

eHealth Framework

Reference Documentation

Imprint

InterComponentWare AG
Altrottstraße 31
69190 Walldorf
Tel.: +49 (0) 6227 385 0
Fax.: +49 (0) 6227 385 199

© Copyright 2006-2010 InterComponentWare AG. All rights reserved.

Document ID: a0caa087-3c67-2d10-e6ab-f01a6124ecfa
Document version: 1.0
Document Language: en (US)
Security Level: Public
Document Status: approved
Product Name: ICW eHealth Framework
Product Version: 2.10.0
Last Change: 11.11.2010

Notice

Disclaimer

The *eHealth Framework Reference Documentation* is in an development state. But it covers almost all important topics of the eHF including the 2.10 release. Some chapters are currently either in a review phase or subject of development.

If you have comments or requests, please contact us using the support contact address: support@intercomponentware.com.

This documentation and the software components are protected by copyright© 2006-2010 InterComponentWare AG.

The wording in this document applies equally to women and men. The masculine form was selected to ease the comprehensibility and legibility of the text.

All company logos are a registered trademark of InterComponentWare AG.

The product names mentioned in this documentation are either trademarks or registered trademarks of the respective owners and are stated for identification purposes only.

The document is collaboratively built with the use of the Darwin-Information-Typing-Architecture (DITA) and has therefore a draft status concerning styles and layout. The necessary adaptations are currently also in a developmental stage.

All rights reserved.

Contents

Part I - Fundamentals.....	1
1 Introduction.....	2
1.1 Target Audience.....	2
1.2 Document Structure.....	2
2 eHealth Framework Overview.....	4
2.1 Metaphors.....	4
2.2 eHealth Framework Elements.....	4
3 Central Concepts.....	8
3.1 Techniques and Methodologies.....	8
3.2 Health Care Domain.....	14
3.3 Software and Information Life Cycle.....	20
Part II - Architecture.....	24
4 Conceptual Architecture.....	25
4.1 Primary Objectives.....	25
4.2 Modularization.....	26
4.3 Componentization.....	27
4.4 Module and Component Meta Model.....	28
4.5 Application Meta Model.....	30
4.6 Architectural Layers.....	31
4.7 Integration.....	32
5 Architecture and Life Cycle Realization.....	34
5.1 Platform Architecture	34
5.2 Standard Service Module Architecture.....	35
5.3 Event processing.....	40
5.4 Software Life Cycle Support	42
5.5 Information Life Cycle Support	43
6 Conventions.....	53
6.1 Naming Schemes.....	53
6.2 Logging.....	59
6.3 Testing Conventions.....	61
Part III - Development Environment.....	65
7 ICW Eclipse Development Environment.....	66
8 Build Plug-in.....	70
8.1 Module Build.....	70
8.2 Assembly Build.....	71
8.3 Property and Fragment Tokens.....	73
8.4 Database Management.....	73
9 Customization Plug-in.....	75
9.1 Defining a Customization Project.....	75
9.2 Defining the Targets for Customization.....	75
9.3 Defining Customization Rules.....	76
9.4 Applying Customizations.....	78

9.5 Extensions.....	78
10 Installation Ant Scripts.....	81
10.1 Prerequisites.....	81
10.2 Script Targets.....	81
11 Continuous Integration.....	86
11.1 Build Server.....	86
11.2 Deploy Server.....	86
11.3 eHF Routines.....	86
11.4 Continuous Integration Infrastructure in Action.....	87
12 Generator.....	88
12.1 Characteristics of the Generator.....	88
12.2 Configuration of the Generator.....	91
12.3 Extension Points of the generated Artifacts.....	93
13 Life Cycle Support.....	94
13.1 Web Service Backwards Compatibility.....	94
13.2 Database Upgrade.....	96
Part IV - Application Platform.....	107
14 Core Modules.....	108
14.1 Core.....	108
14.2 Commons Library.....	115
14.3 Commons Expert Entries Library.....	143
14.4 Commons Camel.....	144
14.5 Commons Security Library.....	146
15 Security Modules.....	151
15.1 Audit.....	151
15.2 Authentication.....	153
15.3 Authorization.....	162
15.4 Certificate Validation	185
15.5 Content Scanner.....	186
15.6 Security Token Service.....	195
15.7 User Management.....	210
15.8 Encryption.....	216
15.9 Pseudonymization.....	221
15.10 Commons Identity.....	225
15.11 LDAP Identity.....	228
16 Infrastructure Modules.....	235
16.1 Terminology	235
16.2 eHF Configuration	267
17 Application Modules.....	275
17.1 Record Facade.....	275
17.2 Record Medical.....	276
17.3 Record Admin.....	284
17.4 Document.....	291
17.5 Composition.....	294
Appendix.....	303
A UML Profile for eHF Generator.....	303

A.1 Stereotype ehf-domainobject.....	303
A.2 Stereotype ehf-attribute.....	307
A.3 Stereotype ehf-association.....	315
A.4 Stereotype ehf-operation	318
A.5 Stereotype ehf-service	320
B eHF Profile Versions.....	321
B.1 Updating the eHF Profile.....	321
B.2 eHF Profile Version 1.0.....	322
B.3 eHF Profile Version 1.1.....	322
B.4 eHF Profile Version 1.2.....	322
B.5 eHF Profile Version 1.3.....	323
C Versioned Template Reference.....	325
C.1 DAO Template 2.0.....	325
C.2 Web Service Template 2.0.....	325
C.3 Web Service Template 2.1.....	326
C.4 Auditable Service Template 2.0.....	327
C.5 Persistence Aggregate Template 2.0.....	327
C.6 Security Annotations Template 2.7.....	327
C.7 Input Validation Template 2.9.....	328
C.8 Persistence Association Template 2.10.....	329
C.9 Persistence Inheritance Annotations Template 2.9.....	330
C.10 Persistence Join Table Template 2.9.....	331
C.11 Persistence Holder Template 1.1.....	331
D OCL Reference.....	332
E Technologies.....	332
E.1 Spring Framework.....	332
E.2 Hibernate.....	340
E.3 Java Authentication And Authorization Service.....	341
F Modularization Addendum.....	341
F.1 Further Advantages of Modularization.....	342
F.2 Related Topics and Concepts.....	342
G Build Plug-in Configuration.....	342
H Installation Script Properties.....	350
I AOP Explained.....	351
J MDSD Explained.....	351
K Fragment Tokens.....	352
K.1 Module-Level Fragment Tokens.....	353
K.2 Assembly-Level Fragment Tokens.....	354
L Database Privileges Explained.....	358
L.1 Creating Database Privileges using the Ant Installation Process.....	359
L.2 Creating Database Privileges using SQL Scripts.....	359
M Create ATNA Compliant Audit Messages.....	360

Part I - Fundamentals

1 Introduction

The world-wide demand for health care has been and will be increasing steadily. So is the demand for electronic health care solutions. Medical data are recorded, processed, and interchanged electronically. Software-aided solutions complement and supersede traditional paperwork approaches. Stakeholders' ever-increasing demands and expectations lead to more sophisticated and complex software systems, while at the same time consumers and patients raise their voices and call for privacy and data protection.

