Common Component Specification

OpenEdge Framework Technical Design Document

OpenEdge Application Architecture Specification (OEAA)

Version	1.0

Spec Team Members		
Freddy Boisseau	flb@smsgroup.com	Structured Management Systems Inc.
Shelley B. Chase*	schase@progress.com	Progress
Rom Elwell*	rome@issol.com	Innovative Software Solutions
Mike Fechner	mike.fechner@consultingwerk.de	Consultingwerk
Damian Fernando	damianf@kingslake.com	Kingslake
Tharanga Herath	tharangah@kingslake.com	Kingslake
Prabhu Jha	prabhu@jktech.com	JKT India
Patrick O'Rielly	patricko@mip.co.za	MIP
Robin Smith*	rosmith@progress.com	Progress-Bravepoint

** Denotes contributing author*

TABLE OF CONTENTS

1 Introduction

The Common Component Specification (CCS) project is designed to simplify the development of modern business applications by defining the architecture and components of a modern application development framework for the OpenEdge platform. The CCS project identifies a set of specifications for the common components needed in developing modern business applications. When these components are built as part of a modernization framework, application developers can concentrate more on the business logic of the application rather than on infrastructure and integration.

The *CCS Specification: OpenEdge Application Architecture Version 1* builds on the Progress® OpenEdge® Reference Architecture (OERA) blueprint and defines a prescriptive architecture to use when building Enterprise Business Applications with OpenEdge. This version uses a service-oriented architecture (SOA) design in which application components provide services to other components via a communications protocol. The principles of service-orientation are independent of any vendor, product or technology.

This architectural specification defines the components of an OEAA-compliant architecture, the responsibilities of each component, and the communication protocol between components. It is this spec's responsibility to define the overall architecture for an OEAA-compliant framework and identify the minimum required components. A separate, detailed CCS specification is provided for each component.

These CCS specifications can be used in multiple ways. Some vendors will provide complete framework implementations, others might provide partial, focused frameworks, and others will implement single components for use with an OEAA-compliant framework. For example, a security component might be provided by a specific vendor that specializes in security while another vendor focuses on UI metadata. For this reason, every component will be versioned independently of the CCS architectural version. As long as the components follow the architectural specification, components compliant to that architecture should work together nicely. For this reason, component specifications must identify the CCS OEAA architectural version or versions for which they are compatible.

1.1 Purpose

The transformation of OpenEdge applications is happening globally with a number of different modernization frameworks. These frameworks provide needed common functionality, such as security, configuration, and session management but are developed independent of each other and are not based on any standards or common architecture principles.

The goal of the CCS project is to define a prescriptive architecture and standard set of specifications for the common components used in business applications by engaging the OpenEdge community and leveraging its expertise in building the best enterprise business applications. Each specification will include the API definitions as Object-oriented ABL (OOABL) interfaces, the expected behavior and other collateral to sufficiently define the component.

1.2 Scope

This OpenEdge Application Architecture (OEAA) specification defines a prescriptive architectural framework for building OpenEdge applications. It defines the model and components of an application as well as the responsibilities of each component. The bootstrapping and interaction between components is also covered in this specification.

Version 1 of the OEAA focuses on the common components. Each of these components is covered in individual CCS specifications, whose details are outside the scope of this specification.

This document also defines the minimum set of requirements for a framework to be OEAA-compliant.

2 Architecture Overview

The OEAA Version 1 defines a framework for the modernization of OpenEdge applications. This architecture builds on the current Progress® OpenEdge® Reference Architecture (OERA) and goes a step deeper by defining a set of ABL Interfaces and APIs for each major subsystem.

CCS specification authors should refer to this document for guiding principles.

2.1 OpenEdge Application Architecture (OEAA)

The OEAA Version 1 has the following subsystems and components. The Common Services components are the focus for version 1 of the OEAA and as such are the scope for this specification.

- Common Components
    - Startup Manager
    - Session Manager
    - Service Manager
    - Connection Manager
    - Property Manager
    - Context Data Manager
    - Authorization Manager
    - Authentication Manager
    - Messaging Manager
    - Logging Manager
    - Translation Manager
    - Analytics Manager

- Data Access
    - Data Servers
    - Database: Open or SQL
    - Data Synchronization DB

- Business Services
    - Business Entity
    - Data Synchronization Temp-table
    - Workflow
    - Task

