Common Code Architecture (CCA) Knowledge Base

Overview

This knowledge base contains information about the Common Code Architecture (CCA) initiative at FNZ. The CCA aims to restructure the platform codebase to improve maintainability, testability, and build performance by consolidating over 300 inconsistently written projects into a unified, structured architecture.

Table of Contents

Summary & Rationale

Current Problems

The existing codebase suffers from:

- **Split and repeated code** across platforms

- **Over 300 projects** written inconsistently

- **Inability to understand** system behaviour

- **Inability to unit test** code effectively

- **Slow build times**

Initiative Goals

1. **One set of projects for core code** (`Fnz.Platform.*` / `Fnz.Framework.*`) - Moving away from monolithic `Fnz.Platform` in favour of smaller, specialised libraries

2. **Consistent platform-specific code** (`.Platform.*` e.g., `StandardLife.Platform.*`, `Axa.Platform.*`, `Jpm.Platform.*`) to enable easy migration into `Fnz.Platform`

3. **Consistent code structure** across all projects (separating data access from business logic and UI code)

4. **Database abstraction** to enable easy refactoring of table structures

Support & Questions

- **General questions**: Email #Common Code Design or post at [Viva Engage - Answers](yammer.com) tagged with "cca"

- **Specific areas**: See Common Code Architecture Contact List

- **Detailed features**: Refer to Fnz.Platform Detailed Guide

---

Project Structure

Where Should My Code Go?

Single.Platform / Single.Platform.Api

- New code written in CCA style

Single.Platform.Adapters

- New CCA-style code that needs to interact with legacy code in `Single.Platform.Legacy`

Single.Platform.Legacy.sln

- Modifications to legacy code only

- **No new projects allowed**

Fnz.Platform / Fnz.Platform.Api (Deprecated)

- Use `Single.Platform` and `Single.Platform.Api` instead

- OR add functionality to equivalent common components (e.g., `Fnz.One.Charges`) after consulting with that team

- Previously used for CCA-style code shared with platforms not using Single.Platform codebase (Close, JPM, AXA, etc.)

- Lives in Accurev stream: Core Depot, `Fnz.Core.Platform_Master` stream

- Team City release build publishes versioned NuGet packages

.Platform

- Lives in individual platform streams

- Used for project-specific code

- When a feature is used by multiple platforms, it moves up to `Fnz.Platform`

Project Architecture Principles

Api Projects Pattern

- Each project has both a **concrete project** and an **Api project**

- Example: `Single.Platform.csproj` & `Single.Platform.Api.csproj`

- **Api projects** define interfaces

- Concrete implementation can be chosen via configuration (factory or dependency injection)

- **All communication** with common code architecture must go via Api project interfaces

Standard Project Folders

The following projects follow the same first and second-level folder structure:

- `Fnz.Platform` / `Single.Platform`

- `Fnz.Platform.Api` / `Single.Platform.Api`

- `Fnz.Platform.Legacy` / `Single.Platform.Legacy`

- `Fnz.Platform.Legacy.Vb` / `Single.Platform.Legacy.Vb`

- `.Platform`

- `.Platform.Api`

- `.Platform.Legacy`

- `.Platform.Legacy.Vb`

Note: Both `Single.Platform.xxx` and `Fnz.Platform.xxx` folders exist due to legacy reasons. Originally, `Fnz.Platform` was in a different repository meant to be shared among all Fnz.One instances, but these were eventually forked per client, negating the need for separate repos.

■■ **Important**: Don't add new projects or first-level folders without checking with #Common Code Design team first.

Folder Structure Hierarchy

Accounts

- AccountHolders
- Accounts
- Addresses
- BankAccounts
- BulkUpload
- Customers (now at top level)
- Linked
- SubAccounts
- Trusts

Banking

- GeneralLedger

Charges

- Accruals
- AdHoc
- PaymentsOut
- Regular
- Schedules

Communication

- AlertingPreferences
- Contacts
- IssueLog

- Notifications

- Reminders

CorporateActions

- Elections

Counterparty

- Accounts

- Transactions

Customers

(Top-level folder)

Documents

- AccountDocuments

- StaticDocuments

Hierarchy

- Ebc

- Employers

- Schemes

MarketData

- EventDiary

- Products

- Series

ModelPortfolios

Orders

- Batches

- Bulk

- Corrections

- Disinvestment

- Drawdown

- Single

Payments

- Deposits

- Payroll

- Withdrawals

Portfolio

- Performance

- Projections

- Transactions

- Valuations

ProductWrappers

Regulations

- Tax

Trading

- Settlement

- Transactions

Transfers

Workflow

Users

Testing Projects

- **Single.Platform.Unit.Test** - Unit tests

- **Single.Platform.Integration.Test** - Integration tests

Coding and Naming Conventions

General Rules

Most conventions are covered by StyleCop/NDepend rules. Additional conventions are listed below.

