Force

# Configuration

$ContainerName = "localhost,1433" # Replace with your SQL Server container name

$DatabaseName = "VirtoCommerce3docker" # Replace with the database name you want to check

$SqlUser = "sa" # Replace with your SQL Server user

$SqlPassword = "Admin@123" # Replace with your SQL Server password

try {

# Check if the container exists and is running

$container = docker ps --filter "name=$ContainerName" --format "{{.ID}}" | Out-String | Trim

if (-not $container) {

Write-Error "Container '$ContainerName' not found or not running."

return

}

# Execute SQL query inside the Docker container

$sqlCommand = "sqlcmd -S localhost -U $SqlUser -P '$SqlPassword' -Q IF EXISTS(SELECT 1 FROM sys.databases WHERE name = '$DatabaseName') SELECT 1 ELSE SELECT 0;"

$result = docker exec $ContainerName $sqlCommand | Out-String | ConvertFrom-String

# Check the result

if ($result -eq 1) {

Write-Host "Database '$DatabaseName' exists in container '$ContainerName'."

} else {

Write-Host "Database '$DatabaseName' does not exist in container '$ContainerName'."

}

}

catch {

Write-Error "An error occurred: $($\_.Exception.Message)"

}