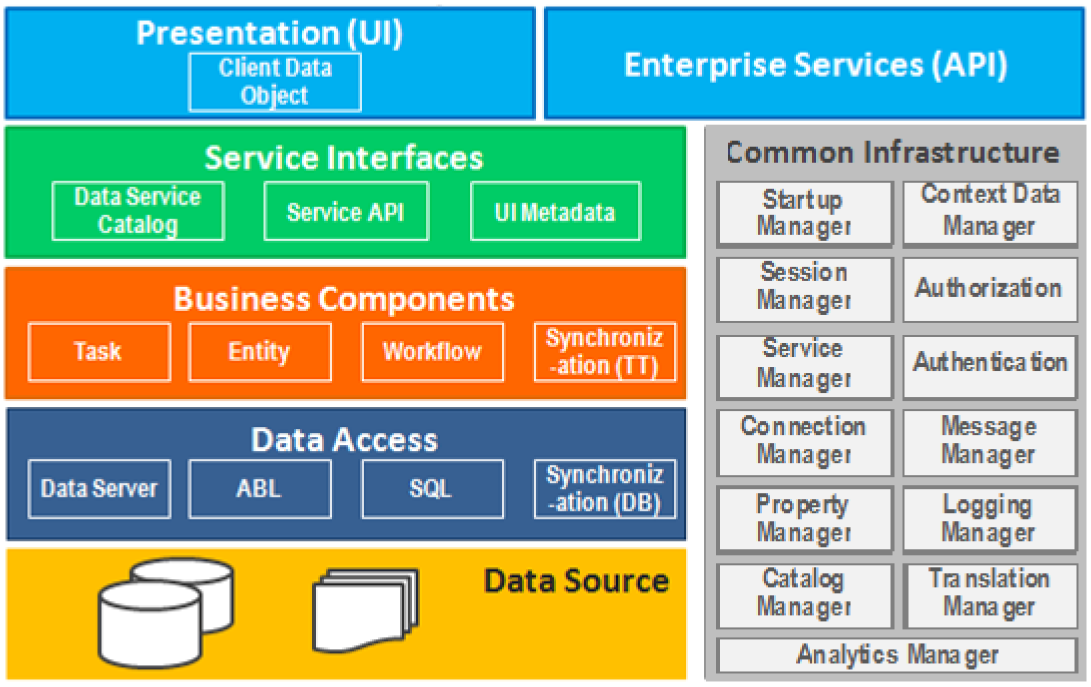
- Service Interfaces
    - Data Service Catalog
    - Service API
    - UI Metadata

- Presentation
    - Client Data Object

2.2 OpenEdge Application Architecture (OEAA) Diagram

The OEAA diagram shows the subsystems and components of the architecture. OEAA version 1 focuses on the Common Infrastructure subsystem.

OpenEdge Application Architecture



2.3 Compliance to the OEAA Version 1 Specification

A framework is OEAA-compliant as long as it follows the architectural model of this specification and implements all required components according to their individual specifications. On its own a framework with only the required components will not be very useful. Recommended components are beneficial to an application architecture framework but are not mandatory.

Compliance to the CCS enables the components of different frameworks and vendors to work interchangeably. CCS compliance will also offer the ability to choose between implementations thus providing developers the flexibility to swap components based on merit.

In addition, developers will be able to switch implementations with a minimal amount of porting effort.

Finally, the Common Components specifications will also enable creation of standard tools that can be used across multiple framework vendors, thus providing more productivity to framework end users.

2.4 Component Naming Requirements

Component specifications define APIs as OOABL interfaces according to the following requirements:

Key:

- PascalCasing – start with capital letter, first letter of each subsequent word capitalized
- camelCasing – start with lowercase letter, first letter of each subsequent word capitalized

Type	**Standards to be followed**	**Example**
Namespaces	Use Ccs*.<oeaalayer>* as root. Pascal case, no underscores. Note that any acronyms of three or more letters should be Pascal case (Oeaa instead of OEAA) instead of all caps.	Ccs.Common.* Ccs.DataSource.* Ccs.DataAccess.* Ccs.BusinessComponents.* Ccs.ServiceInterfaces.* Ccs.Presentation.* Ccs.EnterpriseServices.*
Interfaces	Interface names should begin with the capital letter "I" and use the Pascal casing. Interfaces should not have the same name as the namespace in which they reside. Any acronyms of three or more letters should be Pascal case, not all caps. Try to avoid abbreviations, and try to always use nouns.	Ccs.Common.IStartupManager Ccs.Presentation.IClientDataObje ct
Classes	Classes that implement a CCS interface can use any namespace they like. Class Naming MUST follow the Pascal case, no underscores or leading "C" or "cls" are recommended. If any Class name begins with letter “I” should not have the letter following in Capital letter (may contradict with an Interface	Com.Myvendor.StartupManager Myvendor.Common. StartupManager
Methods	Method names MUST use Camel case.	getManager( )
Properties	Property names MUST use Pascal case.	CurrentClientContext

2.5 Component Specification Versioning

The component versioning MUST follow semantic versioning, or "semver", as defined at http://semver.org/. The Semantic Versioning specification is authored by Tom Preston-Werner, inventor of Gravatars and co-founder of GitHub.

The gist of it is - given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,

2. MINOR version when you add functionality in a backwards-compatible manner, and

3. PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

2.6 Component Documentation (For Implementation)

All comments in source code files MUST adhere to “C-Style” formatting, which allows tools such as ABLDoc (OpenEdge) or the PCT Class Documentation task to generate reference documentation. This style is commonly referred to as a “Multi-line” comment.

2.7 Component Property Data and Organization

Component Specifications can contain the following:

- Public OpenEdge data types exposed in the class and interfaces
- Data naming conventions
- External data storage organization requirements (if any)
- ABL Enums or compatible ABL interface
- Any data collections used by an interface and how that collection’s data values are presented to an application developer

2.8 Component Communication

Application complexity and scope vary widely and a single communication model is not appropriate. As a result this specification defines two available models, one for smaller applications (single domain) versus complex applications (multi-domain). Both communication models define how a component of OEAA communicates with another component of OEAA.