Critical Rules

Dependencies

- ■ **Never import the CCore02 dll** into a project

- ■ Use the new DAL (`Fnz.Framework.Components.Util.dll`) instead

- Note: An NDepend rule will be added for this

Entity/DTO Mapping

- ■ Don't create separate classes for mapping API entity/DTO to internal objects

- ■ Use a single class (e.g., just have `Fnz.Platform.Api.User` class)

- ■ Don't also create a `User` class in the `Fnz.Platform` project

Naming Conventions

Fields and Properties

- **Fields**: Must begin with `_` (underscore) character

- Example: `_userId`, `_customerName`

- **Properties**: Must begin with an upper-case character

- Example: `UserId`, `CustomerName`

Writer Methods

- **Updating objects**: Use `Save` (not 'Create' or 'Update')

- Example: `UserWriter.Save(User user)`

- Behavior: Creates if `id == 0`, otherwise updates

- **Adding references**: Use `AddXXX` or `AssignXXX`

- Example: `UserWriter.AssignUserToRole`

Reader Methods

- **Obtaining objects**: Use `GetXXByYYY` pattern

- Example: `UserReader.GetUserById`

- **Query-based retrieval**: Use `GetByQuery`

- Example: `UserReader.GetByQuery(UserQuery)`

Handler Classes

- **Query Handlers**:

- Suffix: `QueryHandler`

- ■ Don't use 'Get' prefix

- ■ Correct: `CustomersQueryHandler`

- ■ Incorrect: `GetCustomersQueryHandler`

- **Command Handlers**:

- Suffix: `CommandHandler`

- Example: `CreateUserCommandHandler`

Project Organization Rules

Location of Components

Component	Location
Commands	`Fnz.Platform.Api` project
Entities (passed to/from commands)	`Fnz.Platform.Api` project
Query Handlers	`Fnz.Platform` project
Command Handlers	`Fnz.Platform` project
Readers/Writers	`Fnz.Platform` project

Folder Structure for Handlers

- **Query Handler classes**:

- Folder: `Queries` subfolder

- Parent: `QueryHandlers` folder

- **Command Handler classes**:

- Folder: `Commands` subfolder

- Parent: `CommandHandlers` folder

- **Validators**:

- Folder: `Validators` subfolder

- Parent: `CommandHandlers` folder

Implementation Standards

Command Handler Interfaces

- CommandHandlers must implement one or more `IXXXCommandHandler` interfaces

- ■ Don't just implement `ICommandHandler`

- ■ Implement specific interfaces for better VS/ReSharper navigation

Documentation Requirements

- **Public API** (Platform.API Commands/Queries and their interfaces) must be documented with XML comments

- Developers should not need to debug into code to understand what it does

---

Getting Started

Introduction

The Common Code Architecture (CCA) provides a structured approach to organizing platform code. This guide will help you understand where to place your code and how to follow the established conventions.

Prerequisites

- Access to the appropriate Accurev stream

- Understanding of the Api project pattern

- Familiarity with CQRS principles (Commands and Queries)

- StyleCop/NDepend configured in your development environment

Quick Start

1. Determine if your code is new CCA-style or legacy modification

2. Choose the appropriate project location (Single.Platform vs Single.Platform.Legacy)

3. Follow the folder structure hierarchy for your domain area

4. Apply naming conventions and create necessary Api interfaces

5. Write unit tests in Single.Platform.Unit.Test

Core Concepts

Internal Project Structure

Each leaf folder should contain the following subfolders with corresponding classes:

Single.Platform Structure

Single.Platform/

■■■ Validators/

■■■ CommandHandlers/

■■■ QueryHandlers/

■■■ DataAccess/

■ ■■■ Readers/

■ ■■■ Writers/

■ ■■■ Mappers/

■■■ Setup/ (for Castle installers)

Single.Platform.Api Structure

Single.Platform.Api/

■■■ Commands/

■■■ Queries/

■■■ Entities/

■■■ Events/

Important: The subfolders for the Api folder are **always necessary** even if there are only a few classes. This enforces consistency for API users.

See "Folder Structure and Naming Conventions Examples" for detailed examples.

Legacy Tests Namespaces and Folders

Namespace Convention

Test namespace = `XYZ.Platform.Legacy.Integration.Test` + ``

(Same pattern applies for unit tests project)

Folder Location

Test folder location must correspond with the namespace.

Example:

- **Class under test**: `cTasks02.BankTransactions.Hbos.HbosBankTransactionLoader`

- **Test class namespace**: `Single.Platform.Legacy.Integration.Test.cTasks02.BankTransactions.Hbos.HbosBankTransactionLoader`

Command Classes

Definition: Classes that represent a command to change the state of the system. They contain no business logic and purely capture the data required to perform a command.

Example:

namespace Fnz.Provider.Platform.Accounts.Api.IssueLog.Command

{

```
public class CreateIssueCommand : ICreateIssueCommand
{
    public int Id { get; set; }
    public IssueType Type { get; set; }
    public string Description { get; set; }
}
```

}

CommandHandler Classes

Definition: Classes that execute commands by coordinating various objects to perform authorization, validation, and database operations.

