

|  |
| --- |
| State of Maine Evergreen Data System Project Maine Department of Health and Human Services Contract ADS-18-9451 |
| System Design Document v1.0 |
|  |
| February 13, 2019  Submitted to:  State of Maine  Department of Health and Human Services  Submitted by:  FEI Systems  9755 Patuxent Woods Drive  Columbia, Maryland 21046 |

|  |
| --- |
|  |

Table of Contents

[1. Introduction 1](#_Toc968463)

[1.1 Purpose of the SDD 1](#_Toc968464)

[2. General Overview and Design Guidelines/Approach 2](#_Toc968465)

[2.1 General Overview 2](#_Toc968466)

[2.2 Assumptions/Constraints/Risks 2](#_Toc968467)

[2.2.1 Assumptions 2](#_Toc968468)

[2.2.2 Constraints 3](#_Toc968469)

[2.2.3 Risks 3](#_Toc968470)

[3. Design Considerations 4](#_Toc968471)

[3.1 Goals and Guidelines 4](#_Toc968472)

[3.2 Development Methods & Contingencies 5](#_Toc968473)

[3.3 Architectural Strategies 5](#_Toc968474)

[3.4 Performance Engineering 7](#_Toc968475)

[4. System Architecture and Architecture Design 9](#_Toc968476)

[4.1 Logical View 9](#_Toc968477)

[4.2 Hardware Architecture 10](#_Toc968478)

[4.2.1 Performance Hardware Architecture 12](#_Toc968479)

[4.3 Software Architecture 13](#_Toc968480)

[4.3.1 High-level Architecture 14](#_Toc968481)

[4.3.2 User Interface 17](#_Toc968482)

[4.3.3 Application Server 18](#_Toc968483)

[4.3.4 Database 22](#_Toc968484)

[4.3.5 Security Software Architecture 23](#_Toc968485)

[4.4 Information Architecture 25](#_Toc968486)

[4.4.1 Records Management 25](#_Toc968487)

[4.5 Internal Communications Architecture 25](#_Toc968488)

[4.6 Security Design and Architecture 26](#_Toc968489)

[4.6.1 Identification, Authentication, and Authorization 27](#_Toc968490)

[4.6.2 Data Encryption and Integrity 27](#_Toc968491)

[4.6.3 Security Information and Event Management 27](#_Toc968492)

[5. System Design 29](#_Toc968493)

[5.1 Business Analysis 29](#_Toc968494)

[5.2 User Experience Design 30](#_Toc968495)

[5.2.1 User Centered Design (UCD) Approach 30](#_Toc968496)

[5.2.1 Lean UX 30](#_Toc968497)

[5.2.2 Usability and Accessibility 31](#_Toc968498)

[5.3 Domain Modeling 31](#_Toc968499)

[6. System Integrity Controls 33](#_Toc968500)

[6.1 Access to Critical Data 33](#_Toc968501)

[6.2 System Audit Controls 34](#_Toc968502)

[7. External Interfaces 35](#_Toc968503)

[Appendix A: Revision History 36](#_Toc968504)

List of Tables

[Figure 1 - Logical View 9](#_Toc968562)

[Figure 2 - Physical Model 11](#_Toc968563)

[Figure 3 - Domain Driven Design Layers 14](#_Toc968564)

[Figure 4 - Application Tiers 15](#_Toc968565)

[Figure 5 - Application Components 16](#_Toc968566)

[Figure 6 - Web Service Pipeline 18](#_Toc968567)

[Figure 7 - Metadata Example 20](#_Toc968568)

[Figure 8 - Organization Domain Model 21](#_Toc968569)

[Figure 9 - Organization Database Model 23](#_Toc968570)

[Figure 10 - Internal Communications Architecture 26](#_Toc968571)

## Introduction

This System Design Document (SDD) describes how the functional and nonfunctional requirements recorded in the Requirements Document transform into more technical system design specifications from which the system is built. This SDD documents the high-level system design and the low-level detailed design specifications.

The SDD describes design goals and considerations and provides a high-level overview of the system architecture. It describes the data design associated with the system, the human-machine interface, and operational scenarios. The high-level system design is further decomposed into low-level detailed design specifications for each system component, including hardware, internal communications, software, system integrity controls, and external interfaces. The high-level and low-level detailed design serve as input to the Detailed Design Review (DDR).

### Purpose of the SDD

This SDD documents and tracks the necessary information required to effectively define architecture and system design in order to give the development team guidance on the architecture of the system to be developed. Design documents are incrementally and iteratively produced during the system development life cycle, based on the particular circumstances of the information technology (IT) project and the system development methodology used for developing the system. Its intended audience is the project manager, project team, and development team. Some portions of this document, such as the user interface (UI), may be shared with the client/user, and other stakeholders whose input and approval into the UI is needed.

## General Overview and Design Guidelines/Approach

This section describes the principles and strategies to be used as guidelines when designing and implementing the system.

### General Overview

The scope of the Long-Term Services and Supports (LTSS) Care Management System Project spans planning, design, development, testing, training and implementation of the following Care Management software modules:

* Client Management
* Agency Management
* Staff Management
* Enrolment and Eligibility Determination
* Waitlist Management
* Plan of Care/Plan of Service Review and Approval
* Service Authorization
* Case Management
* Crisis Management and Critical Incident Reporting
* Adult Protective Services (APS) Intake and Referral
* APS Supervisor Review and Screening
* APS Investigation and Court Study

The project also includes development of four system interfaces with legacy systems with daily transactional batch file transfers, as well as initial data migration for the programs being transitioned the to the Care Management system (i.e., APS, Section 21, Section 29, Section 13, Section 18, Section 20, Section 102, and Preadmission Screening and Resident Review [PASRR]).

Within the discussion of the system design, the core FEI Systems product and system is referred to as “Carity” and the product and system configured to Maine DHHS requirements is referred to as “Evergreen.”

### Assumptions/Constraints/Risks

#### Assumptions

The following assumptions are considered during the course of determining the system design:

1. The Evergreen product and system is subjected to a battery of tests that culminate in obtaining Application Deployment Certification by DHHS. These tests provide evidence that the product and system conforms to DHHS policies referenced in the LTSS Care Management System Project contract ADS-18-9451, dated 06/18/2018.
2. The Evergreen system is accessed through desktop and laptop computing devices connected via local area networks (LAN) or wireless networks (WiFi). The system is also accessed by portable devices, such as tablets.
3. Features of the Evergreen system identified by DDHS, such as assessment forms, may be accessed in an off-line mode via a mobile device and subsequently loaded to the Evergreen system after the mobile device is connected to a LAN or WiFi network.

#### Constraints

The following constraints are considered during the course of determining the system design:

1. The Evergreen system is housed within a primary contractor data center and a secondary data center, both within the continental U.S. The data centers house commercially available hardware, software, and network components. Remote hosting policy is defined by Maine Remote Hosting Policy.
2. Protected data (e.g., protected health information (PHI) and personally identifiable information (PII) within the Evergreen system cannot be housed in repositories outside of the continental U.S.
3. Evergreen end users generally use the system within an office environment. Mobile devices that may be used for off-line capability are used within an office, home, or healthcare facility settings.
4. Accessibility standards are defined through Maine Web Accessibility and Usability Policy.
5. The Evergreen product and system is subjected to a battery of tests that culminate in obtaining Application Deployment Certification by DHHS. These tests provide evidence that the product and system conforms to DHHS policies referenced in the LTSS Care Management System Project contract ADS-18-9451, dated 6/18/2018.
6. Interface with legacy systems for data integration purposes must be compliant with techniques and technologies mutually agreed to by FEI and DHHS OIT.
7. The system design must align with the Medicaid Information Technology Architecture 3.0 (MITA).