It is in this area that the eHealth Framework (eHF) attempts to position itself as the leading platform for developing electronic health care solutions. It empowers software developers to build state-of-the-art eHealth applications by providing reusable software components, development tools, and architectural guidelines and conventions, covering the full software-development and product life cycle.

1.1 Target Audience

This document addresses software developers who want to work with the eHealth Framework (eHF). It will help you create applications that make use of the eHF, or build modules that are based on the eHF. Since the eHF platform is in large parts built with eHF itself, the audience includes developers who work on eHF platform modules themselves. Especially the later, technical chapters of this document provide valuable reference information for this group of readers.

Architects of eHF-based applications will also find this document helpful in their decision-making process. However, they may want to focus their attention on the first chapters, which explain the overall motivation and general concepts of the eHealth Framework.

1.2 Document Structure

The *eHealth Framework Reference Documentation* is divided into 4 parts. All readers can profit from the initial sections of [Part I - Fundamentals](#) on page 1 and [Part II - Architecture](#) on page 24. Developers or customizers of the framework will be particularly interested in [Part III - Development Environment](#) on page 65 and [Part IV - Application Platform](#) on page 107 where usage of eHF is described. Although you don't necessarily need to read the book sequentially you of course can.

Following this Introduction section, the sections have been designed to lead successively from the fundamental concepts, architectural considerations plus their implementation, that underpin the eHealth Framework (eHF), through to descriptions of the various types of modules that make up the framework.

eHealth Framework Overview

Here you are introduced to the main elements of the framework, namely the development environment and the solution platform. It describes the tools, frameworks, and documentation that assist you throughout the entire software development life cycle.

Central Concepts

In this section central concepts underpinning the design of eHF are listed and explained with a particular focus on where they have been applied to the eHF.

Conceptual Architecture

In this section the architectural principles and concepts are detailed. The main purpose is to describe the architecture on an abstract level and to not include technology and implementation details.

Architecture and Life Cycle Realization

This section shows how the conceptual architecture has in fact been realized. This section complements the conceptual architecture described in section four with details on the realization of such an architecture and provides details down to the implementation level. Furthermore, this section explains how it manages the software life cycle.

Conventions

This section looks at conventions used in naming artifacts, exception and error handling along with testing conventions used in the development of eHF modules.

Development Environment

Here you are introduced to the IDE and the associated tools like ANT build scripts, the Code Generator and various Software Life Cycle concerns.

Application Platform

This large section lists the modules that make up the eHealth Framework. The modules are categorized under the following types: Core, Security, Infrastructure, and Application

Appendix

The appendix contains further information on the background and technical details of various topics.

2 eHealth Framework Overview

The eHealth Framework (eHF) is a holistic approach for developing electronic health care solutions. It provides tools, interfaces, software modules, and documentation assisting you throughout the entire software development and product life cycle.



Figure 1: eHF supports the entire Software Development and Product Life Cycle

2.1 Metaphors

In order to introduce to the eHealth Framework and to phrase its purpose and goals the following metaphors are utilized. They provide different perspectives on the eHF and mention various keywords motivating the central concepts of the framework.

The eHealth Framework is the response to the latest industry challenges and trends.

In respect to this metaphor the eHF has to deal with industry standards and well accepted technologies. Furthermore methodologies derived or implied by the aforementioned trends have to be manifested.

The eHealth Framework is an optimized infrastructure for the development, hosting and maintenance of applications and solutions in the health care sector.

In order to provide an optimized infrastructure the eHF has taken a holistic approach to developing software in health care. It supports the complete software life cycle from conception and design to deployment and maintenance. Accompanying the whole development and deploy cycle, eHF is able to identify opportunities for improvement and can eventually optimize the complete software supply chain.

The eHealth Framework offers a kick-start advantage for the development of health care solutions in an environment with high demands.

In the context of this metaphor the various non-functional requirements can be introduced. Up front security and data protection are of central importance in the eHealth domain and - in combination with country specific legal restrictions or specifics - indicate a high demand on the software and its extensibility. Furthermore - apart from the software life cycle perspective that was already mentioned above - this metaphor leads directly into information life cycle concerns. In health care the lifetime of data may easily reach to the lifetime of a person and even beyond. The eHF has to anticipate this fact into the holistic approach and enable and optimize handling this concern.

2.2 eHealth Framework Elements

The eHealth Framework (eHF) consists of two main elements as depicted in [Figure 2](#). The first is represented by the Development Environment that provides tools to design, develop, build, deploy, and run eHF applications. The second element is the Solution Platform, which is further subdivided separating the Application Platform and the Integration Platform. While the Application Platform is dedicated to application development, the Integration Platform provides means to mediate and expose higher-level interfaces wiring applications together.

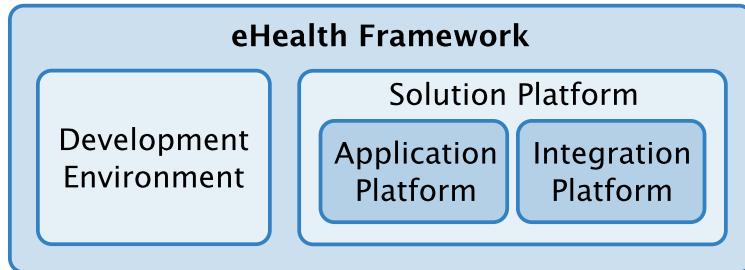


Figure 2: eHealth Framework

2.2.1 Development Environment

The Development Environment is your all-in-one productivity platform for developing applications with the eHealth Framework.