Responsibilities:

- Authorization

- Validation

- Reading/Writing objects to/from the database

Example:

```
public interface ICreateIssueCommandHandler : ICommandHandler

{

}

internal class CreateIssueCommandHandler : ICreateIssueCommandHandler

{

            private readonly IIssueWriter _issueWriter;
            private readonly IUserContextProvider _userContextProvider;

            public CreateIssueCommandHandler(IIssueWriter issueWriter, IUserContextProvider userContextProvi
            {
                _issueWriter = issueWriter;
                _userContextProvider = userContextProvider;
            }

            public void Execute(ICreateIssueCommand command)
            {
                var issue = CreateIssue(command);
                _issueWriter.Save(issue);
                command.Id = issue.Id;
            }

            private Issue CreateIssue(ICreateIssueCommand command)
            {
                return new Issue
                {
                    Description = command.Description,
                    Type = command.Type,
                    CreatedByUserId = _userContextProvider.GetUserContext().UserId,
                };
            }
```

}

Best Practice: Create, Update, and Delete commands for the same entity (or aggregate root) should be implemented in **one command handler** where possible to prevent an explosion of handler classes.

Example:

IssueCommandHandler

```
            : ICommandHandler<CreateIssueCommand>,
              ICommandHandler<UpdateIssueCommand>,
              ICommandHandler<DeleteIssueCommand>
```

Query Classes

Definition: Classes that represent criteria to query the system. They contain no business logic and purely capture the data required to perform a query.

Example:

public class IssueQuery

{

```
            public string Id { get; set; }
            public int? StatusId { get; set; }
            public int? TypeId { get; set; }
            public string Owner { get; set; }
            public Priority Priority { get; set; }
            public string LoggedBy { get; set; }
            public string Description { get; set; }
            public string Company { get; set; }
            public int? CategoryId { get; set; }
            public int? WorkstreamId { get; set; }
            public int? ComponentId { get; set; }
            public int? ProjectId { get; set; }
            public string Account { get; set; }
            public DateTime? LoggedAfter { get; set; }
            public DateTime? LoggedBefore { get; set; }
```

}

QueryHandler Classes

Definition: Classes that execute queries by coordinating various objects to perform authorization and read operations from the database.

Responsibilities:

- Authorization

- Reading objects from the database

Example:

```
public interface IIssueQueryHandler : IQueryHandler

{

}


internal class IssueQueryHandler : IIssueQueryHandler

{
            private readonly IIssueReader _reader;

            public IssueQueryHandler(IIssueReader reader)
            {
                _reader = reader;
            }

            public Issue Execute(IssueQuery query)
            {
                return _reader.GetByQuery(query);
            }
}
```

Validator Classes

Definition: Classes that perform validation of commands to ensure they're valid. Validation is done in these classes rather than baked into the CommandHandler.

Key Points:

- CommandHandlers call validators using `ValidationInterceptor`

- Uses Castle AOP Interceptors to ensure validation is called before command handler execution

FluentValidator

A custom build of the third-party tool **FluentValidation** has been integrated with `Fnz.Platform` (and `Fnz.Framework 5`) to simplify writing command validation rules.

Key Features:

- Validators inherit from `FluentValidator`

- Automatically registered with the appropriate CommandHandler (of type T)

- Automatically called via Castle AOP before command handler executes

Example:

```
// Automatically registered to be called before ProcessSamlLogoutRequestCommandHandler

public class SamlLogoutRequestCommandValidator : FluentValidator

{

            public SamlLogoutRequestCommandValidator()
            {
                // Note: None of these validations should have a dependency on query/command handlers
                // If they do, it breaks the Castle registration
                RuleFor(x => x.SamlConfigurationKey)
                    .Must(x => x.IsNullOrEmpty() == false)
                    .WithMessage("SAML configuration must be specified");
                RuleFor(x => x.Request)
                    .Must(x => x.IsNullOrEmpty() == false)
                    .WithMessage("SAML request must be included");
            }

}
```

Important: Validations should **not** have dependencies on query/command handlers, as this breaks Castle registration.

Additional Notes:

- Validator classes should not depend on query/command handlers - it breaks Castle registration

- This also aligns with having Command/Query handlers as the 'entry points' to the system

- `[ValidationRequired]` might be required on command handler implementation for validation to be registered

- See Knowledge Transfer for more examples and information on FluentValidator

Legacy Validator Example (Pre-FluentValidator)