#### Risks

No risks associated with the system design have been identified. Should risks occur in the future, they will be addressed through the Evergreen project’s risk management process documented in the Project Management Plan.

## Design Considerations

### Goals and Guidelines

The implementation of the Carity system has been guided by a fundamental set of design principles that focus on the end users’ experience, architectural simplicity, and the security of PHI and PII. Those principles are:

* User Interface Responsiveness
* User Experience
* Security of PHI/PII
* Scalability
* Keep the System Simple

**User Interface Responsiveness—**In 1968, Robert Miller published his classic paper “Response Time in Man-computer Conversational Transactions.” He described three orders of magnitude of computer mainframe responsiveness:

* A response time of 100ms is perceived as instantaneous.
* Response times of 1 second or less are fast enough for users to feel they are interacting freely with the information.
* Response times greater than 10 seconds completely lose the user’s attention.

Based on these concepts, Miller concluded that a consistent 2-second response would be ideal. Years later, the 2-seconds response continues to be used as a performance target for web-based applications. Carity strives to achieve the 100ms mark of instantaneous response for common interactions such as edit operations, and a 1-second response time mark for complex queries and large screen loads. These response times are not always possible and of course are subject to network latency.

**User Experience—**Carity has been designed with a focus on all aspects of end users interaction. The screens are designed to be intuitive by following user interface design patterns that are common in modern web applications.

**Security of PHI/PII—**Maintaining PHI and PII security is paramount. The access control environment is configured to be “Secure by Default,” and all web service requests must pass through several layers of gated controls ensuring limited access to data.

**Scalability—**Web service end points are designed to be short-lived and granular. This enables Carity to support more users with less hardware. However, when needed, Carity can be scaled up to support larger numbers of concurrent users.

**Keep the System Simple—**Simplicity of design leads to a system that is easy to deploy, operate, and support. Code structures that follow industry best practices and established design patterns are easier to maintain and teach.

### Development Methods & Contingencies

The Carity browser based user interface is implemented with Google’s Angular Single Page Application (SPA) architecture. An SPA is a website that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. Rewriting the page is achieved by rendering user interface templates directly in the browser rather than “posting back” to the server to render new HTML pages. Since rendering is implemented in the browser, there is a reduced need for server-side CPU processing.

The Carity service oriented application (SOA) server is developed using Microsoft’s .NET technology and the Microsoft WebAPI framework. The source code is written in the C# programming language.

Microsoft’s SQL Server database engine is used as the transactional database backend along with SQL Server Reporting Services for reports.

**Future Considerations**

Microsoft has created a new implementation of .NET called .NET Core that will eventually replace the current .NET implementation. The .NET Core is a fully open source platform that runs on all architectures including Windows, Linux, and Mac. The Carity system has been implemented in alignment with .NET Core, simplifying a future migration to .NET Core. Microsoft’s .NET Core Compatibility tool has shown that the Carity platform is currently 98 percent compatible with .NET Core.

### Architectural Strategies

The Carity system is based on the software development principles of Domain Driven Design[[1]](#footnote-2) (DDD). It is organized into four separate software layers, in addition to a transactional database.



The *user interface* is developed as an SPA[[2]](#footnote-3). An SPA renders the user interface directly in the browser. This dramatically changes the traditional computing model, since a web server that renders web pages is no longer needed. Rather, the server simply hosts static user interface templates as well as a host of web services. Carity has been developed using the popular Angular SPA architecture, which is programmed using Microsoft’s TypeScript language. TypeScript is compiled into JavaScript and runs in the browser.

Carity’s *web services layer* is implemented in the Microsoft .NET platform using the Web API web services framework. It is programmed with the C# programming language. The Web API framework implements HTTP services, sometimes referred to as RESTful services[[3]](#footnote-4). RESTful services are commonly implemented when building an SPA user interface, since the JavaScript running in the browser can directly interact them. The web services layer is responsible for enforcing access control, controlling database transactions, and acting as a mediator between the user interface and the domain. Data flowing from the domain to the user interface needs to be mapped to user interface models, and actions in the user interface need to be translated to business operations in the domain.

The *domain model* is a software model intended to represent the real world domain. Following traditional object oriented software development principles and the DDD design patterns, the domain model contains business objects representing the objects found in the real domain. Objects such as Organization, Person, and Person-Centered Plan contain the information, functions, and rules that govern their behavior. The domain is “persistence ignorant” in that it is unaware of the existence of the transactional database. Rather, the mapping of domain objects to a database is handled by the infrastructure layer seen in the above diagram.

Carity is a Test Driven Development [[4]](#footnote-5) (TDD) system, meaning that software “unit” tests are written to test the code prior to actually writing the code. Since the test is written first, it essentially drives the development process. Test-driven development is a software development best practice that results in a higher quality system that tends to be resilient to software changes.

### Performance Engineering

The initial impression that users of the Carity system experience will be based on the look and feel of the user interface. The long-term impression will be defined by a sense that the system is responsive and allows them to do their jobs without long delays waiting for the system to complete a task. Developing a responsive system requires different design strategies for different types of screens.

*Search screens* in Carity allow the user to query for a group of database records. Careful planning has gone into defining proper database indexing in order to avoid table scans. Carity uses Microsoft SQL Server Table-Valued Functions (TVF)[[5]](#footnote-6) to perform these searches. TVFs are programmed directly in the database and provide a database engineer fine control over the query. Furthermore, searches employ a paging strategy so that only a segment of the data are returned to the user at a time. The user can choose to “page” through the data if they want to see more records. Using these strategies, Carity searches are highly performant and give the user a sense of instantaneous response.

*Edit screens* are the most complex types of screens in Carity. Edit screens often contain large amounts of data and long lists of lookup data. Complex screens like this are the most common cause of slow system load times. Loading a Person record is a common user task and needs to consistently perform well. Carity achieves fast load times by reducing the number of outbound requests to the server. A “chatty” screen makes dozens of requests to the server in order to load all necessary data. Carity loads the entire record as well as lookup data in a single server request. One exception to this is lookup data that contain long lists. Certain lookups, such as Human Languages that can contain hundreds of values, are not initially loaded into the page. Rather, they are loaded dynamically as the user is editing the field.

Small edit screens can sometimes have a single save operation. Large edit screens may have dozens of sections with a large number of interactions. These individual interactions are the primary means with which the user experiences the system. Actions such as adding a phone number to a Person record or submitting a record as part of a form workflow are the interactions that enable a user to do their job. The user should not have to wait for the system to respond. These interactions should feel instantaneous.

Carity employs several techniques to enable all such tasks to be responsive. Keeping tasks targeted and small, such as adding a phone number to a Person record, allows the web service to begin a database transaction, execute a single business operation, and commit the transaction. These types of web services usually respond in less than 100 milliseconds (usually far less time), giving the user the feel of an instantaneous response. Tasks that are long-lived, such as batch operations, are executed in the background so that the user does not have to wait for them to complete.

Regardless of the careful planning that goes into building a responsive system, the system will inevitably slow down as the number of users grows. If the user load is causing a system slowdown, Carity can easily be scaled up by simply adding additional application servers in a load balanced environment.