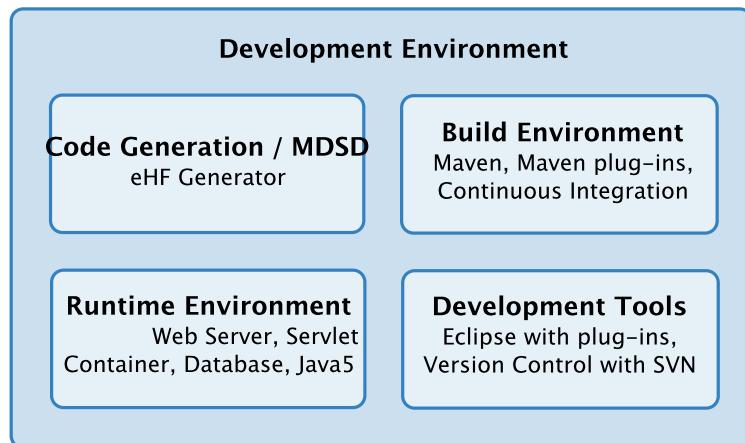


Figure 3: eHealth Framework Development Environment

Its most prominent part is the integrated modeling and development environment based on the Eclipse platform. The eHF Generator, driven by [openArchitectureWare](#) technology, boosts developer productivity by eliminating repetitive coding tasks. Build, assembly, and deployment are taken care of by the sophisticated build infrastructure based on [Apache Maven](#). And finally, the Development Platform bundles a comprehensive Runtime Environment that is required to run eHF applications.

2.2.2 Application Platform

The Application Platform is at the heart of all applications that you develop with the eHealth Framework. It consists of ready-to-use, modular building blocks. These range from fundamental core, infrastructure, and security modules to high-level, data-centric health care-specific application modules. They provide application programming interface (API) functionality, which can be used by health care applications or custom add-on modules. Modules can expose web service interfaces that can participate in an orchestrated service-oriented architecture (SOA) infrastructure.

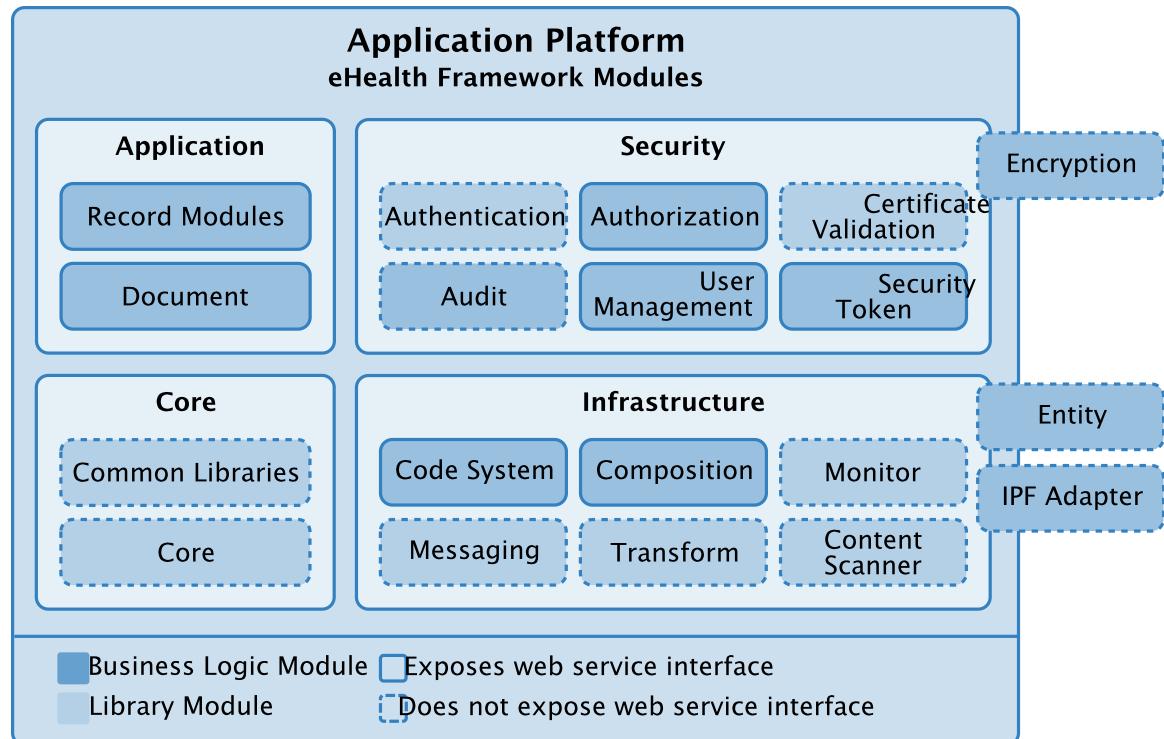


Figure 4: eHealth Framework Application Platform

2.2.3 Integration Platform

In the eHealth domain there are many possible scenarios where applications typically need to work together to support common business processes and to share data across application systems. These application systems need to be integrated.

The eHF Solution Platform provides an Integration Platform in order to leverage the Application Platform to cope with those integration scenarios. The integration strategy is based on the Open eHealth Foundation Integration Platform (IPF) (see [IPF](#) on page 365).

IPF is an extension of the Apache Camel Routing and Mediation Engine (see [Apache Camel](#) on page 365) and comes with comprehensive support for message processing and connecting information systems in the healthcare sector. Both, IPF and Camel, focus on a domain-specific language (DSL) to implement and combine Enterprise Integration Patterns (EIPs) in integration solutions. IPF leverages the Groovy programming language for application development and for extending the Apache Camel DSL. One example of a healthcare-related use case of IPF is the processing of HL7 messages.

In eHF, IPF is used in an embedded fashion and integrates with the modules based on the Application Platform. It offers a flexible and consistent approach to integrate with different

applications on different platforms and protocols. Any kind of protocol and technology can be used that is provided as connectivity component by IPF and Camel. Furthermore, eHF defines custom connectivity components that can be used in production environments to expose HTTP based services while preserving the overall modularization strategy. The integration solution supports two types for the exposure of services in eHF-based applications that can be classified into procedural-oriented and data-centric services. The choice for using one over the other is primarily determined by the concrete use case.

3 Central Concepts

This chapter lists some fundamental concepts that have driven the design of the eHealth Framework (eHF). Being familiar with these concepts will help you in reading and understanding the subsequent chapters of this reference documentation.

Each concept is introduced by a motivational section, followed by a brief description of the concept and its application in eHF. Please note that the general description is deliberately kept short in order to focus on eHF-specific aspects.

Please refer to the external references given in each chapter to learn more about the technical details.

3.1 Techniques and Methodologies

This chapter provides an overview on the technical concepts in eHF. Apart from central technological choices the methodologies applied in eHF are listed and explained.