This is how validation was done before FluentValidator integration:

```
public class SetCustomerAddressCommandValidator : Validator

{

        /// <summary>Validation for address</summary>
        /// <exception cref="Validate">AddressLine1Required</exception>
        /// <exception cref="Validate">AddressLine2Required</exception>
        /// <exception cref="Validate">AddressPostCodeRequired</exception>
        public override ValidationResult Validate(SetCustomerAddressCommand command)
        {
            var validator = new CustomerAddressCommandValidator();
            var validatorErrors = validator.Validate(command);
            var result = new ValidationResult();
            result.AddError(validatorErrors.Errors.Select(x =>
                new ValidationError(new ErrorCode(x.ErrorCode.HasValue ? x.ErrorCode.Value : 1, x.ErrorM
            return result;
        }

}
```

```
public class CustomerAddressCommandValidator : AbstractValidator

{

        private readonly CountryCode[] _supportedDomiciles = {
            CountryCode.HKG, CountryCode.AUS, CountryCode.GBR,
            CountryCode.CHN, CountryCode.NZL, CountryCode.SGP
        };

        public CustomerAddressCommandValidator()
        {
            RuleFor(c => c.CustomerAddress.Type)
                .Must(x => x == AddressType.Postal || x == AddressType.Residential)
```

```
                    .WithErrorCodeAndMessage(ErrorCodes.CustomerAddressTypeRequired);
                RuleFor(c => c.CustomerAddress.Address.CountryCode)
                    .NotEmpty()
                    .WithErrorCodeAndMessage(ErrorCodes.AddressCountryCodeRequired);
                RuleFor(c => c.CustomerId)
                    .NotEmpty()
                    .WithErrorCodeAndMessage(ErrorCodes.CustomerIdRequired);
                RuleFor(c => c.CustomerAddress.Address)
                    .SetValidator(new GenericAddressValidator())
                    .Unless(c => _supportedDomiciles.Contains(c.CustomerAddress.Address.CountryCode),
                        ApplyConditionTo.CurrentValidator)
                    .SetValidator(new AustraliaAddressValidator())
                    .When(c => c.CustomerAddress.Address.CountryCode == CountryCode.AUS,
                        ApplyConditionTo.CurrentValidator)
                    .SetValidator(new ChinaAddressValidator())
                    .When(c => c.CustomerAddress.Address.CountryCode == CountryCode.CHN ||
                        c.CustomerAddress.Address.CountryCode == CountryCode.HKG,
                        ApplyConditionTo.CurrentValidator);
            }
}
```

Authorization

Similar to validation, authorization is provided via an interceptor.

Work in Progress (still to implement):

- Have an `AuthorisationInterceptor` that calls `ICheckAuthorisation` classes

- **Performance consideration**: In some cases, it's less performant to check access in a decorator/interceptor

Example: For bank accounts, you need to retrieve the account anyway to check access:

1. Retrieve bank account

2. Get `claccountid`

3. Check user has access to `claccountid`

- If true: return bank account

- If false: throw authorization exception

Reader Classes

Definition: Classes responsible for reading objects from the database.

Example:

```
internal interface IIssueReader

{

            Issue GetById(int issueId);

}



internal class IssueReader : DataAccessBase, IIssueReader

{

            private readonly ICommentReader _commentReader;
            private readonly IEventReader _eventReader;
            public IssueReader(IDataAccess dataAccess, ICommentReader commentReader, IEventReader eventReade
                : base(dataAccess)
            {
                _commentReader = commentReader;
                _eventReader = eventReader;
            }
            public Issue GetById(int issueId)
            {
                var fields = new Params
                {
                    HelpDeskLogTable.Columns.id,
                    HelpDeskLogTable.Columns.Description,
                    HelpDeskLogTable.Columns.Type,
                };
                var search = new Params
                {
                    { HelpDeskLogTable.Columns.id, SqlOperatorString.Equal, issueId }
                };
                // QueryFactory is a property in DataAccessBase created using IDataAccess
                // passed into the constructor
                Issue issue = QueryFactory
                    .Select(fields)
                    .From<HelpDeskLogTable>()
                    .Where(search)
                    .ExecuteAndReturnFirst(new IssueMapper());
                    // Mappers should be instantiated where you need them,
                    // not in the constructor (we don't use them every time)
                if (issue != null)
                {
                    issue.Comments = _commentReader.GetByIssueId(issueId);
                    issue.History = _eventReader.GetByIssueId(issueId);
                }
                return issue;
            }

}
```

Key Point: Mappers should be instantiated where needed, not in the constructor, because they aren't used every time the reader is used.

Writer Classes

Definition: Classes responsible for writing objects to the database.

Example:

internal interface IIssueWriter

{

```
            void Save(Issue issue);
```

}

internal class IssueWriter : DataAccessBase, IIssueWriter

{

```
            public IssueWriter(IDataAccess dataAccess) : base(dataAccess)
            {
                _dataAccess = dataAccess;
            }
            // Mappers should not be initialised in constructor - instantiate where needed
            public void Save(Issue issue)
            {
                var rs = new IssueMapper().CreateRecordset(issue);
                if (issue.Id == 0)
                {
                    rs.RowStatus = Recordset.RecordStatus.Added;
                }
                else
                {
                    rs.RowStatus = Recordset.RecordStatus.Modified;
                }
                _dataAccess.SaveRecordset<HelpDeskLogTable>(rs, true);
                issue.Id = rs.GetInteger(HelpDeskLogTable.Columns.Id);
                // Save any dependent objects too - in this case the IssueLogHistory table
                var commentRecords = new CommentMapper().CreateRecordset(issue.Comments.ToList());
                commentRecords.RowStatus = Recordset.RecordStatus.Added;
                _dataAccess.SaveRecordset<IssueLogHistoryTable>(commentRecords, false);
            }
```

}

Key Point: Mappers should not be initialized in the constructor - instantiate them where needed.

Events

Definition: Events provide a new syntax for creating task requests, making code more readable than calls like `_taskManager.AddTaskRequest(12345, params)`.

Note: This might not be used in practice and may be deprecated.