## System Architecture and Architecture Design

This section outlines the system and hardware architecture design of the system.

### Logical View

Figure 1 - Logical View illustrates a high-level overview of the Carity System.



Figure 1 - Logical View

**Browser–**Carity is a browser-based web application implemented as an SPA using Google’s Angular technology.

**Authentication Server–**Carity uses the Open Source Thinktecture Identity Server[[6]](#footnote-7) for user authentication. Thinktecture implements the OpenID Connect and OAuth 2.0 protocols. The authentication server has its own separate system account database.

**Application Server–**The web services that implement all Carity functionality are hosted in the Carity Application Server. The application server can be thought of as the core of the Carity system with all other components serving a supporting role.

**Transactional Database–**The transactional database is the primary component used to store data in the Carity system. With the exception of attachment files and audit information, all data can be found in the transactional database.

**File Attachment Server–**Carity supports the ability for the user to attach files in various parts of the system. Files tend to be large and are more efficiently stored in a server designed for file storage. Carity uses the Minio Private Cloud Storage server[[7]](#footnote-8) for all file storage.

**Scheduled Process Server–**Carity depends on certain periodic tasks or batch operations to be performed in the background. This is achieved with use of a server designed for scheduling tasks. Carity uses the JAMS Job Scheduling Server[[8]](#footnote-9) for all background processing. The JAMS server has its own Scheduled Process Database used for recording the list of scheduled tasks.

**SQL Server Reporting Services–**Carity uses Microsoft’s SQL Server Reporting Services (SSRS) for all canned and ad hoc reports. The reporting database is a replica of the transactional database.

**Auditing–**Carity records all requests to the application server web services in an audit system. The audit records are first recorded in an Audit Staging Database and are later moved to an Audit Data Warehouse by SQL Server Integration Services (SSIS).

### Hardware Architecture

Figure 2 - Physical Model provides a high level overview of the physical layout of the deployed Carity System.



Figure 2 - Physical Model

Carity is deployed across two separate networks, a perimeter network or demilitarized zone (DMZ), and an internal trusted network. The application server is deployed to the trusted network. The DMZ protects the application server and support services by limiting incoming traffic to a specialized reverse proxy server.

The reverse proxy server restricts unauthenticated traffic from entering the system. Only requests from users who have been authenticated are forwarded to the application server in the trusted network.

Both the reverse proxy server and the application server are load balanced and are capable of scaling up to serve the needs of increased traffic demands.

#### Performance Hardware Architecture

##### Load Balanced Web Applications

All web layers of the application are load balanced via a pair of highly available physical load balancers. We employ two Kemp Load Master Appliances in an active\passive configuration to balance load across web applications. These two appliances are bound to addresses within the virtual networks where they are servicing traffic. A virtual IP address is assigned to each load balanced service within the network for which it is servicing traffic to real servers.

Application services within the solution that are serviced in this way are listed below. The listed services will generally have three or more nodes depending on system demand:

* Reverse Proxy Server
* File Attachment Server API
* Authentication Server
* Application Server

##### Clustered File Attachment Server

While the file attachment servers APIs are load balanced as noted in the previous section, the underlying services that present the storage to the APIs are clustered as well. Each node has two allocated storage spaces with a total of six nodes.

The file attachment server manages it storage redundancy using erasure code. This shards data and parity across all six nodes. There are N/2 data disks and N/2 parity disks. With a six node configuration each node has both a data disk and a parity disk. In this configuration you can lose up to six individual disks or three full nodes before data loss.

##### SQL Always-On Availability Groups with Windows Server Failover Clustering

Data within the application is stored using Microsoft SQL Server. All databases are assigned to and take advantage of the Always-On Availability Group functionality provided by SQL Server Enterprise. All application end points communicate with their databases through Availability Group listeners which are actively managed by Windows Server Failover Clustering. The minimum configuration for the application utilizes three nodes per Failover Cluster with availability groups spanning all three nodes. Data integrity is ensured with at least two stand by replicas using synchronous commit. Further redundancy may be added by including more stand by replicas in asynchronous commit mode.

Application components with databases utilizing Always-On availability groups are listed below:

* Authentication Database
* Transaction Database
* Reporting Database
* Audit Database
* Scheduled Process Database

##### Scheduled Process Server Active\Passive Cluster

The Scheduled Process Server uses an active\passive failover configuration. The application endpoints interact with its APIs via a virtual IP that is managed by the Scheduled Process Server. Note that these servers are not load balanced, merely redundant.

##### Physical Infrastructure Redundancy

All application services are hosted on virtual machines using Microsoft Hyper-V managed by Microsoft System Center Virtual Machine Manager. Our physical production hypervisor infrastructure is split into three clusters. Each layer of the application is resident on at least three virtual machines or more. This allows us to spread the virtual machines across all three hypervisor clusters (at least one on each). This adds a layer of physical redundancy, based on our bare metal hosts separation.

### Software Architecture

The Carity system primarily consists of a browser-based user interface and a backend application server. There are several support services used to assist in the operations of the system. These services are either commercial off-the-shelf (COTS) products or open source applications which have their own documentation and will not be addressed in this SDD artifact.

#### High-level Architecture

Carity’s high-level software architecture is based on the principles of the Domain Driven Design[[9]](#footnote-10) (DDD) set of software design patterns. A DDD system is divided into four software layers as shown in Figure 3 - Domain Driven Design Layers below.



Figure 3 - Domain Driven Design Layers

The *Domain Layer,* sometimes referred to as the business layer, contains software objects that represent their real world domain counterparts. For example, the Carity system has objects in the domain that represent Organizations, Locations, Persons, and so on. The domain contains all of the systems business logic including business rules, events, and workflows.

The *User Interface Layer* is the part of the system that users can see and with which they can interact with Carity. There are screens in the user interface that allow users to search for, create and modify information in Carity. The Carity user interface is implemented as a web application that runs in the browser.

The *Application Layer* of a DDD system acts as a mediator between the user interface and the domain. In the Carity system, the application layer is implemented as a set of web services that serve the needs of the user interface. Access control functions, database transactions, and translation between the objects in the domain and the objects in the user interface take place in the application layer.

The *Infrastructure Layer* contains all of the logic that maps the object oriented domain layer to the Microsoft SQL Server database tables. This Object Relational Mapping (ORM) functionality is provided by the open source NHibernate library.



Figure 4 - Application Tiers

Figure 4 - Application Tiers shows how the layers of the system are distributed. The application server houses the web services that call upon the domain layer to make changes to business objects which are saved to the database by the infrastructure.



Figure 5 - Application Components

The Carity source code is segregated into application components (another form of layering) that groups the system into its platform code, core code, and customer-level code. Each component contains code for the front end (Angular Code), the web services layer (WebAPI Code), the domain logic (Domain Code), and the database (Database Code).

As shown in Figure 5 - Application Components, the platform sits on the bottom and forms the foundation of the system. Fundamental services allow the system to operate, or in the case of the user interface that forms the basic UI shell, are implemented in the Platform. A good analogy is to think of the Platform as the foundation and framing that one would build when constructing a house. The Core contains the main features of the system that users interact with on a daily basis. Features such as the Person Centered Plan, Attachments, or Progress Notes are implemented in the Core. Finally, the top layer component is the customer component. This layer serves two purposes. First, it provides a customization layer for changes business rules or metadata in order to change the look or the behavior of core components. Second, it provides a layer to implement features that are unique to a given customer.