Communication between components within a single application domain

This style is used for communication within a single OEAA layer and between OEAA layers. All functionality is encapsulated in a single domain so no “façade” is required. The goal is to allow customization of a component API by using the Service Manager to obtain a reference to the component. The calling component does not need to determine which component implementation to use because it is all managed by the configuration of the Service Manager.

This communication model allows the calling component to query the Service Manager for a reference to another managed component. The calling component can then use this reference to access the methods on the associated components directly.

Communication between application domains

This style is used for communication between application domains regardless if the component is in the same OERA layer, different OERA layers, a legacy system or an external system. The goal is to provide a single model to communicate outside of the current domain regardless of where or how that system is implemented. All functionality for a domain is defined by an external “façade” that abstracts the actual implementation.

This communication model allows the calling component to query the Service Manager for a reference to the interface type they wish to communicate with. The calling component can then use this reference to access methods on the API.

2.9 Component Error Handling

Exceptions that can be thrown from each interface method should be identified in the individual Component Specification.

A component can use Progress.Lang.SysError or define their own custom exceptions extended from Progress.Lang.AppError. Errors classes can be a general exception common to the framework or it can be a specific implementation relevant only to the respective component. Class naming should highlight the Type of Error and the Component if it relates to a component.

All methods of a component should return all errors to their caller. It is recommended that all calls to component methods be wrapped in a try-catch block. Samples in the component spec should reflect this recommended best practice

3 OpenEdge Application Architecture Components

There are 2 types of components in the OEAA – required and recommended. Required components are mandatory to bootstrap the framework. In other words a framework cannot exist without these components however a framework with only these components is not very useful.

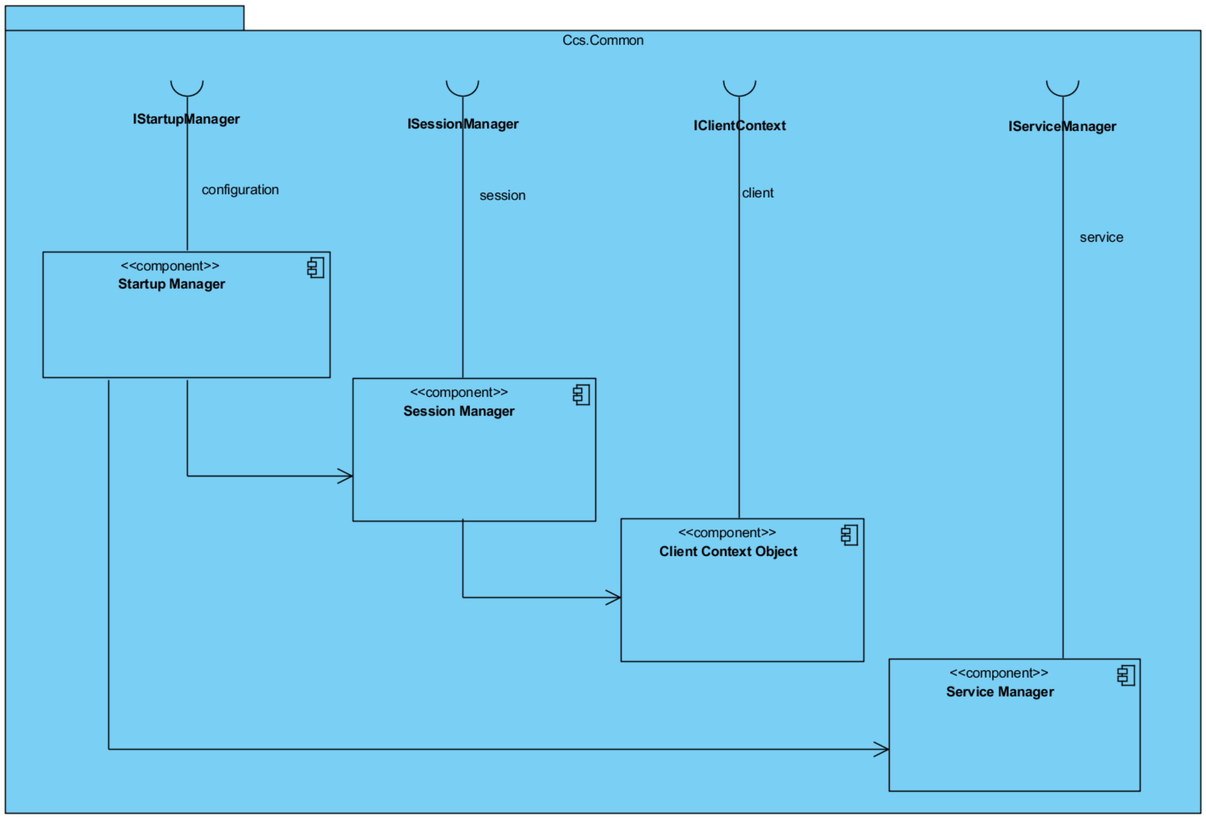
Recommended components provide specific functional behaviour that might not be needed by every framework.

It is important to note the difference between managers and services. Managers are a single instance and perform a specific task. A Service represents the business functions that are available to the client or other application components or services.