Example:

```
IEventBus eventBus = new EventBus();

var userDetailsChangedEvent = new UserDetailsChangedEvent

{

            UserId = 123456,
            NewUsername = "Joe Bloggs"

};

eventBus.Raise(userDetailsChangedEvent);


internal class UserDetailsChangedEvent : IEvent

{

            public int UserId { get; set; }
            public string NewUsername { get; set; }
            public int EventTypeId()
            {
                return 123423;
            }
            public string EventName()
            {
                return "User Details Changed";
            }
            public IDictionary<string, object> Parameters()
            {
                return new Dictionary<string, object>
                {
                    { "UserId", UserId },
                    { "NewUsername", NewUsername }
                };
            }

}
```

Execution Chains

CommandHandler Execution Chain

1. Command received

2. ValidationInterceptor triggers (via Castle AOP)

3. Validator executes (FluentValidator)

4. If valid, CommandHandler executes

5. Business logic performed

6. Database operations (via Writers)

QueryHandler Execution Chain

1. Query received

2. Authorization check

3. QueryHandler executes

4. Reader retrieves data from database

5. Results returned

Documents

IDocumentRepository is the central API for loading and storing documents.

Critical Rules:

- ■ **Do NOT access the DocumentImages table directly**

- ■ **Always use IDocumentRepository**

Why: A new feature is being built with a new implementation of `IDocumentRepository` that stores blob data in a 3rd-party filestore (rather than in the DocumentImages table). Any code bypassing this API won't be able to use the new filestore.

See **Fnz.Framework.DocumentRepository Developer Guide** for more information.

Configuration / Bootstrapping

Configuration Assembly Structure

Each configuration should have its own configuration assembly:

- `StandardLife.Platform.Config`

- `Axa.Platform.Config`

- `Barclays.Platform.Config`

This is where Castle Windsor Container bootstrapping is performed and platform-specific classes are configured.

Castle Windsor Installers

We have a Castle Windsor "Installer" per DLL or area of functionality:

Installer	DLL	Registers
`DataAccessInstaller`	`Fnz.Framework.dll`	`IDataAccess` → `Dal`, `IExceptionLogger`, `IQueryBuilderFactory`
`CoreFrameworkInstaller`	`Fnz.Platform.Framework.dll`	All `DataAccessBase` and `RecordsetMappers` (In Fnz.Framework 5, now in `Fnz.Framework.Cca.dll`)
`PlatformInstaller`	`Fnz.Platform.dll`	All `ICommand/QueryHandlers` and other specific registrations

| `ReportsInstaller` | `Fnz.Platform.Reports.dll` | All `ICommand/QueryHandlers` in Reports and other specific registrations |

Container Installation Example

Note: If using Fnz.Framework 5.0, follow that guide instead.

Container.Install(

```
            // Registers IDataAccess (Dal), IQueryBuilderFactory and IExceptionLogger
            new DataAccessInstaller(true),
            // Registers ISystemVariablesReader and IEmail
            new ComponentsInstaller(),
            // Registers all DataAccessBase and RecordsetMappers classes
            new CoreFrameworkInstaller(),
            // Registers all DataAccessBase, RecordsetMappers, CommandHandlers,
            // QueryHandlers and Validator classes
            new PlatformInstaller()
```

);

Base Installer Class Structure

Each entry point (website, task loaders, services) must perform Castle installation on startup.

BaseInstaller (Abstract class)

- Creates the Castle Windsor Container

- Creates the ServiceLocator class (used in legacy code when dependency injection isn't available)

- Registers Castle facilities (logging, collection, factory)

- Calls `Container.Install` with required installers

TaskInstaller (Extends BaseInstaller)

- Usually doesn't do anything except override BaseInstaller

- Requires system variable in DB to call this class when task loaders startup

LegacyWebInstaller (Extends BaseInstaller)

- Usually just registers DataAccessInstaller and ComponentsInstaller

- Requires code changes to call this class

(Distribution)ServicesInstaller (Extends BaseInstaller)

- Registers DataAccessInstaller and ComponentsInstaller

- Adds service-specific registrations

- Requires code changes to call this class

Task Configuration / Bootstrapping

In **Fnz.Framework 4.x**, Task Loaders have Castle Windsor Container built in, allowing the task scheduler to call CommandHandlers directly.

How It Works

- Tasks with `classname` column ending in `CommandHandler` will be resolved from the container

- Task loader looks for two system variables on startup:

- `ConfigurationAssembly`

- `ConfigurationInstaller`

Example Installer:

namespace Axa.Framework.Config

{

```
public class TaskInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Install(SomeInstaller(), AnotherInstaller());
        ServiceLocator.CreateWithContainer(container);
    }
```

```
            }
}
```

Database Configuration:

```
INSERT INTO [SystemVariables].[dbo].[SystemVariables]
([Location], [Application], [Variable], [Value], [id])
VALUES
('Global', 'TaskManager', 'ConfigurationAssembly', 'Axa.Platform.Config', 'TBC')

INSERT INTO [SystemVariables].[dbo].[SystemVariables]
([Location], [Application], [Variable], [Value], [id])
VALUES
('Global', 'TaskManager', 'ConfigurationInstaller', 'Axa.Platform.Config.TaskInstaller', 'TBC')
```

Webforms Configuration / Bootstrapping

Webforms applications use the configuration assembly in `Application_Start`:

```
public class Global : HttpApplication
{
            private static IWindsorContainer _container;
            protected void Application_Start(object sender, EventArgs e)
            {
                // If website is shared, installer could be set via systemvariable/web.config
                _container = new WindsorContainer();
                _container.Install(new Axa.Platform.Config.WebInstaller());
                ServiceLocator.CreateWithContainer(container);
            }
            // ...
}
```

MVC / WebApi Service Configuration / Bootstrapping

Hosts with existing containers should install the configuration installer:

_container.Install(new Axa.Platform.Config.WebInstaller());

ServiceLocator.CreateWithContainer(container);

Existing dependencies can (should?) be moved into the installer.