#### User Interface

The Carity user interface is implemented with Google’s Angular[[10]](#footnote-11) SPA architecture. SPA technology is a paradigm shift from traditional web application development. In the past, web browsers would send a request to a web server and receive a full HTML page as a response. The browser would then discard the previous page and display the new page. The user would see the page very briefly flash a white background as the browser was preparing to render the new page. The brief flash or blink was disruptive. Not only did it break users’ concentration, but also caused the browser to lose the placement of the cursor and the scroll position of the page.

With an SPA, a single HTML page is loaded into the browser when the user first visits the site. That single page is then dynamically updated to render different content as the user interacts with it. The HTML is no longer generated by a web server. Rather, it is generated and rendered directly in the browser. This approach gives the user interface a more natural and native feel that is less disruptive to the end user.

Carity has a highly flexible and configurable design. Individual screens are rendered with the assistance of a metadata architecture. *Metadata* contains information about various user interface components such as labels, field sizes, and whether a field is read-only or required. The metadata can be changed for different implementations of Carity allowing customers to relabel fields, change field sizes, and so on. The metadata architecture is documented in section 4.3.4 Metadata.

Rendering web content in the browser requires the presence of JavaScript code. JavaScript is the only scripting language that has been able to take hold in modern web browsers. Unfortunately, JavaScript has not type system. When working with large code bases, development teams have continually arrived at the conclusion that type systems make a code system more understandable and improve code quality.

*TypeScript* is a light-weight wrapper over JavaScript that introduces a type system to JavaScript. When TypeScript is compiled, it is converted into JavaScript where it is able to run in the browser. TypeScript was created by Microsoft and has been adopted by Google. Angular is implemented in TypeScript and all Angular code written in Carity is implemented in TypeScript.

#### Application Server

As indicated in Figure 4 - Application Tiers, the application server houses the web services, domain and infrastructure layers. These layers implemented in Microsoft’s Open Source .NET technology and coded with the C# programming language. At the time of writing, Carity is implemented in Version 4.7.2 of .NET; however, the software has been structured to be in line with Microsoft’s .NET Core platform. The .NET Core is a cross platform, fully open source, implementation of .NET. It will run on Windows, Linux, or Mac OS.

##### Web Services

Web services in Carity are implemented with Microsoft’s WebAPI framework for HTTP Services (sometimes referred to as RESTful Services).



Figure 6 - Web Service Pipeline

All web service requests that are received by the application server are processed by a pipeline of handlers prior to being routed to the actual web service. As shown in Figure 6 - Web Service Pipeline, there are four areas of control that are handled by the pipeline. An *Audit Handler* records parts of the incoming request and the outgoing response to the Carity audit system. A *Security Handler* protects all web service endpoints. No requests are allowed to access individual web services unless they are authorized. A *Database Transaction Handler* guarantees that successful requests are properly committed to the database and unsuccessful ones are discarded. The *Metadata Management* handler controls the flow of metadata to the browser in order to make sure the UI is rendered properly for the customer.

In general, web service end points are either *performing an action* or they are *retrieving information* from the database. Most actions are intended to execute a business operation in the domain such as ‘Submit to Supervisor’, ‘Create a Person’, ‘Update Phone Number’, and so on. All such business logic resides in the domain. The web service simply acts as a mediator between the requesting client and the domain. When the web service receives such a request, it delegates the responsibility of executing the request to the domain.

The domains’ structure in focused on managing business operations, rules, and logic. It is knows nothing about the user interface and even assumes that there may be more than one user interface. Therefore, it is necessary to have a different object model that is shaped for the needs of the user interface. The web services layer maintains a set of *view models*, sometimes referred to as *data transfer objects* (or DTOs). These objects are structured for the needs of the UI and are often shaped differently than their counterparts in the domain. It is the responsibility of the web service to map the information from the domain objects to the view models when a response is returned to the client.

Web service requests that retrieve information from the system typically require a database query to be performed. Database queries can be fraught with problems. One needs to be concerned with SQL injection attacks and open-ended SQL queries. *SQL injection attacks* are intended to trick the server into appending dangerous statements to a SQL query. *Open-ended queries* are SQL Select statements that have no limit to the number of records they could return.

Carity employs a custom query framework that uses query parameters rather than constructing SQL dynamically. This technique negates the possibility of a SQL injection attack. Furthermore, this framework forces all queries to implement paging. By returning data a page at a time, it is impossible to have an open-ended query.

The data returned from such queries is mapped to view models and returned to the client in a similar manner to the way that domain objects are mapped.

##### Metadata

Metadata, defined as ‘a set of data that describes and gives information about other data’, are used to describe the view models returned by the web services layer.



Figure 7 - Metadata Example

Figure 7 – Metadata Example above illustrates that the first name of the person has a label equal to “First”, a text length of 100 characters, and it is a required field. The same is true for the other properties.

In Carity, metadata such as this is used to drive the rendering of the user interface. Each web service is designed to serve specific user interface components. The view models returned by the service are shaped to meet the needs of that particular UI, and the metadata are tied to those models. This allows granular control over the shape of the user interface.

The metadata framework has a growing list of metadata types. The following are examples of the most common metadata types in current use:

* Description: Used for user interface tool tips
* Disabled: Indicates that the property is disabled and shouldn’t allow the user to interact
* Display Name: Used for property labels
* Display Order: Used to indicate how this property should be sorted on the screen with other properties
* Hidden: Indicates that this property should be hidden from view
* Lookup: Informs the user interface how to retrieve lookup values that are displayed in drop down lists
* Range: Indicates the range of numbers supported in a numeric property (e.g. 0…100)
* Read-Only: Indicates that this property may not be edited
* Required: Indicates that the property is required in order to save the record
* String Length: Indicates the maximum number of characters the user may enter into a text property

The Customer component layer, described in section 4.3.1, provides the ability for customers to customize the metadata for all screens. Customizing the metadata enables customers to change screen labels, or hide fields that they do not need.

##### Domain

Carity is a DDD system. The domain is a software model designed to reflect their real world counterparts. The domain is organized into modules, which are simply groupings of similarly related objects. The business objects, referred to as entities in DDD terms, contain data properties as well as business operations that implement the business logic. Figure 8 - Organization Domain Model shows a typical example of a model. In this example, it is a model of the Organization in the Carity system.