All managers are run as a single instance by the Startup Manager and persist for the lifetime of the ABL session. Services are provided and managed by the Service Manager which controls their lifecycle.

3.1 Required Components

The following diagram displays the static assembly for the required components of the Ccs.Common namespace – StartupManager, SessionManager (and context object) and ServiceManager, respectively. The diagram also displays the strong associations between the component StartupManager and the (object) references it contains for SessionManager and ServiceManager, respectively.



3.1.1 Manager Base Class (Ccs.Common.IManager)

This base class defines a common method that can be used by the Startup Manager to start any Manager. The method signature is:

```
initializeManager ( ).
```

Each Manager must implement a common initialization interface:

```
Using Ccs.Common.*.

Interface Ccs.Common.IManager:

    Method public void initializeManager ( ).

End interface.
```

3.1.2 Startup Manager (Ccs.Common.IStartupManager)

The Startup Manager is responsible for bootstrapping the session by instantiating all other common infrastructure components (managers) and providing a reference to each configured component. The Startup Manager is the single point within the application to reference or gain access to the other managers. In order to have a pluggable model for the framework and allow components from various vendors to work together, the Startup Manager is the factory that instantiates all other common components.

When the Startup Manager is instantiated, it will locate and load its own configuration data or have it passed in by a bootstrap process. This configuration data defines an implementation class for each common infrastructure component interface to be instantiated by the Startup Manager. Each common infrastructure component, or manager, including the Startup Manager, will implement a common interface ensuring that each manager implements a common initialization interface required by the Startup Manager and bootstrap procedure. The Startup Manager will then read through each configured service component, in the order they are defined, and instantiate the manager object. A reference to the object is then assigned to the relevant property in the Startup Manager as defined by the Startup Manager Interface.

The Startup Manager Interface would contain public properties referencing each common services component interface required by the CCS. For example:

```
Using Ccs.Common.*.

Interface Ccs.Common.IStartupManager inherits IManager:

    /* References to Service and Session Managers */
    Define public property ServiceManager as IServiceManager no-undo get.
    Define public property SessionManager as ISessionManager no-undo get.

End interface.
```

The Startup Manager must be able to provide a reference to all the configured managers and itself. The Startup Manager must also allow for multiple implementations of the Startup Manager Interface to be injected by a bootstrap process if required.

To provide a static reference to the Startup Manager, all CCS compliant frameworks must provide a reference to the Startup Manager. This class will contain a static public instance property and a private constructor. The instance property is defined as the Startup Manager Interface with a public setter, allowing for any implementation of the Startup Manager Interface to be injected by the bootstrap process.

For example, the StartupManager implementation may be defined as follows:

```
Using Ccs.Common.*.

Block-level on error undo, throw.

Class Ccs.Common.StartupManager:

    Define public static property Instance as IStartupManager no-undo get.
Set.

    Constructor private StartupManager ( ):

    End constructor.

End class.
```

A bootstrap procedure, configured as the application server startup procedure, would instantiate a new Startup Manager and set the Startup Manager Instance property.

```
/*------------------------------------------------------------------------
    File        : bootstrap.p
    Purpose     : Instantiate the ConfigManager
    Description : Ccs Framework startup procedure
  ----------------------------------------------------------------------*/
Using Ccs.Common.StartupManager.

Block-level on error undo, throw.

Define input parameter pcStartupParam as character no-undo.

StartupManager:Instance = new VendorA.Common.StartupManager().
```

```
StartupManager:Instance:initializeManager ( ).
```

To get a reference to any manager, the static Instance property is used to return the managers instance that was started by the Startup Manager. This example shows how the reference to the Service Manager is retrieved.

```
Define variable oServiceManager as IServiceManager no-undo.

oServiceManager = StartupManager:Instance:ServiceManager.
```

3.1.3 Session Manager (Ccs.Common.ISessionManager)

Each request made to the business services is for a particular client session. The client session has context information associated with it such as who is the logged in user, what branch the user is logged into, date and numeric formats, time zone, etc. The Session Manager is responsible for validating the authentication of the request made on the business services, establishing the application server runtime session to service the request and provide access to the client context data.

When establishing the session, the Session Manager must validate the authentication of the client request and then establish the context data on the first client request (or re-establish on subsequent requests) based on the client identity. The Session Manager may use other service components such as the Context Data Manager and Client Context Object to establish the session.

A sealed Client Principal Object (or Security Token) will be passed to the Session Manager that will identify who the client is. The Session Manager will use the C-P to authenticate the request and assert the client identity on the session. The Session Manager may simply assert the C-P against the session/databases relying on the database domain configuration for authentication and then if successful, establishes the session context.

The Session Manager must provide access to the client context data by instantiating a Client Context Object using the C-P as identity and assigning it to a public property on the Session Manager. This Client Context object will be defined as part of the Session Manager specification.

The Session Manager will also set the appropriate environmental properties such as date and numeric formats, time zone, etc. Once the request has been completed the Session Manager must “end” or “reset” the session. The Session Manager must reset the application server runtime session to a “safe” state so that the identity and context data of the client from the last request is not left asserted against the session and its databases.