Product / Legacy Platform Configuration

Legacy platforms (AXA, SL, JPM, Close, etc.) should call `Fnz.Platform.Legacy.PlatformInstaller` **before any other installer**. This overrides unwanted platform functionality (e.g., Customer Centric).

Smoke Tests

Critical: Since bootstrapping happens during application startup, failures prevent the website or task loaders from starting.

Required: Write xUnit smoke tests to verify the entry point installer works.

See `src_one_dev 'test\single.platform.config.unit.test'` for examples.

Test approach:

1. Call TaskInstaller, LegacyWebInstaller, or ServicesInstaller

2. Assert items can be resolved directly from container

3. Assert items can be resolved via ServiceLocator

Common Errors

"Component 'IDataAccess is already registered"

- **Cause**: `container.Install(new DataAccessInstaller(true))` called twice

- **Common scenario**: TaskInstaller calling DataAccessInstaller when IAPP task loaders already install it

- **Solution**: TaskInstaller should NOT call DataAccessInstaller

---

Security

UserContext

UserContext provides information about the currently executing user:

- User Id

- Branch

- Company

- Network

- WrapProvider

Initialising UserContext

UserContext is populated when:

- User logs onto the website

- User makes a web service call

- Each subsequent web request or web service call

Services: Configure via `Fnz.Platform.Services.Core.XUserContextAuthorizationFilter`

ASP.NET MVC: May be able to do something similar (TODO: clarify)

ASP.NET Websites: Modify FnzPage `Authenticate` method:

private const string XUserContextHeader = "x-usercontext";

private void SetPrincipal(int userId)

{

```
            var principal = CreatePrincipal(userId);
            Thread.CurrentPrincipal = principal;
            if (HttpContext.Current != null)
            {
                HttpContext.Current.User = principal;
            }
```

}

private ClaimsPrincipal CreatePrincipal(string userId)

{

```
            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.Name, string.Empty),
                new Claim(ClaimTypes.NameIdentifier, userId)
            };
            var claimsIdentityCollection = new ClaimsIdentityCollection
            {
                new ClaimsIdentity(claims, XUserContextHeader)
            };
            return new ClaimsPrincipal(claimsIdentityCollection);
```

}

Using UserContext

Step 1: Inject `IUserContextProvider` into your Handler/Reader/Writer:

```
public CreateIssueCommandHandler(IIssueWriter issueWriter, IUserContextProvider userContextProvider)

{

            _issueWriter = issueWriter;
            _userContextProvider = userContextProvider;

}
```

Step 2: Call `GetUserContext()`:

```
var userContext = _userContextProvider.GetUserContext();

var userId = userContext.UserId;
```

Transaction Control

- Commands are atomic

- Use Castle interceptor to bootstrap a `TransactionAndConnectionScope` (if one doesn't already exist)

Testing

Unit Testing

The CCA distinguishes between:

- **Unit Tests**: Don't access database/disk (`Single.Platform.Unit.Test`)

- **Integration Tests**: Access database/disk (`Single.Platform.Integration.Test`)

See examples of unit testing in the codebase.

Integration Testing

See examples of integration testing in the codebase.

Logging

log4net is used for logging throughout the platform.

See examples of logging in the codebase.

Exceptions

Exception Handling Principles

- **'Try' 'catch' (and possibly 'log')** at the application level unless you can do something about the exception

- **Recoverable errors** are usually not logged

- **Use .NET exception types** or define your own exception type

- ■ **Don't throw the standard `Exception` class**

Exception Logging Example

internal class PurchaseEquityCommandHandler

{

```
// Configured by Castle to log to the ErrorHandler table
private IExceptionLogger _exceptionLogger;
public PurchaseEquityCommandHandler(IExceptionLogger exceptionLogger)
{
    _exceptionLogger = exceptionLogger;
}
public void Execute(PurchaseEquityCommand command)
{
    try
    {
        // ... attempt to execute command
    }
    catch (InsufficientFundsException e)
    {
        _exceptionLogger.LogException(e, command.Customer,
            "Could not buy the equity because the customer ({0}) does not have enough Cash avail
            .FormatWith(command.Customer.Id, e.RequiredFunds, e.ActualFunds));
        // Potentially do something else here instead of throwing
        // e.g. purchase a smaller amount of funds
        throw;
    }
}
```

}

Note: IAPP task loaders automatically catch and log exceptions when calling CommandHandlers. Similarly, the website *should* do something similar - this avoids boilerplate try...catch...log in every command/query handler.