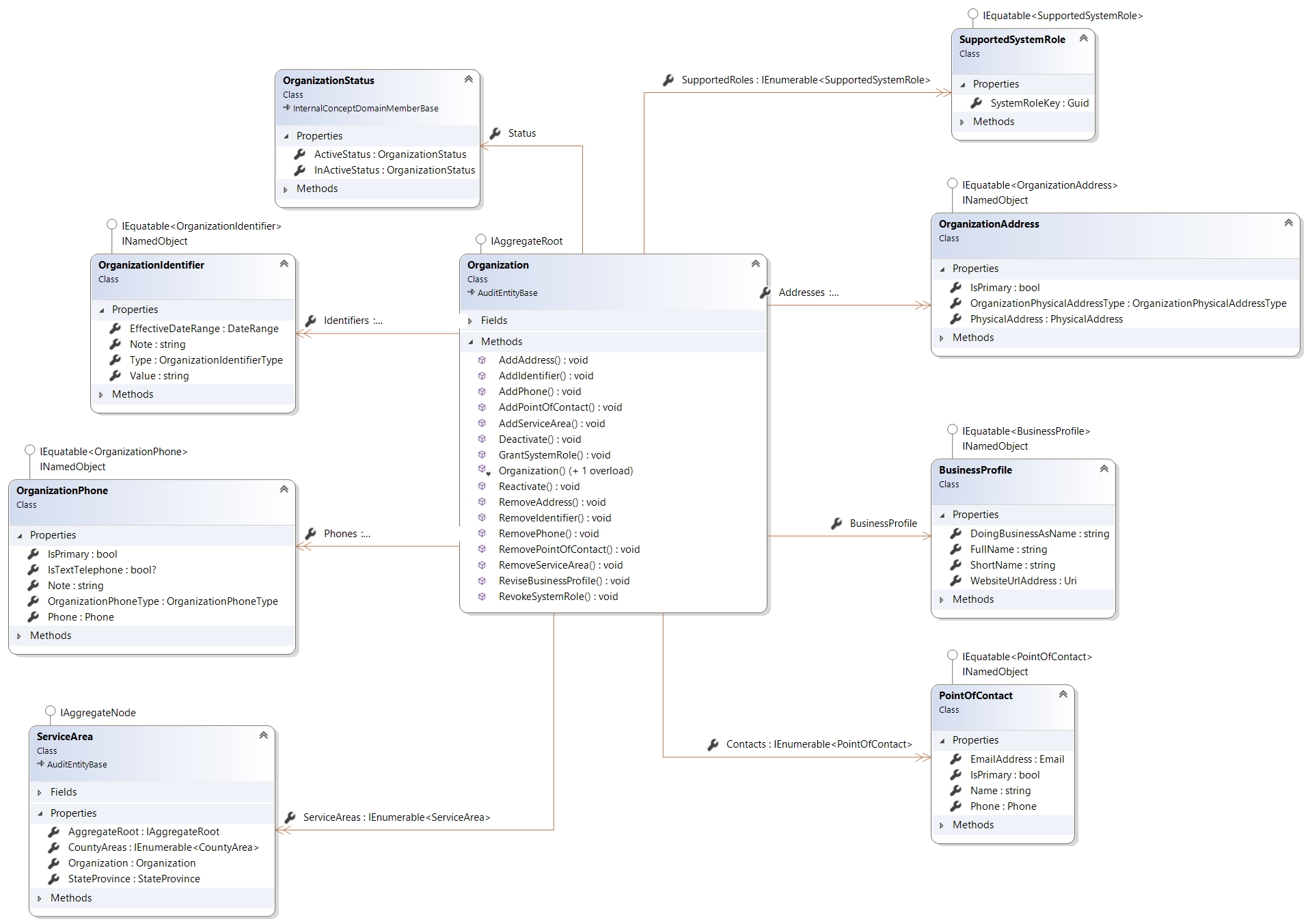


Figure 8 - Organization Domain Model

Models contain properties like the ShortName property in the BusinessProfile and the Note property in the OrganizationIdentifier objects. They also contain business operations such as the AddAddress and Deactivate operations in the Organization object.

Business operations are the primary means with which business logic is implemented in a DDD system. Operations may make changes to the properties in the objects but the entire domain layer is *persistence ignorant*, meaning that it doesn’t know that the database exists and therefore has no idea how to save the property changes to the database. Saving these changes to the database is the responsibility of the infrastructure layer.

Business rules are applied to business operations in the form of rule sets. There is a rule set for every operation. The above AddAddress operation has an AddAddressRuleSet counterpart. The customer component layer has support for changing the rules in a rule set. This gives customers the ability to configure the system with their own business rules.

##### Infrastructure

The Infrastructure layer houses the mapping logic to translate objects from the domain to the database tables. Carity uses the NHibernate Open Source object-relational mapping library along with the Fluent NHibernate Automapping framework. Fluent NHibernate provides an additional capability that provides for convention based mapping. This means that conventions can be defined which allow objects to be mapped without any additional code (they are mapped by convention). This style of mapping provides a very consistent mapping approach and enables predictability. When you understand the domain, you will inherently understand the database.

#### Database

Carity uses the Microsoft SQL Server suite of servers for all database processing. That includes the SQL Server Database Server, SQL Server Reporting Services (SSRS), and SQL Server Integration Services (SSIS).

Large systems can be difficult to navigate. As the number of files and objects increases, organizing those objects becomes increasingly important. Carity has been designed with the intention to enable developers and administrators to intuitively understand its design and be able to easily find components. To that end, the database is designed along-side the domain. The domain is organized into modules. Likewise, the database is organized into schemas.

For example, the domain module shown in Figure 8 - Organization Domain Model is mapped to the database tables shown in Figure 9 - Organization Database Model. The database schema has the same name as the namespace of the domain code. The database tables are derived from the object names in the domain. The column names in those tables are derived from the property names in the objects. This design makes it easy for the entire development team as well as the operations team to intuitively understand the design of the system.

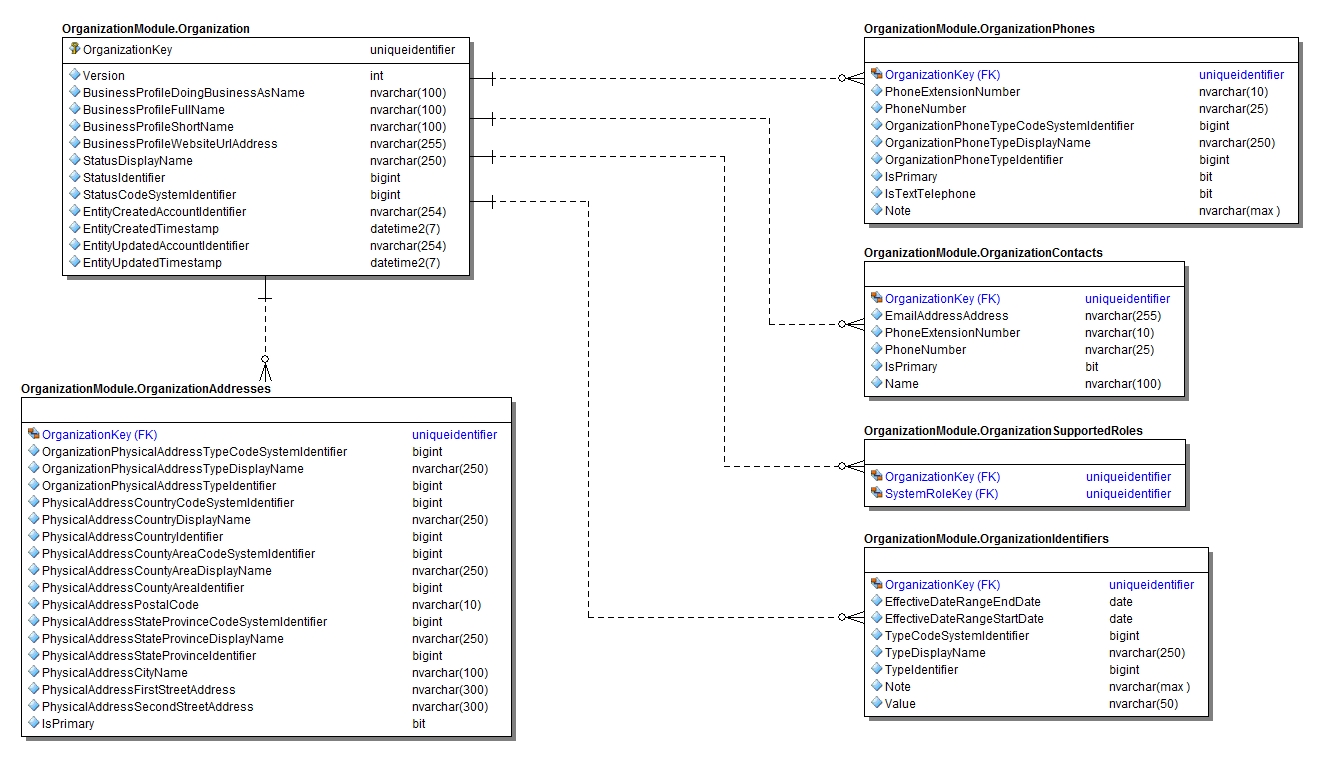


Figure 9 - Organization Database Model

#### Security Software Architecture

Carity’s security design consists of an authentication, authorization, and access control (AAA) system which control access to system services, along with an auditing system to track which services and data that a user has accessed.