3.1.1 Plain Old Java Objects

Plain Old Java Objects (POJOs) are ordinary Java objects that do not extend classes, implement interfaces or contain any annotations.

The eHF makes use of POJOs for the following reasons:

1. The resulting code is simpler, easier to understand and easier to debug.
2. POJOs do their job in every runtime environment. They are not dependent on a specific container or runtime environment.
3. Testing them with unit tests, without being tied to a specific application container is easy and fast. In addition they improve the code quality and increase the overall development speed which is also known as *edit-compile-debug cycle*.

With the help of the Spring Inversion-of-Control (IoC) container, POJOs are used in all layers of the application. Service objects, adapters, transfer objects and domain objects are all designed as POJOs.

There are two small deviations from the POJO principle: First of all annotations on domain objects are used for storing objects persistently. Second these domain objects have to extend a common base class. But these deviations occur in generated classes and thus in a controlled manner. The testability of the domain objects is not affected by these deviations either. These minor violations of the POJO principle improve the handling of these objects in the persistence layer of the Health Framework.

3.1.2 Aspect-Oriented Programming

Software engineering has come a long way in terms of modularization. Data and behavior are encapsulated in classes with well defined dependencies and interrelations. Classes can be bundled into libraries and larger-scale components. There are, however, problems that cannot be addressed by traditional approaches to modularization in that they cut across multiple modules at the same time. Typical examples include logging, exception handling, and transaction management. As a developer, you usually end up writing similar code over

and over again, scattered throughout the whole system resulting in tangled code, which is hard to maintain.

Aspect-oriented programming (AOP) is a paradigm that tackles these *cross-cutting concerns* by capturing them as *aspects*. Each aspect represents one such concern, untangled from the rest of the code. Aspects can be combined (*woven*) with the base system according to well defined rules, thus producing a complete software system. This results in a better overall modularization of the system as well as less tangled, more concise code.

Please see [AOP Explained](#) on page 351 for more details on the AOP paradigm.

The eHealth Framework (eHF) leverages aspect-oriented techniques to enable clean encapsulation of cross-cutting concerns in various areas. As a developer this allows you to concentrate more on a module's business logic and have the framework take care of the other aspects (literally) of the system.

In order to implement a solution for cross-cutting concerns, eHF makes use of two different AOP technologies: [Spring AOP](#) on page 365 and [AspectJ](#). The rationale for using two AOP technologies simultaneously is that Spring AOP can only operate on Spring-managed beans that expose their methods through well-defined interfaces. Whenever this is not the case, AspectJ has to be used.

eHF applies AOP in these areas:

- **Transaction management.**
- **Access control** and **object filtering**. See [Authorization](#) on page 162 A specialized approach is presented in [Usermanagement and Security Annotation Framework](#) on page 215.
- **Method parameter validation.** See [Commons](#) on page .
- **Generic service strategies.** See [Service Strategies](#) on page 142.

3.1.3 Object-Relational Mapping

During software development an object-relational impedance mismatch occurs if objects from an object-oriented programming language have to be stored in a relational database. This problem is rooted in the different underlying paradigms of these two techniques. Object/relational mapping (ORM) is often used to solve this problem.

ORM constitutes the mapping of object-oriented data structures to tables of a relational database. The O/R mapper largely hides this concern. It hides the relational database schema behind an API and mimics the functionality of an object-oriented database. It provides an object-oriented view of tables and relations in relational data management systems. Thus, developers may operate on objects instead of using SQL statements. ORM uses metadata to describe the mapping between the objects and the database and provides reversible data transformations from one representation to another (and vice versa).

The eHealth Framework (eHF) uses Hibernate (<http://www.hibernate.org/>), an open source object/relational mapping implementation for Java.

3.1.4 Inversion of Control

In software systems there are a lot of dependencies between objects. If they are hard-coded (for example, by instantiating objects in constructors) a single change may break the code in several places. Thus, one single change results in a cascade of changes in dependent objects. Therefore, managing these dependencies is an important issue.

Inversion of Control (IoC) is a way to break apart dependencies in applications. It is a paradigm which describes a specific way of managing dependencies between objects. A function which should be provided to other objects will be registered in a standard library.

This library may call the function later on. This approach is also called the *Hollywood Principle* ("don't call us, we'll call you.").

The eHF uses the Spring Framework to manage the dependencies between its objects. Spring provides a lightweight IoC container, which enables a loosely coupled design.

3.1.5 Public API

Contracts in software development are manifold. They are established between the provider and the consumers of a functionality.

The motivation for the contract is different depending on the perspective. The provider wants to make his functionality available. However he is interested to narrow the contract as much as possible in order to not expose functionality that is not ready to use or exposes parts of the contract that is unstable and may be subject to a future change, which may cause support issues in terms of backwards compatibility.

The consumers on the other hand have a strong desire to get a stable contract that satisfies their needs. Mostly they exploit what they get offered by the contract. In particular they are interested to not require adaptations on code level with every minor release of the functionality.

In the above description it is clear that a definition for a contract is required. The most primitive contract in Java is an interface. A more sophisticated contract can be defined by an API consisting of interfaces and classes. But where can the boundary between the public available classes and interfaces be drawn? In terms of a Java API this contract may not be made explicit. In such a case the consumer may use interfaces and classes that were not intended for being used (at least not at this point in time).

In Java and with other technologies (WSDL, technology specific configuration) a contract may be very wide. How can this contracts be narrowed to meet the requirement of both - provider and consumer? Meet the desire for stable interface, when they are ready.

In eHF this aspect is addressed by the public API approach. The public API approach provides means to define the contract and also to technically enforce the contract. With this in place the provider can be very careful to not expose too much of his internals and therefore stay flexible in the regions that have not been leveraged to a public visibility yet. The consumers are satisfied as they get a well-defined contract and have not to expect that this parts are unstable. Moreover the consumer is as satisfied as the provider - while extending the functionality of his software - is able to meet the challenges of his feature road map, because he doesn't have to worry about changes to take effect of the API being - however unknowingly or accidentally - used by one or more consumers.

3.1.6 Modularization

In order to control large software systems, a decomposition into smaller, more manageable units is required. These units must have defined boundaries and a functional responsibility in the overall system. One can even imagine having various levels of decomposing a system in such a way.

The software industry has been very inventive in organizing the code base and functional units of large systems. Terms like components, modules, packages, bundles, artifacts are widely used and have different meanings in their respective original contexts. Also more and more technologies define such functional units and standards are evolving around the aforementioned terms. For instance, the [Java Servlet API specification](#) defines a separation using the term *web application archive* or shorter *WAR* file. [OSGi](#) uses the term bundles, others the term module (for example the [Java Module System](#) JSR).