FAQ

Q: Where should my new code go?

A:

- **New CCA-style code**: `Single.Platform` / `Single.Platform.Api`

- **CCA code interfacing with legacy**: `Single.Platform.Adapters`

- **Legacy modifications only**: `Single.Platform.Legacy.sln` (no new projects!)

Q: How do I access documents?

A: Always use `IDocumentRepository`. Never access the DocumentImages table directly, as the platform is moving to a 3rd-party filestore.

Q: Why am I getting "Component 'IDataAccess is already registered"?

A: You've called `DataAccessInstaller` twice. This commonly happens in TaskInstaller when IAPP task loaders already install it. TaskInstaller should NOT call DataAccessInstaller.

Q: Should I create separate DTO and internal entity classes?

A: No. Don't create another class for mapping an API entity/dto to an internal object. Just have one class in the Api project (e.g., `Fnz.Platform.Api.User`).

Q: Where should validators live?

A: In the `Validators` subfolder under `CommandHandlers` folder.

Q: How do I handle multiple commands for the same entity?

A: Implement Create, Update, and Delete commands in one CommandHandler to prevent an explosion of handler classes:

IssueCommandHandler : ICommandHandler,

```
                                     ICommandHandler<UpdateIssueCommand>,
ICommandHandler<DeleteIssueCommand>
```

Examples

Real World Examples

Calling from Tasks

In **Fnz.Platform 10.0**, configure a task in the `tasksdb..tasks2` table with:

- **classname**: CommandHandler interface

- **ProcedureName**: `Execute`

Note: This relies on having a `.Platform.Config.sln` for Castle container bootstrapping.

Example:

UPDATE taskdb..tasks2

SET classname = 'IBlahCommandHandler', procedurename = 'Execute'

WHERE id = 123456

When the task runs, it will:

1. Resolve `IBlahCommandHandler` from the container (rather than calling 'new' using reflection)

2. Map the parameters recordset to a `BlahCommand` instance

3. Call the `IBlahCommandHandler.Execute(BlahCommand)` method

For previous versions, see "Common Code Architecture Guide - Archived".

Calling from Legacy Components and Websites without DI

Legacy components and websites should only reference Platform API project(s) - not the implementations.

Key Difference: Legacy components don't use Castle (Dependency Injection), so we use **Service Locator** pattern.

Service Locator

Class: `Fnz.Platform.Framework.Configuration.ServiceLocator`

Example - SL legacy component usage:

```
public ConversionStatementGenerator(IDataAccess dataAccess)

{

            _fundConversionStatementTransactionsProcessedWriter =
                ServiceLocator.Instance.Resolve<IFundConversionStatementTransactionsProcessedWriter>();
            // Don't forget to release Castle's reference to prevent memory leaks
            // Call immediately if object doesn't implement IDisposable
            // Otherwise call Release when this object is disposed
            ServiceLocator.Instance.Release(_fundConversionStatementTransactionsProcessedWriter);

}

public override ConversionStatement Populate(CRecordset3 parameters)

{

            _fundConversionStatementTransactionsProcessedWriter.UpdateProcessed(...);

}
```

■■ **Important**: Always prefer dependency injection over service locator. Service locator is an **anti-pattern** as it hides dependencies. Use only when constructor injection is not possible (e.g., large legacy classes that can't be easily refactored).

Requirements: ServiceLocator requires components to bootstrap the container. See "Task Configuration / Bootstrapping".

How to Upgrade to New Version of Fnz.Framework/Fnz.Platform

Scenario: Upgrading from `Fnz.Platform.6.0.1-DEV023` to `Fnz.Platform.6.0.1-DEV030` without spending half a day on VS Package Manager NuGet updates.

Solution: Use the PowerShell script in `Fnz.Build`:

- See **Fnz.Platform Versions & Streams#Consuminganewversion**

Best Practices

Non-Functional Standards

StyleCop

Purpose: Ensure consistency with coding standards across all Platform code.

Configuration:

- Default Microsoft StyleCop settings

- Exceptions defined in `src\settings.stylecop` (copied from `packages\Fnz.Build` when running `build.bat`)

- Detailed rules described in StyleCop documentation

Requirements: Microsoft StyleCop 4.4.0.x or later

Running StyleCop:

1. In Visual Studio: Tools → 'Run StyleCop'

2. Results appear in Output window (e.g., "SA1025 There should be no space ...")

3. Violation count displayed

ReSharper Integration:

- Install **StyleCopForResharper** plugin

- Shows StyleCop violations as you type

- Enables ReSharper quick fixes

- Code cleanup can auto-fix violations

See "Setting up AVD" for installation details.

NDepend

Purpose: Enforce various code rules on CI server.

Configuration: Detailed rules described in NDepend documentation.

Monitoring:

- Check CI build and tests job is green before promoting code

- Check if 'Fnz.Platform - Verify Code Metrics' has changed

- Ensure violation count hasn't increased

- Review NDepend Report tab for failure details

Example Failure Report:

// Types with too many methods

warnif count > 0

from t in Application.Types

where t.NbMethods > 30 &&

```
            !t.HasAttribute("Fnz.Framework.Core.Util.CodeQuality.IgnoreNumberOfMethodsAttribute".AllowNoMatc
```

...