The authentication of users is implemented with the use of the Thinktecture[[11]](#footnote-12) open source authentication server (sometimes referred to as the security server, the security token server, or the identity server). The authentication server implements the OpenID Connect protocol, which is an extension of the OAuth 2.0 protocol. This security server provides Carity with a single sign on (SSO) capability. Users are given access to the system by granting user roles. Roles can be granted to users in three ways, as shown below.

1. System Account–Roles may be granted directly to system accounts. Roles granted to system accounts are applied to a system session regardless of which Organization or Location a user is accessing.
2. Organization Access–Carity is a multi-tenant application which is segregated into Organizations and Locations. Users always operate within the context of an organization/location combination. Users are given access to organizations through an Organization Access record. Roles may be granted to an Organization Access record. Roles granted to organization access records are applied to a system session when the user accesses the organization.
3. Location Assignment–Most users of Carity are staff members that work for an Organization. A staff member record is automatically assigned an Organization Access record when the staff members’ account is created. Staff members are typically assigned to work for one or more locations. In such cases, a Staff Member Location Assignment is created. These users are only allowed to access the locations to which they have been assigned. Roles may be granted to Staff Member Location Assignments. Roles granted to Staff Member Location Assignments are applied to a system session when the user accesses the location.

System roles may be one of three possible types shown below:

1. Task–Roles that are tasks will always have a *verb or verb phrase* name defining a precise action a user may perform. Examples of tasks are: ‘Update Staff Member Address, and ‘View Person Note Comments’.
2. Task Group–Roles that are task groups, like Tasks, will have a *verb or verb phrase* name. Task Groups are a combination of other tasks and even other task groups that collectively define an action a user may perform. Examples of tasks are: ‘Update Staff Member’, and ‘Perform Intake Operation’.
3. Job Function–Roles that are Job Functions will always have a *noun* name defining an actual. Job Functions are a combination of tasks and task groups that collectively define the types of users that access the system. Examples of Job Functions are: ‘Case Worker’, and ‘System Administrator’.

System roles which are tasks are granted a set of Permissions. Permissions are low-level objects that are not seen by end users and are directly tied to the systems source code. Every web service defines a permission that a user must have in order to access it. If a user attempts to access a web service without the proper permission then the service will respond with the *HTTP Code 403 – Forbidden*. Web services may also have business rules that require a user to have a particular permission in order to access certain data. These permissions permit a user to access web services and data. Collectively these permissions define the *Access Control* portion of the security infrastructure.

### Information Architecture

Information stored within Evergreen includes person, provider, services, and claims data. Data within evergreen will span PII, PHI, and individually identifiable information (IIF).

#### Records Management

Federal regulations issued by the National Archives and Records Administration (NARA) are outlined in 36 Code of Federal Regulations (CFR) - Subchapter B - Records Management. Business owners must contact the Office of Strategic Operations and Regulatory Affairs (OSORA) to initiate the record management process.

##### Data

Data supplied to the system is in electronic form only. APS related data is supplied to Evergreen via electronic file transfer from legacy MAPSIS. Person, provider, and claims data is supplied to Evergreen via electronic file transfer from MIHMS (PUB files). Waitlist data is supplied to Evergreen via spreadsheet export (e.g., comma-delimited file format). One-time and recurring data inputs and outputs are accomplished through formats, paths, and frequencies determined jointly with Maine DHHS, including OADS.

##### Manual/Electronic Inputs

Inbound electronic inputs are loaded to a staging database. Subsequently, inbound data is loaded to the Evergreen database or to the Data Warehouse.

### Internal Communications Architecture

The entirety of the application stack is virtualized. All communication is done over virtual LAN interfaces provided by Microsoft Hyper-V and managed by Microsoft System Center Virtual Machine Manager. The main application stack is separated into two networks (DMZ and TRUST). The DMZ zone houses the proxy server and the authentication server. The TRUST zone houses the application server. There are several management networks that also exist which house various enterprise services that the application consumes.

All network boundary traversals are routed through physical firewall appliances. The firewalls enforce policies on network boundaries through a practice of implicit denial and explicit allowance. All firewall policies are vetted and approved by both the operations and security teams independently. All network traffic must be identified and assigned to already-approved application groups with listed source and destination for both addresses and ports.

Access to the application from the internet is achieved through assigning a publicly addressable IP to a NAT route. The route maps to the load balanced virtual IP for the reverse proxy layer of the application as well as the load balanced virtual IP for the authentication server. This traffic is also passed through the firewall, as this counts as an external network boundary. Only port 443 using HTTPS over TCP is allowed over this NAT route from the firewall.

All internal communication within the application networks is encrypted. All APIs are exposed only over HTTPS and all other connections are required to use TLS\SSL or some form of FIPS 140-2 encrypted traffic.



Figure 10 - Internal Communications Architecture

### Security Design and Architecture

The FEI infrastructure is designed and engineered to be compliant with the Health Insurance Portability and Accountability Act of 1996 (HIPAA). Additionally, the solution and hosting operations adhere to the NIST SP800-53r4 (Security and Privacy Controls for Federal Information Systems and Organizations) control framework for FIPS-199 Moderate systems. The system itself is web-based and built with the use of Microsoft .NET technologies on Windows IIS Web Servers. It relies on hosting an infrastructure with virtual servers running the Windows OS to host Web and Database Servers. Firewalls and VLANs separate the system into the following zones, which maps to FEI’s three-tiered architecture blueprint:

* Internet facing DMZ
* Application Services
* Database Services Zone

All network communications across zones are encrypted with TLS/SSL protocol and protected by next-generation firewalls (NGFW). Additionally, externally-facing web traffic is protected via Web Application Firewalls (WAF).

#### Identification, Authentication, and Authorization

Formal vetting and training are required before users receive access to any information system. After successful vetting and training, FEI personnel are provided with a unique UserID. Next, they can create a password. This will grant them access to information systems. Role-Based Access Controls (RBAC), in which each user can only access information assets necessary to perform their assigned function, are created at all access levels.

#### Data Encryption and Integrity

As a matter of policy, FEI requires that whenever cryptographic mechanisms are used, the information system implements cryptographic modules that comply with applicable federal laws, Executive Orders, directives, policies, regulations, standards, and guidance. Furthermore, as described in our Encryption Standard, FIPS 140-2 compliant and NIST-validated cryptography must be used when supporting Federal systems customers (and should be used in all other situations) to protect unclassified information.

Traffic that flows throughout the LTSS ME application is protected via SSL to ensure confidentiality. In addition to securing data in transit, all information is encrypted at the LUN to ensure confidentiality of information at rest.