An example of the minimum expected interface for the Session Manager is:

```
Using Ccs.Common.*.

Interface Ccs.Common.ISessionManager inherits IManager:

    Define public property CurrentClientContext as IClientContext no-undo
get.

    Method public void establishSession( input phClientPrincipal as handle
).

    Method public void endSession( ).

End interface.
```

3.1.4 Client Context Object (Ccs.Common.IClientContext)

The Client Context Object is a sub component of the Session Manager. The Session Manager establishes the session environment using the user (or client) context data. This data is made available through the Client Context Object.

The Client Context Object is the logical representation of the client session data and it represents the properties of the client session, such as who the user is, what the date and number formats are and any application specific data representing the state of the client session. This object is also responsible for initializing any application specific properties when a new session is created. This is a serializable object that can be persisted at the end of the request and re-instantiated at the start of subsequent requests.

The Client Context object can store as much information as is useful and provide a way for the programmer to request only the data they need for the current request,

3.1.5 Service Base Class (Ccs.Common.IService)

This base class defines a common method that can be used by the Service Manager to start any Service. The method signature is:

```
initializeService ( ).
```

Each Business Service must implement a common initialization interface:

```
Using Ccs.Common.*.

Interface Ccs.Common.IService:

    Method public void initializeService ( ).

End interface.
```

3.1.6 Service Manager (Ccs.Common.IServiceManager)

A Business Service is an object containing business functions that are either exposed to the client through the service interface or may be common reusable business functions used by other application components or services. These business services are typically defined class based objects or persistent procedures. These include, but are not limited to, Business Entities, Tasks and Workflow components.

The Service Manager is responsible for instantiating all Business Services and managing their life cycle. The Service Manager is used to instantiate these objects or procedures and shut them down as appropriate based on a life cycle configuration of the service. The Service Manager is the central controller or factory that ensures that Business Services are initialized consistently and not left consuming resources unnecessarily or started multiple times.

Each Business Service will implement a common interface ensuring that each service implements a common initialization interface required by the Service Manager.

The Service Manager is also responsible for instantiating a configured implementation of the requested service. This allows for a local customization to be provided as configuration data to the Service Manager for a service rather than having the application instantiate appropriate implementations for each deployment.

When the Service Manager is instantiated by the Startup Manager, the configuration data for the Service Manager is loaded. This configuration data will define the service implementations and life cycle configurations for the services. Since large enterprise applications contain 1000's of services, the configuration data for service implementations should be by exception where a custom implementation is required rather than having to configure all services. Likewise, the life cycle configuration for services should be defined using rules rather than defining the life cycle for each service.

The Service Manager is responsible for managing the life cycle of the business services. That is, how long services remains instantiated and available for reuse. The life cycle may be one of “single”, “request” or “session”.

- If the service life cycle is “single”, the Service Manager will instantiate the service and returns its reference but does not keep a copy of the service reference. The request initiator will be responsible for deleting the service object or leaving it to the garbage collection to destroy. If another request is made for the same service, the Service Manager will instantiate a new service and returns a new reference to the service.
- If the service life cycle is “request, the Service Manager will instantiate the service and returns its reference and will keep a copy of the service reference. If another request is made for the same service during the life time of the client request, then the same reference to the instantiated service will be returned without starting a new service. When the client request is complete, the Service Manager will be notified that the client request is complete and the Service Manager will shut down the service and remove the reference to the service.
- If the service life cycle is “session”, the Service Manager will instantiate the service and returns its reference and will keep a copy of the service

reference. If another request is made for the same service at any time, then the reference to the instantiated service is returned.

An example of the minimum expected interface for a Service Manager would be:

```
Using Ccs.Common.*.
Using Progress.Lang.*.

Interface Ccs.Common.IServiceManager inherits IManager:

    Method public IService getService( input poServiceClass as Class).

    Method public IService getService( input poServiceClass as Class,
                                       input pcInstanceName as character ).

    Method public void stopLifeCycle( input poLifeCycle as
ServiceLifeCycleEnum ).

    Method public void stopService( input pcServiceTypeName as character,
                                    input pcInstanceName    as character ).

    Method public Class getServiceImplementation(input poService as Class
).

End interface.
```

3.2 Recommended Components

3.2.1 Context Manager (Ccs.Common.IContextManager)

The Context Data Manager is a sub component of the Session Manager and is responsible for retrieving and storing/caching the session state or context data for each service request. The session context data is managed and made available by the Session Manager. The Context Data Manager is used to store the data so that it can be retrieved and made available across all Application Server sessions and agents for all subsequent requests by the client. The Context Data Manager needs to be able to deal with multiple simultaneous requests to retrieve and persist the client context data

3.2.2 Authentication Manager (Ccs.Common.IAuthenticationManager)

(Including the Security Token Service)
The Authentication Manager accepts a sealed client-principle and validates the token. This client-principle is normally generated by the Spring Security of a PAS for OE server, referred to the [external] *Security Token Service.* The Authentication Manager will assert the user's identity for the ABL session resources (AppServer connections, DB connections, etc.).

The sealed client-principle is stored in the client context for use by other components such as the Authentication Manager. For example: the Authorization Manager may obtain the current-user-identity's identity for resource access control.