BankAccount 35 Fnz.Platform.Api.Accounts.BankAccounts.BankAccount

Interpretation:

- `BankAccount` has 35 methods (maximum: 30)

- Can exclude with `IgnoreNumberOfMethodsAttribute` (use as last resort, e.g., generated code)

- Reasoning: http://www.ndepend.com/metrics.aspx#NbMethods

Note: Local NDepend licenses not available - must wait for CI to check for failures.

Folder Structure and Naming Conventions Examples

Fnz.Platform Structure

Fnz.Platform\Regulations

■■■ CommandHandlers

■ ■■■ AccountFatcaCommandHandler.cs

■■■ DataAccess

■ ■■■ Mappers

■ ■ ■■■ AccountEligibilityCheckFailureReaderMapper.cs

■ ■■■ Readers

■ ■ ■■■ AccountRegulationComplianceReader.cs

■ ■ (Contains IAccountRegulationComplianceReader &

■ ■ AccountRegulationComplianceReader)

■ ■■■ Writers

■ ■■■ AccountEligibilityCheckFailureLogWriter.cs

■ (Contains IAccountEligibilityCheckFailureLogWriter &

■ AccountEligibilityCheckFailureLogWriter)

■■■ QueryHandlers

■ ■■■ GetAccountRegulationComplianceQueryHandler.cs

■■■ Setup

■ ■■■ RegulationInstaller.cs

■■■ Validation

```
■■■ AccountFatcaValidator.cs
```

Note: All classes are in the namespace that matches the folder structure.

Fnz.Platform.Api Structure

Fnz.Platform.Api\Regulations

■■■ Commands

■ ■■■ ISaveAccountComplianceCommandHandler.cs

■ ■■■ SaveAccountComplianceCommand.cs

■ ■■■ [other files ...]

■■■ Entities

■ ■■■ CustomerRegulation.cs

■ ■■■ [other files ...]

■■■ Queries

```
            ■■■ CheckEligibilityRulesQuery.cs
            ■■■ ICheckEligibilityRulesQueryHandler.cs
            ■■■ [other files ...]
```

Note: All classes are in the namespace that matches the folder structure.

Database and Metadata

The database schema is held in the **same stream as the code**.

Local NuGet Server

Purpose: Deploy and consume Fnz.Platform changes locally during development for testing before promoting.

One-Time Setup

Step 1: Create NuGet repository folder

C:\nugetlocal

Step 2: Add package source in Visual Studio

1. Tools → Library Package Manager → Package Manager Settings → Package Sources

2. Click plus icon

3. Name: "local"

4. Source: "C:\nugetlocal"

Step 3: Update client platform `.NugetTargets` file (don't promote this change)

```
            <PackageSource Include="http://penuget2/nuget" />
            <PackageSource Include="local" />
```

Using the Local NuGet Server

Step 1: Package and install locally

local-nuget-install-all.bat

Example:

local-nuget-install-all.bat 1.0.0-SCP016

This script lives in the root folder of the Core Platform code.

Step 2: Update client code

- Update `packages.config` and `*.*proj` files to use the new version

- See **Fnz.Platform Versions & Streams#Consuminganewversion**

Step 3: Run build

dll_compile.bat

NuGet will download platform DLLs from the local NuGet server.

Do's ✓

- Always use `IDocumentRepository` for document operations

- Follow folder structure hierarchy strictly

- Use dependency injection over service locator

- Implement Create/Update/Delete in one CommandHandler where possible

- Release ServiceLocator references to prevent memory leaks

- Write smoke tests for all bootstrapping entry points

- Check NDepend metrics before promoting code

- Use FluentValidator for command validation

- Document public API with XML comments

- Instantiate mappers where needed (not in constructors)

- Follow naming conventions consistently

Don'ts ✗

- Never access DocumentImages table directly

- Don't import CCore02 dll (use new DAL instead)

- Don't create separate DTO mapping classes

- Don't add new projects to Single.Platform.Legacy

- Don't add first-level folders without checking with #Common Code Design

- Don't have validators depend on query/command handlers

- Don't throw standard `Exception` class (use specific types)

- Don't initialize mappers in constructors

- Don't call DataAccessInstaller twice

- Don't use service locator when dependency injection is possible

---

Troubleshooting

Issue 1: [Problem Description]

Symptoms: What you'll see when this occurs

Cause: Why this happens

Solution:

1. First step to resolve

2. Second step to resolve

3. Verification step

Issue 2: [Problem Description]

Symptoms: Observable signs of the issue

Cause: Root cause explanation

Solution: How to fix it

Glossary

- **Term 1**: Definition

- **Term 2**: Definition

- **Term 3**: Definition

- **Term 4**: Definition

Additional Resources

Documentation

- [Resource 1](url)

- [Resource 2](url)

Tutorials

- [Tutorial 1](url)

- [Tutorial 2](url)

Community

- [Forum/Community link](url)

- [Support channel](url)

Version History

- **v1.0.0** (2026-02-12): Initial version

- **v1.1.0**: [Future updates]

Contributing

Information about how to contribute to this knowledge base.

License

Specify the license or usage terms if applicable.