#### Security Information and Event Management

All production systems generate logs of system access, O/S security event activity, firewall, intrusion detection, and related security point product transactions. This activity is collected, aggregated, and fed into our AlienVault SOC/SIEM where alerts are generated accordingly. All systems have computer operator logs that show production application start/stop times, system boot/restart times, system configuration changes, system errors and corrective actions taken, and confirmation that files and output were handled correctly. Furthermore, all computer systems (everything from single-user workstations to mainframes) running production application systems and/or accessing data belonging to or entrusted to FEI must automatically include the generation of logs. All logs must record, at a minimum, the following:

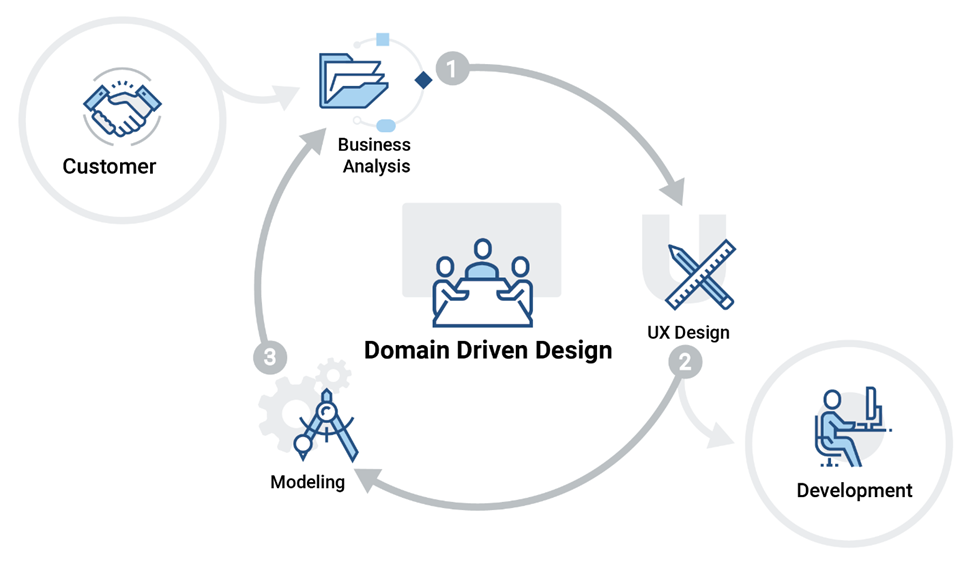
* User account management activities
* System shutdown
* System reboot
* System errors
* Application shutdown
* Application restart
* Application errors
* File creation
* File deletion
* File modification
* Failed and successful logons
* Security policy modifications
* Use of administrator privileges.

Additionally, for perimeter devices (including firewalls and routers), logging shall be enabled for the following:

* Log packet screening denials originating from un-trusted networks
* Packet screening denials originating from trusted networks
* User account management
* Modification of packet filters
* Application errors
* System shutdown and reboot
* System errors
* Modification of proxy services.

## System Design

FEI Systems’ DDD development process focuses on understanding the customers’ business domain through application of a Business Analysis process that ensures a complete understanding of the customers’ needs, a UX Design process to align the user interface design with the use cases and workflows of the systems users, and a Domain Modeling process that delivers a software model of the customers’ business domain.



**Figure 11 - Domain Driven Design Process**

### Business Analysis

The system is designed to meet the business requirements gathered with the client through the requirements elicitation process. Through multiple sessions with the client, we determine the current business process and therefore how to architect the system in order to accommodate this process and any applicable policies, procedures and regulations. Requirements gathered are housed in a Functional Requirements Document (FRD) which includes a Data Elements Table, Access Control Rules, Notifications to be Generated, Queries for Features, Business Rules, and Authorized Users. Once the FRDs are signed off by the customer, the system is built based on these FRDs. Any requests to changes in the functionality in the system that are different from what is stated and agreed upon in the FRD will be treated as Change Requests. Therefore, design of the system first and foremost incorporates the needs of the client and ensures:

1. Business process can be incorporated successfully into the system
2. Login users are given access to the proper persons in the system to enable coordination of their care
3. Login users are not provided access to see any persons they are not authorized to access as a function of their job responsibilities

### User Experience Design

The Carity system user interface design is guided by a User Experience (UX) Framework, a framework comprising of three key areas that drive and inform the design and development of quality user interfaces that are intuitive, usable, and user centered. The three key areas are described below.

#### 5.2.1 User Centered Design (UCD) Approach

Carity is designed from a user-centered design approach which always begins with research and discovery. Utilizing research and inquiry methodologies allows us to gain an understanding of the user needs that can inform good design. Our research focuses on learning about the user's goals and tasks, aspirations and motivations, behavior and sentiment, as well as context and challenges the user may encounter. The information gathered throughout our user-centered design approach provides us with the insights necessary to make sound decisions in both the design and development of the system

#### Lean UX

Our user experience team uses an iterative design approach called Lean UX. This design process is ideal for designing and building user interface solutions. Influenced by Agile product development principles, Lean UX is a cyclical design process that compliments and supports projects and development environments that use Scrum Agile and Scaled Agile for Enterprise (SAFe). Lean UX is comprised of the following three stages: *Think*, *Do*, and *Test*.

* **Think (Analysis)—**Relies on a domain driven design (DDD) approach that considers the business model such as organizational process, protocols, and data requirements. Sources of input that inform our UI design include use cases, domain models, and workflows.
* **Do (Design)—**This phase seeks to arrive at a user interface design solution that solves the user needs with consideration for the business and/or domain requirements. Typically, this design will take the form of wireframes, mockups, and/or prototypes.
* **Test (Evaluate)—**In the test phase, the design is published and shared with stakeholders for the purposes so that the design can be evaluated. The resulting feedback is documented and taken into consideration as part of the "Think" stage in the Lean UX Design cycle.

#### Usability and Accessibility

Our UX Framework uses best practices and methodologies to achieve the highest level of *usability* and *accessibility*.

* **Section 508 and Accessibility—**Our Accessibility framework provides guidance and a process for achieving conformance and certification of compliance with Section 508 of the American with Disabilities Act. Our Accessibility Framework, based on the WCAG 2.0 guidelines, allow us to ensure that our customer’s end-users can access and experience the products and services created without hindrance. Our Accessibility Framework is comprised of:
  + Section 508 Regulation
  + ADA Standards for Accessibility Design
  + WCAG 2.0
  + Accessibility Testing
  + Remediation Plan
  + Accessibility Certification
* **Usability—**The user interface characteristic found in software that make features and functionality useful and easy-to-use. Usability is accomplished through the application of methodologies that provide insight into how users behave, think and process information. When these methodologies are coupled with UI design best practices, such as establishing UI patterns and guidelines, usability is further supported through consistency and uniformity across the user interface. Our user interface solutions achieve usability when we leverage methodologies and best practices, including Style Guides and a UI Pattern Library.

### Domain Modeling

Domain Modeling involves the design and development of an object oriented software model that represents the real world business domain. DDD is an industry best practice that consists of a set of object oriented design patters that are grouped into two pattern families; strategic design patterns and tactical design patterns.

The strategic design patterns are higher level organizational patterns. The primary focus of the model is on the ‘core’ domain. The core domain is a part of the overall business domain that is of primary importance. Other parts of the domain can be grouped into sub-domains that exist to serve or support the core domain.