3.2.3 Authorization Manager (Ccs.Common.IAuthorizationManager)

The role of the Authorization Manager is to implement role-based access-control for local ABL session resources. The Authorization Manager performs standard resource-action tests using the ABL session's current identity (i.e. the Client-Principal security token) established by the Authentication Manager. The business application uses the Authorization Manager to query a client's rights to execute some action on a resource before it is used.

3.2.4 Catalog Manager (Ccs.Common.ICatalogManager)

A Catalog Manager is responsible for providing the catalog definition for the JSDO call from the client and the definition of the service function so that the Service Manager can invoke the call to execute the client request. This is used for publishing a catalog of available services and supporting the JSDO catalog in a dynamic fashion.

3.2.5 Connection Manager (Ccs.Common.IConnectionManager)

The Connection Manager is responsible for creating and managing connections to external services. External services could be other ABL applications running on a separate application server or 3rd party web service. The connection manager is used to establish the connection and manage the life cycle of those connections.

3.2.6 Logging Manager (Ccs.Common.ILoggingManager)

The Logging Manager is responsible for logging, formatting and managing errors that are caught by the service interface and are encapsulated into the response message by the Message Manager. The Logging Manager provides a single interface for the application to log messages in a consistent format. The physical storage mechanism should be able to be specified by the caller such as file system, db…

3.2.7 Message Manager (Ccs.Common.IMessageManager)

The Message Manager is responsible for creating and managing message objects that can be passed between clients and servers in a decoupled system. The primary messages managed by the Message Manager are the request and response messages that are passed through the Service Interface for each client request. Each request made to the business services has request and response information associated with it or in other words, the input and output parameters to the business service. This information is defined as a request and response message that passes through the service interface. This could be a JSON object transported over HTTP. The Message Manager provides access to these 2 primary messages so that they are accessible from any business component or manager. The Message Manager is responsible for parsing the request message from the client and formatting the response message to be returned to the client.

3.2.8 Property Manager (Ccs.Common.IPropertyManager)

The Property Manager is responsible for providing access to configuration data that other common components may require. The Property Manager's interfaces would effectively produce a single methodology by which external configuration data could be consumed from one or more physical sources of varying types and locations.

3.2.9 Translation Manager (Ccs.Common.ITranslationManager)

The Translation Manager is responsible for language and context message translation. All application messages and text to be displayed to the client should be passed through the Translation Manager that will then translate the message or text to the appropriate message based on the current user context. This might be translating to the user's language or translating business terms based on the division or branch the user belongs to.

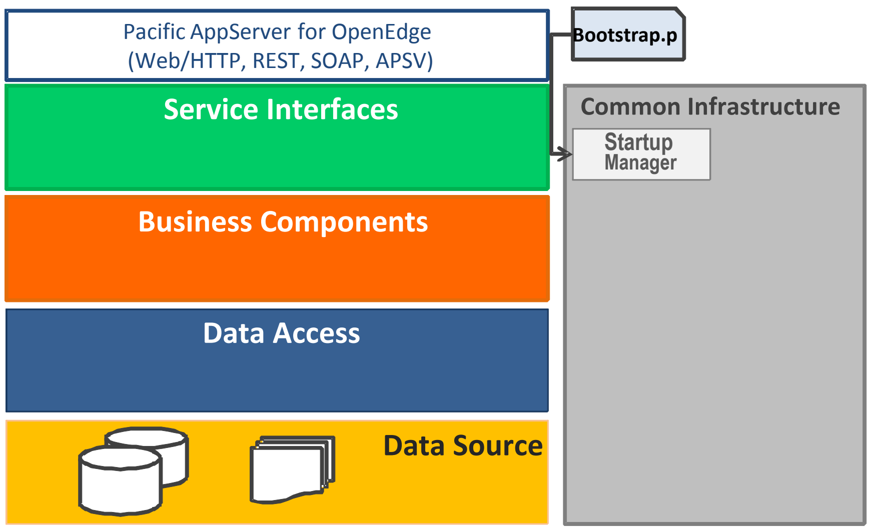
3.2.10 Analytics Manager (Ccs.Common.IAnalyticsManager)

The Analytics Manager is responsible for collecting data regarding the usage and metrics for the application. All data collected is persisted for reporting.