Common to all those approaches is the desire, to organize functional units and to introduce sustainable concepts and tools, supporting the definition and management of these units.

A very prominent term in this context is *modularization* and the concepts can be lead back to it. In a modularized approach you are decomposing your system in functional units called modules. Modularized systems are in strong contrast to monolithic systems. Moreover modularized systems may add a life cycle on top of a module and consider the replacement of modules even at runtime (hot deployment). It is possible to distinguish logical modularization (isolation via interfaces) and physical modularization (technical measures to isolate a module). For instance the above described technology can be classified as physical modularization approaches. This is mostly the case because Java does not provide support for physical modularization.

Modularization has been an important concept to eHF from the very beginning in the early design and implementation phases. Modularization is regarded as a key concept to isolate the internal implementation of a module from other parts of the system and regarding it as a black box. This approach has many benefits. Clarity of the overall system design, clear interface-based usage of functionality, enabling internal module evolution, hiding module internal details that are subject to change or would otherwise enable to compromise the functional responsibility of the module, are only a few examples to list here.

Furthermore eHF also uses a hierarchical decomposition as mentioned above. A module itself can be composed of components, while components can themselves be classified to belong to specific layers of the software architecture. See [Module and Component Meta Model](#) on page 28.

To define and manage modules, additional concepts were realized in eHF – like the concept of a [public API](#) on page 10 – and technical support and tooling. Those concepts enable the adopters of eHF to develop on top of all the advantages of modularization while abstracting from the technical concerns, which are covered by the framework.

Additionally eHF supports a self-contained modules approach. This means that a module has to provide all information that is associated with it. In particular this means that all accompanying artifacts to manage and maintain the module (that is its documentation, upgrade configuration, backward compatibility configuration) are part of the module.

Ultimately modules can be composed to construct applications and therefore offer themselves for reuse. Essentially this is the approach ICW uses to develop modularized and extensible products. For more details on modularization see [Modularization](#) on page 26.

3.1.7 Domain-Driven Design

Software development often faces a great divide between the results of domain analysis and the system's actual implementation. The latter tends to be driven by technological factors and the domain is often only an afterthought. Developers and domain experts regularly do have difficulties in communication, and the resulting systems fall short of the stakeholders' expectations.

Domain-driven design (DDD) is all about capturing the essence of an application's domain in a rich, yet concise model as the central part of a software system.

DDD attempts to build software systems around a central domain model, which becomes an integral part of a ubiquitous language and a focal point of software development.



Note: It is important to point out that domain-driven design is not a technology or a methodology. It is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains.

Apart from the general DDD mindset, the eHealth Framework (eHF) employs many concrete patterns that facilitate the implementation of such a design. Examples include fundamental building blocks like *Entity*, *ValueObject*, *Service*, *Factory*, and *Repository*, as well as *Layered Architecture* on a larger scale, *Anticorruption Layer* for evolutionary and boundary concerns, or *Continuous Integration* to support the development and integration process.

DDD patterns support the partitioning of both the domain model and the code, and they also address life cycle issues in order to provide a software system that can evolve, yet remain stable over a long period of time. Domain-driven design is explained in great detail in the book [Domain-driven Design](#) on page 365. Please note, however, that the terminology and characteristics of the patterns presented in the book may not always match common preconceptions and may present themselves in various forms and combinations. So while eHF is true to the spirit of domain-driven design, it may not always be true to the word of it.

One such example is that DDD tends to be code-centric. The domain model is typically presented as a piece of code. However, DDD lends itself to model-driven development, and eHF uses this approach to capture the domain. Instead of drawing the domain model on whiteboards or graphic documents and implementing it manually, a more formal approach creates an even more concise domain model and puts it at the center of application development. That is why an already potent concept makes it even more powerful. See [Model-Driven Software Development](#) on page 12 for more information on this approach in general, and [Generator](#) on page 88 for its application in eHF in particular.

3.1.8 Model-Driven Software Development

When building any non-trivial software system, and especially when employing frameworks and external execution environments, you often end up writing a multitude of boilerplate code and configuration files, whose sole purpose is to integrate your application with other system components. This work is tedious, error-prone, and distracts you from the actual task at hand: understanding and implementing a complex health care solution.

Model-driven software development (MDSD) is a paradigm that leverages the concept of abstraction to increase software development efficiency. Instead of writing large amounts of repetitive code, the developer can capture the core concepts of a software system in a more abstract form, namely that of a *model*. Automated transformation, generation, and sometimes interpretation of such a model creates a substantial part of the final software product, which is typically enhanced with manually coded fragments. The main idea is that you can gain time and increase quality by expressing parts of the software in a language that is better suited to the problem at hand than a general purpose programming language.

In the eHealth Framework (eHF), a module (see [Modularization](#) on page 10) typically revolves around a domain model (see [Domain-Driven Design](#) on page 11). This model captures the essence of a module's business assets: its data structures, interrelations, validation rules, service exposure guidelines, and more. This model manifests itself in various forms at many different places in a module's implementation. See [Architectural Layers](#) on page 31 for an overview of eHF's architectural layers.

eHF uses an architecture-centric MDSD approach to help developers create eHF modules. Taking the domain model as input, the eHF Generator creates Java classes, Spring configuration files, and other artifacts required to fit a module into the overall eHF architecture. This approach is architecture-centric as the developer concentrates primarily on the domain model and business logic, while most architecture-related artifacts are taken care of by the generator. Apart from the obvious benefit of relieving the developer of hours of error-prone, repetitive coding tasks, this also ensures a consistent architecture throughout all eHF modules. How exactly the eHF MDSD approach works and what kind of artifacts are created is explained in more detail in [Generator](#) on page 88.

The domain model itself is conceptually very close to a Java class model, and UML2 captures most of the required concepts. Actually, UML2, being a general purpose language, covers much more than the required concepts. For that reason, UML2 in combination with a custom UML2 Profile and a set of constraints is employed to create a Domain Specific Language (DSL) tailored to the eHF.

3.1.9 Meta Modeling

Developing software and everything that goes along with it is based on a significant amount of communication and discussion. Especially when developing in a larger group of people with different educational backgrounds and professions, communication and discussion may lack a common vocabulary or terminology. As a consequence misunderstandings lead to false interpretation of goals and tasks.