The core domain and the various sub-domains are further segmented into modules. Modules can be considered as groupings or folders that organize the domain entities that implement the business logic.

All of the significant “nouns” in the customers’ business domain are implemented as entities in the domain modules. Objects such as *Organization*, *Location*, *Person*, and *Person Centered Plan* define physical structure (e.g. fields), business operations, events, and business rules that implement business functionality.

For every feature that is created in the system an initial domain design is defined in a set of Visio diagrams that may contain UML Class, Sequence, or State Chart Diagrams that define to the development staff how the object should be written. The entity design is then aligned with the UX Design and the Business Analysis to make sure that no gaps exist and that all designs are in agreement.

## System Integrity Controls

A formal documented system and information integrity policy that addresses purpose, scope, roles, responsibilities, management commitment, coordination among organizational entities, and compliance has been developed and implemented for all systems maintained and operated by FEI Systems. This policy is updated at least annually or when new integrity-specific requirements are established and are approved by appropriate FEI leadership. Detailed integrity controls are provided in the following sections.

### Access to Critical Data

As referenced in security controls AC-02 and AC-05 of the LTSS ME System Security Plan (SSP) and FEI Master SSP, several layers of Access Control are implemented to ensure that access is only provided to personnel with job responsibilities that require access to such data. FEI has implemented well-designed Role-Based Access Controls (RBAC) such that each user can only access those information assets necessary to perform their assigned function.

In terms of information flow, we have deployed and maintain firewalls to restrict access between multiple zones (DMZs, and VLANS) on the network. These zones each have firewall policies that explicitly allow or deny access from one zone to another. Second, connectivity between FEI and our partners leverage IPSEC VPN tunneling with tunnel authentication algorithms, encryption algorithms and complex shared keys. Third, all email correspondences from FEI are evaluated with rules on routing triggered by various conditions. Similarly, users have been trained on marking sensitive emails appropriately before being sent.

It is FEI’s policy that whenever a computer-based process involves sensitive, valuable, or critical information, the system must include controls involving a separation or segregation of duties or other compensating control measures. Whenever practical, a single person should not be responsible for completing a task involving sensitive, valuable, or critical information from beginning to end. Likewise, a single person must not be responsible for approving his or her own work. To the extent possible, for every task involving sensitive, valuable, or critical information.

To this end, FEI has developed and implemented control measures to ensure that no one individual has exclusive control over these types of information assets. Specifically, FEI has prepared and maintains a definition of what it considers potentially incompatible duties. Furthermore, a segregation of duties analysis is completed and documented on no less than an annual basis, with requisite Separation of Duty Control Forms to be completed as necessary to indicate what actions are taken when a potential issue is identified.

### System Audit Controls

All systems have computer operator logs that show production application start/stop times, system boot/restart times, system configuration changes, system errors and corrective actions taken, and confirmation that files and output were handled correctly. Furthermore, all computer systems (everything from single-user workstations to mainframes) running production application systems and/or accessing data belonging to or entrusted to FEI must automatically include the generation of logs. In addition to the preceding policies, such logs must record, at a minimum, the following:

* User account management activities
* System shutdown
* System reboot
* System errors
* Application shutdown
* Application restart
* Application errors
* File creation
* File deletion
* File modification
* Failed and successful logons
* Security policy modifications
* Use of administrator privileges.

Additionally, for perimeter devices (including firewalls and routers), logging shall be enabled for the following:

* Log packet screening denials originating from untrusted networks
* Packet screening denials originating from trusted networks
* User account management
* Modification of packet filters
* Application errors
* System shutdown and reboot
* System errors
* Modification of proxy services.

FEI has allocated sufficient storage capacity to meet or exceed our audit log retention requirements. To begin, FEI’s host environment is a high availability GSS with redundant infrastructure capable of operating in active-active mode in two physically separated locations such that the likelihood of the failure anticipated by this control is extremely small. Furthermore, from a capacity standpoint, FEI’s has upwards of 500 TB of thin provisioned storage available should it be needed.

## External Interfaces

Interfaces have been developed between Maine’s MIHMS system and Evergreen for the regular import of provider, member, and claims information. An interface has also been developed between MAPSIS and Evergreen for a one-time import of existing Adult Protective Service (APS) Reports.

Utilizing a delimited file format, incremental member, provider, claims files, and full member and provider files will be sent through the flat file interface. Information from these files will be loaded into Evergreen staging tables and then into the application to ensure:

1. Members, providers and claims attached to members in Evergreen are created when no match exists in Evergreen
2. Members, providers, and claims are updated when new information comes through the files and those members/providers exist in the system
3. Therefore all member, provider and claims information remains in sync with the MIHMS system

For more detail about the specific interfaces employed in the system, please refer to Interface Analysis Document, and the Evergreen Data Migration and Integration Plan.

Appendix A: Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision History** | | | |
| **Project** | **State of Maine Evergreen Data System Project** | | |
| **Title** | **System Design Document** | | |
| **Contract Number** | **Advantage CT No. 10A-2018010000000002289**  **DHHS Agreement No. ADS-18-9451**  **Vendor Customer No. VS0000017510** | | |
| **Date** | **Version** | **Description** | **Author(s)** |
| 02/13/2019 | 1.0 | Submitted Initial Draft | FEI |
|  |  |  |  |
|  |  |  |  |

1. DDD is an architectural approach for tackling complex system. A high level summary can be found here: <https://en.wikipedia.org/wiki/Domain-driven_design>. The term was first used in the book *Domain Driven Design: Tackling Complexity in the Heart of Software*, by Eric Evans. [↑](#footnote-ref-2)
2. A Single Page Application (SPA) is a web application that “interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server” - <https://en.wikipedia.org/wiki/Single-page_application>. [↑](#footnote-ref-3)
3. Representational State Transfer, or REST, is a web services architectural approach that is usually implemented with the HTTP protocol. A high level summary may be found here: <https://en.wikipedia.org/wiki/Representational_state_transfer> [↑](#footnote-ref-4)
4. Kent Beck is credited with coining the term Test Driven Development, about which can read here: <https://en.wikipedia.org/wiki/Test-driven_development> [↑](#footnote-ref-5)
5. Learn about Table-Valued Functions here: <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/how-to-use-table-valued-user-defined-functions> [↑](#footnote-ref-6)
6. Thinktecture is a fully open source implementation of the OpenID Connect protocol. Details can be found here: https://identityserver.io/ [↑](#footnote-ref-7)
7. The Minio Private Cloud Storage server is an open source cloud storage server that supports the full array of Amazon Web Services S3 (Simple Storage Server) web services. Learn more about Minio here: <https://www.minio.io/> [↑](#footnote-ref-8)
8. The JAMS Job Scheduling server provides the ability to automate, monitor, and manage background tasks. Learn more about it here: <http://www.jamsscheduler.com/job-scheduling/> [↑](#footnote-ref-9)
9. Domain Driven Design is a well-established industry best practice for software design. Textbooks, training seminars, as well as online videos, blogs, and articles are easily found that can teach the principles of DDD. The preeminent text on the subject is “Domain Driven Design; Tackling Complexity in the Heart of Software” by Eric Evans. [↑](#footnote-ref-10)
10. Review information about Angular here: <https://angular.io/> [↑](#footnote-ref-11)
11. The Thinktecture identity server can be found here: <https://www.thinktecture.com/identityserver> [↑](#footnote-ref-12)