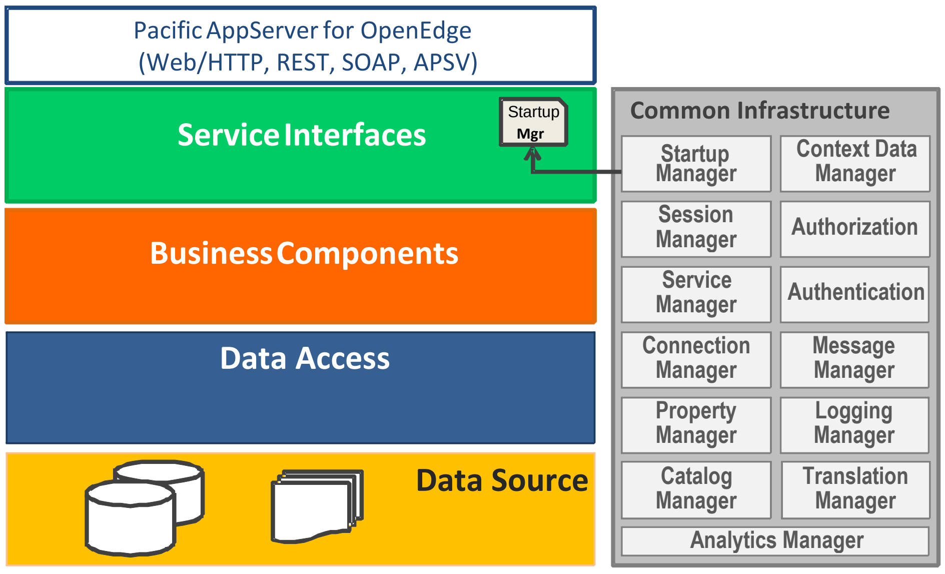
4 Sample Application Flow

4.1.1 Bootstrap the Server

The application server's start up procedure must bootstrap the framework by initializing a Startup Manager instance and assign the static Instance property of the CSS Startup Manager.



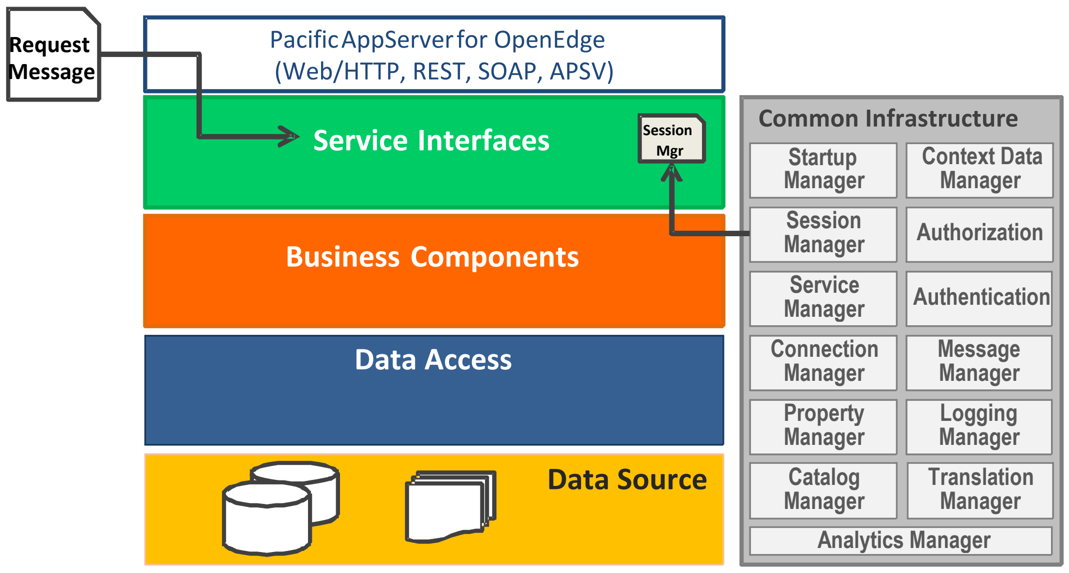
The Startup Manager will then instantiate all other common components that are registered. A reference to the Startup Manager is global for the session.



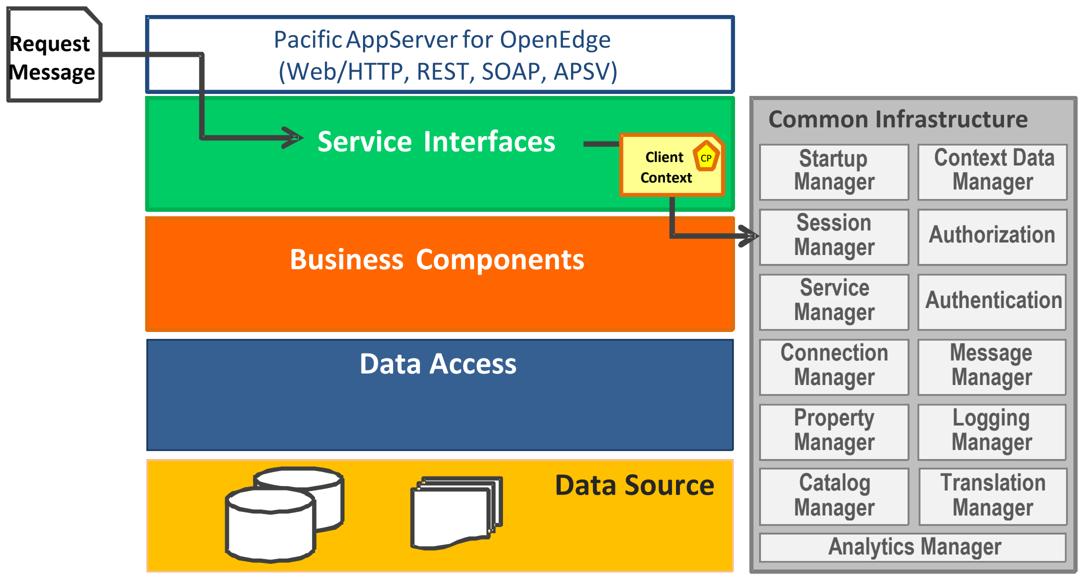
4.1.2 Establish a Session

When a request comes into a web server without a session identifier, the session must be created first and the user must be authorized. This initial request message often contains a signed and sealed security token that is passed to the Service Manager for validation.