In order to reduce general misunderstanding and to create a solid foundation for discussions and the derived work, an approach called meta modeling can be used. In this approach you start to depict the understanding of your problem or concept at hand by creating a visual representation (standard UML proves very helpful in this respect) and by discussing and tuning this model with others.

By raising questions and giving answers the model can be quickly turned into something that can be very useful, if applied correctly. Good meta models define a tool for communication without much of a documentation and definition overhead.

When developing eHF, a significant amount of meta modeling was used and had an impact on the way the framework was derived and communicated. Especially the [Module and Component Meta Model](#) on page 28 defines a valuable tool for illustrating inter-module dependencies.

While it may seem awkward to approach every problem using meta modeling, you may experience a shift in discussion culture and the way people communicate and work using meta models.

3.1.10 Enterprise Integration Patterns

Integration solutions have to deal with multiple applications running on diverse platforms talking via a variety of large number of transport protocols. This can get quite difficult as a significant complexity stems from mainly two challenges. First, you have to deal with the specifics of applications and, second, you have to come up with a good and sustainable solution to the integration problem at hand.

One of the major challenges faced by integration solutions is the usage of different protocols and technologies. Although the usage of those is relatively simple on its own, the mixed combination in an integration solution brings complexity into it. However, this technical infrastructure addresses only a small portion of the integration complexities. The other challenge of integration is about problems when focusing on the high level design of how applications interact. This is where design patterns are applicable and useful. In general

the purpose of design patterns in software engineering is to document best practices of a recurring design problem within a particular field. In the context of integration, design patterns have been derived that are commonly called Enterprise Integration Patterns (EIPs). EIPs focus on the interaction of applications while ignoring the mechanics of how to connect with multiple protocols.

EIPs have been described by Gregor Hohpe and Bobby Woolf in their book Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (<http://www.eaipatterns.com/>). They represent experiences of integration architects and can be used as building blocks for designing integration solutions. These patterns were published as recommendations without implying a concrete implementation.

The routing and mediation engine Apache Camel (see [Apache Camel](#) on page 365) provides concrete implementations of the most common EIPs. Further, it provides connectivity to a great variety of transports and protocols. The Open eHealth Integration Platform (IPF) (see [IPF](#) on page 365) is an extension of Apache Camel and comes with comprehensive support for message processing and connecting information systems in the health care. The eHF Solution Platform uses IPF in an embedded fashion adapted for web applications. Therefore eHF-based applications can profit from all the available integration features IPF and Apache Camel provide.

3.2 Health Care Domain

Many concepts in eHF are not technically motivated, but result from insights in the health care domain. The intention of this chapter is to provide you with an overview of the fundamental domain concepts that influenced the development and architecture of the framework.

3.2.1 Codes, Code Sets and Code Systems

Applications based on the eHealth Framework have two main requirements regarding the information they manage and exchange with other applications. Within an application there are language independent value ranges to manage. With the exchange of information between applications, the focus is primarily on ensuring semantic interoperability.

Managing Language Independent Values Ranges

If you want to have an application that can support multiple languages you should avoid saving information in a free text format as then the saved information can only be displayed in the language in which it was entered. Lets take the example of a person in the role of patient. A person has besides first name and family name also information about gender and marital status.

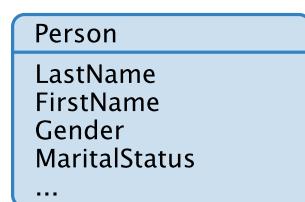


Figure 5: Domain Class "Person"

If you save the marital status of the patient as "single" or "married" this information can no longer be determined when another language is chosen. When the information comes from a well-defined range of values (for example, gender and marital status) it's useful to save the language-independent code instead of the actual text, as shown in [Table 1](#).

Code	English Description	German Description
1	Divorced	Geschieden
2	Legally seperated	Getrennt lebend
3	Married	Verheiratet
4	Never married	Ledig
5	Domestic partner	Lebensgemeinschaft
6	Widowed	Verwitwet
7	Polygamous	Polygam

Table1. Coded Value Range for Marital Status

Additionally, when you introduce codes information it has the effect that information about a person based on the coded values can be further analyzed and processed. The set of all codes that logically belong together is called a code system.

Ensuring Semantic Interoperability

In the context of integration with other applications the exchange of encoded information plays a special role. Here the challenge is to ensure the correct interpretation of information across application boundaries. This is done by defining terms to have only one fixed meaning that can then be used across applications. When the exchanged information is analyzed by the applications, for example, to control more processes or to create evaluations, then not just agreement on the terms, but also language independent and system interpretable codes are necessary. Therefore, a standardization of the code systems used is required.

Information about the current status of some international standard organizations can be found in the Table: Standardization of Vocabularies.



Note: This list represents only a subset of organizations that deal with standardization issues, some of whom are involved in health care.

Organization	Link	Comment
Health Level Seven (HL7)	http://www.hl7.org	Here (under Resources / Vocab Resources) you find a list of all the code systems defined in the context of HL7.
Integrating the Healthcare Enterprise (IHE)	http://www.ihe.net	For many of IHE profiles there are defined code systems to be used.
World Health Organization (WHO)	http://www.who.int	The WHO specifies the International Classification of Diseases (ICD). See http://www.who.int/classifications/icd/en/

Organization	Link	Comment
International Organization for Standardization (ISO)	http://www.iso.org	

Table2. Standardization of Vocabularies

Vocabularies can not always be defined independently of localization. Therefore, there is in every country legal requirements or recommendations regarding the use of vocabularies and classifications.

The ICD, defined internationally by the WHO, is extended in many countries to provide additional diagnostic codes. For example every year DIMDI (the German Institute for Medical Documentation and Information, see <http://www.dimdi.de>) publishes an ICD classification adjusted for German use. To clearly identify code systems worldwide an object identifier (OID) is assigned to each code system. The structure and encoding of OIDs is defined in ISO IEC-8824 and ISO IEC-8825 (for more information, see <http://asn1.elibel.tm.fr/oid/standards.htm>).

Links to OID registries and repositories, which also provide you with information on the registration of OIDs, include:

- <http://www.oid-info.com> One of the most comprehensive OID repositories containing information on more than 90,000 Object Identifiers.
- <http://www.hl7.org/oid/index.cfm> Lists all HL7 registered OIDs, both for code systems, which HL7 has defined itself as well as for code systems managed by other standards organizations.
- <http://www.dimdi.de/dynamic/de/ehealth/oid/verzeichnis.html> Many countries manage their own OID repositories, as an example the German OID register is listed.

For the example introduced above, the marital status HL7 has defined the following code system under the OID 2.16.840.1.113883.5.2.

Code	Print Name	Definition
A	Annulled	Marriage contract has been declared null and to not have existed
D	Divorced	Marriage contract has been declared dissolved and inactive
I	Interlocutory	Subject to an Interlocutory Decree
L	Legally Separated	
M	Married	A current marriage contract is active
P	Polygamous	More than 1 current spouse
S	Never Married	No marriage contract has ever been entered
T	Domestic partner	Person declares that a domestic partner relationship exists.
W	Widowed	The spouse has died

Table3. HL7 Code System MaritalStatus

Use of Code Systems in eHealth Framework

eHF Data Type "Code"

In eHF the data type Code is defined to support the encoding of domain information in application modules. The data type Code has the attributes systemID, version and key. The systemID contains an OID, that uniquely identifies the code system, to which the key belongs. Currently the version number is not used (by default it is set to: 1.0.0).

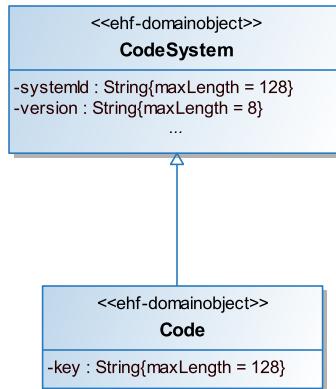


Figure 6: Data Type "Code"

Use of Code Sets

While code systems are often specified independently of the application applications define their own values through their application context, for example through the localization context. Therefore, application-related code sets are defined, which depending on the localization can differ in the quantity of valid codes.

A code set includes a subset of codes from a code system, corresponding to the range of values of the attributes in a localized application context. As an example we define, based on the HL7 code system MaritalStatus, two localized code sets as shown in [Figure 7](#). The code set EXT-GEN-MARITAL-STATUS-DE contains the marital status relevant for Germany and the code set EXT-GEN-MARITAL-STATUS-US contains the marital status relevant for the United States, regardless of the code system supported translations. They make differing use of the code P (Polygamous).

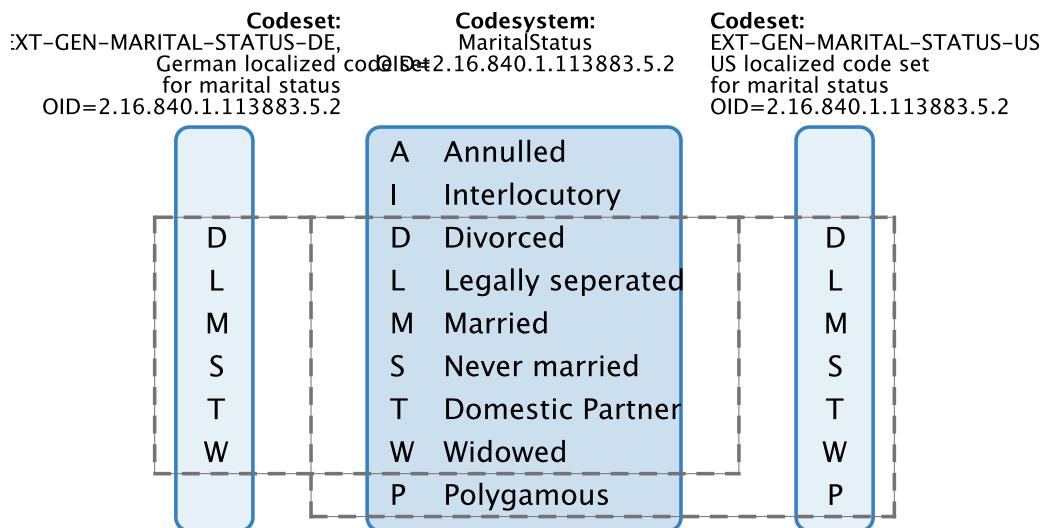


Figure 7: Definition of Code Sets for Marital Status

Versioning of Code Sets

When codes are used to exchange information between applications, the problem of backwards compatibility needs to be taken into account. As a result eHF supports the versioning of code systems (see "versioning schemes"). If new codes are added, the minor version number is changed (1.x.0). If codes are removed from the code set because they are no longer supported or if a code set based on another code set is restructured the major version number changes (x.0.0). The third integer, the patch version, is not currently used.

Localization Independence with Code Categories

Applications must make it possible to localize domain attributes, such as for example marital status. However, you should avoid linking the localization information directly with the domain attribute. Therefore, Code Category is introduced as an abstraction from the localized values for this attribute. The Code Category is clearly identified by a name and provides a reference to the values of the corresponding attribute. But now how do you get from a category to a code set? During the bootstrap, a location specific configuration file imports the code sets including their assignment to the appropriate category code. At runtime the application module can then, by stating the code Category, identify the valid codes. [Figure 8](#) shows the use of a German code set for marital status.

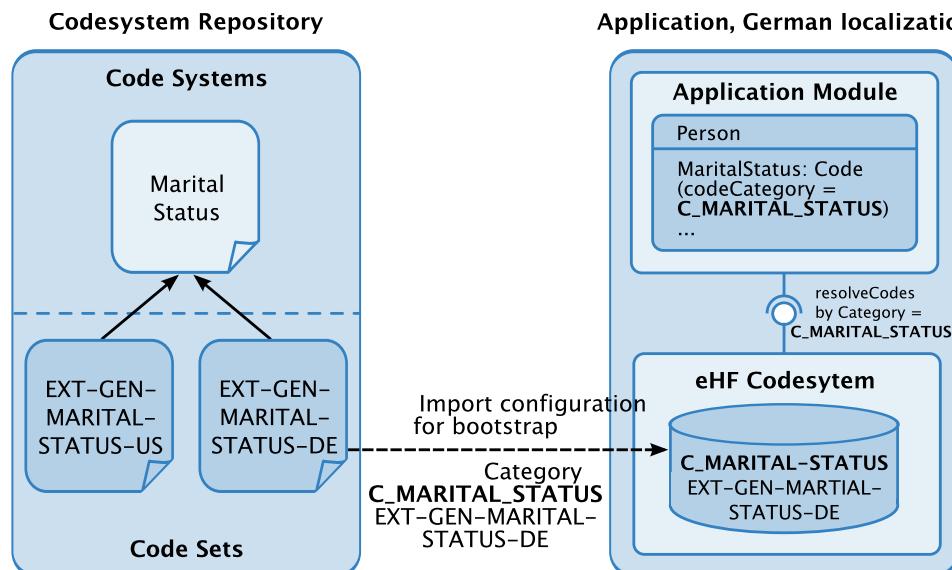


Figure 8: Localization Independence with Categories

3.2.2 Application Security

It lies in the nature of the health care domain that basically all data are confidential. It is mandatory to define and control who is granted what kind of access to which parts of the system. Ultimately, this is the responsibility of a health care application, but the eHealth Framework (eHF) provides building blocks that facilitate the implementation of effective application security.