For the initial request, the Service Interface must create the session. To do this it gets reference to the Session Manager from the Startup Manager.

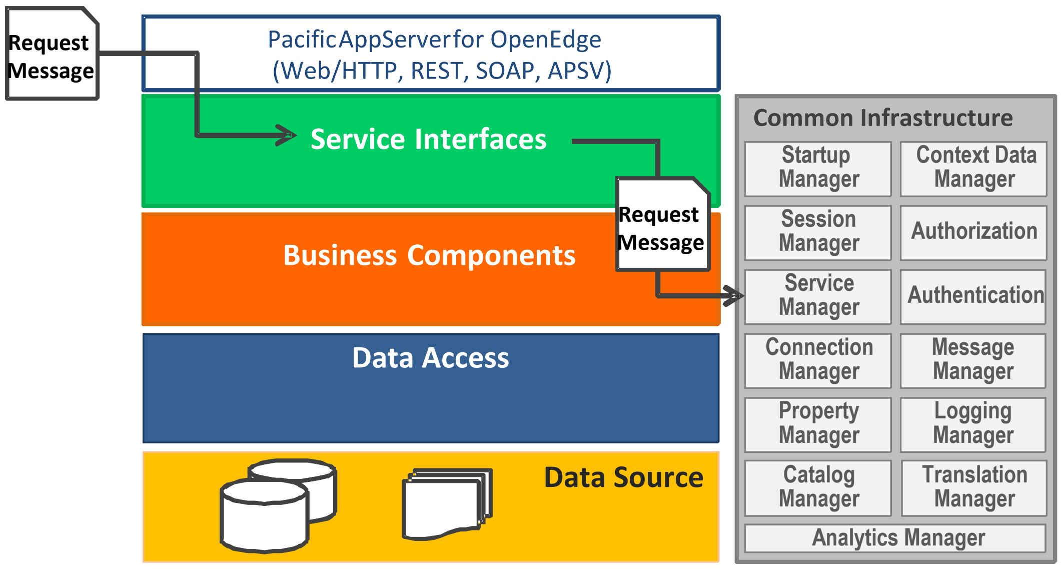


The Service Interface calls the Session Manager passing a signed and sealed security token (a Client-Principle Object) and the Session Manager establishes the session context.

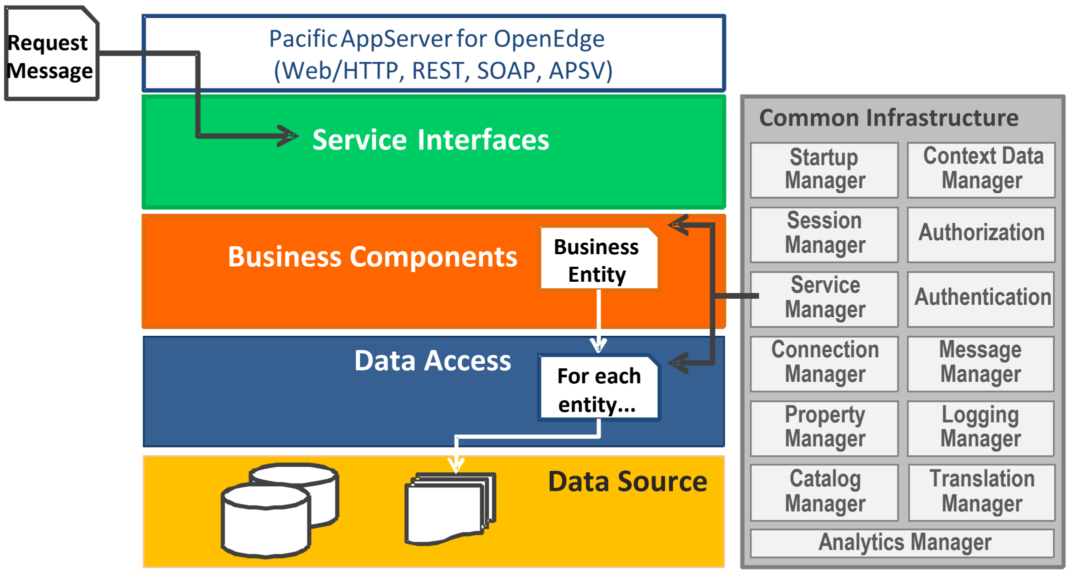


4.1.3 Call Service to handle the request

Now that the session is created, it is time to handle the request sent in to the Service Interface. The Service Interface calls the Service Manager with the service name and the request. The Service Manager runs the requested Business Service and passes the request to the Business Service.



The Business Service handles the request and creates the business objects. In this example, the Service Manager creates the Business Entity object which in turn requests a Data Access object.



Finally, the result is returned back to the caller through the Service Interface.

5 Document History

Date	Version	Author	Change Details
20-November-2015	1.0_CR	Shelley Chase Rom Elwell Robin Smith	Final draft for the Steering Committee.
24-January-2016	1.0_Final	Shelley Chase	Responded to feedback from the Ccs community. • Rename Configuration Manager • Clarify purpose of Client Context object • Use consistent casing in naming
27-January-2016	1.0	Shelley Chase	Team feedback. • Added IManager and IService interfaces. • Fixed more casing.