Application security comprises methods to verify who is interacting with the system, and to check whether this interaction is permitted. Additionally, application security attempts to prevent unauthorized system access through forgery or circumvention of security measures.

The eHealth Framework supports application security in the following areas:

- The **Authentication** module verifies the identity of interacting users, typically by username/password or PIN combinations. See [Authentication](#) on page 153.

- The **Security Token Service** supports authentication by providing additional authenticity validations based on WS-Trust security tokens. See [Token Service](#) on page .
- The **Certificate Validation** service manages authentication through X.509 certificates, including rejection of revoked certificates. See [Certificate Validation](#) on page .
- The **Authorization** module ensures that an authenticated user may only perform permitted actions. See [Authorization](#) on page 162.
- The **CSRF Guard** prevents cross-site request forgery attacks by encoding unique request/response identifiers into the client/server communication. See [Commons Security Library](#) on page 146.
- **Hardened Apache and Tomcat configurations** attempt to close loopholes that may be used by intruders to gain access to the system, for example through unused modules.

3.2.3 Data Security

Medical data are the most precious content in health care applications. It is therefore mandatory for any health care application to protect their data from unauthorized access and tampering. The eHealth Framework (eHF) provides building blocks that facilitate the implementation of effective data security.

Data security comprises methods to protect an application's raw data. While application security works *within* the application, data security attempts to protect the data *outside* of the application's reach. There, it may be subject to access or tampering by unauthorized, yet privileged persons, for instance administrators.

The eHealth Framework supports data security in the following areas:

- At the lowest possible level, **Oracle Database Encryption** is used to render the raw database contents illegible unless the appropriate security credentials are provided.
- **Transport Level Encryption** (HTTPS, SSL) is used whenever data are transported over a network. This prevents sensitive data from being captured on the network layer.
- A **Content Scanner** can be used to filter malicious content, for example computer viruses or worms, which may gain access to servers or users' PCs. See [Content Scanner](#) on page 186.
- The **Audit** module logs security-related events such as authentication attempts or access to medical information. See [Audit](#) on page .
- A hardened **Exception Handling** mechanism is employed to prevent stack traces from leaking out of the system boundary. They may contain sensitive information which may be used by intruders to attack the system. See [Exception and Error Handling](#) on page 127.
- **Patches for the Axis web service library** are applied in order to prevent host names from being transmitted as part of web service responses. This information may be used by potential attackers.

3.2.4 Scope

The health care sector has high demands regarding application and data security. The eHealth Framework (eHF) provides facilities to control the access to data in a very fine-grained manner. However, managing security access information for every individual chunk of data consumes both processing time and database volume. This may ultimately lead to security related data exceeding the amount of actual payload data being stored.

The eHF therefore leverages the concept of **scope** to group related data into a sort of logical record. This is a typical pattern in the health care domain, where several pieces of information, for example medical observations or measurements, share a common logical

context that they belong to. The scope groups the different parts in some kind of ownership relation, where the individual parts themselves need not be interconnected in any way, neither directly nor indirectly.

Associating data with a scope significantly simplifies permission management since such a scope typically lends itself to assigning access permissions to a large set of data at one stroke. The peculiarities of the health care domain thereby help the eHF to be more focused and more efficient at the same time.

3.3 Software and Information Life Cycle

Health care software puts a high demand on continuity of the software, which is an important factor in its development. Medical data stored by an application must be continuously available throughout a person's lifetime - and possibly even beyond.

Therefore it is essential that eHF provides means to support the information life cycle of the data as well as software life cycle aspects of the applications accessing this data. This chapter introduces the basic features of eHF in this respect.

3.3.1 Backwards Compatibility

The backwards compatibility concept has been developed as part of the eHF. This concept embodies a change enabling, modern framework architecture that is equally durable, stable, and compatible.

Backwards compatibility refers to a client software system that can use interfaces and data from earlier generations of the application. If existing users are unaffected then the change is backwards compatible. If existing users are affected then the change is not backwards compatible and a strategy is required to manage the impact of the change.

Versioning Schemes of Modules

In particular, when a developer wants to use a framework, he usually needs to know which version of the software he can use. This step is usually the least complicated; he will pick one of the existing versions and start using it. However, it can become complicated when he wants to upgrade to a newer version of the current framework at a later point in time. He will need information about the backwards compatibility, the amount of new features and the possible issues that may occur when switching to a newer version.

To address these issues, the eHF applies the versioning scheme of the [Apache Portable Runtime](#) (APR) project. Versions are denoted by using a standard triplet of integers MAJOR . MINOR . PATCH (for example 2.9.4).

APR defines rules concerning major, minor and patch version increments. In eHF the following rules apply on the level of the public api. Please note that these rules are not as strict as the original rules from APR. The reason for a relaxation of the rules are to gain more flexibility for developing eHF, while still considering the needs of a consumer.

Major version

A major version can be incompatible with any other major version. Major versions represent large-scale upgrades. Generally, when you upgrade to a newer major version, you have to expect significant effort in changing your client applications.

Minor version

A minor version is backwards compatible with older minor versions on source level (source compatibility). This means, if you are using an older minor version in your application, you can easily upgrade to the newer version. However, once you have started to use

functionality, which was introduced in the newer version for the first time, you cannot use your application with older versions anymore. Please note that this rule was relaxed to describe only source compatibility.

Binary compatibility is not considered relevant for minor upgrades and eHF does not define any binary compatibility support. This relaxation means that you first have to recompile your projects against the new version of eHF.

Patch version

The third integer of the version number identifies the patch version. Patch versions are backwards compatible on source and binary level. APR claims that patch versions have also to be forward compatible (or upgrade compatible). This means that you can also go back to a former patch release.

Web Services Backwards Compatibility Infrastructure

The versioning scheme of modules is the rationale for the backwards compatibility concept. However in eHF, the backwards compatibility of web services plays an important role. The eHF has anticipated this and therefore provides a multi-level strategy for backwards compatibility as shown in [Figure 9](#).

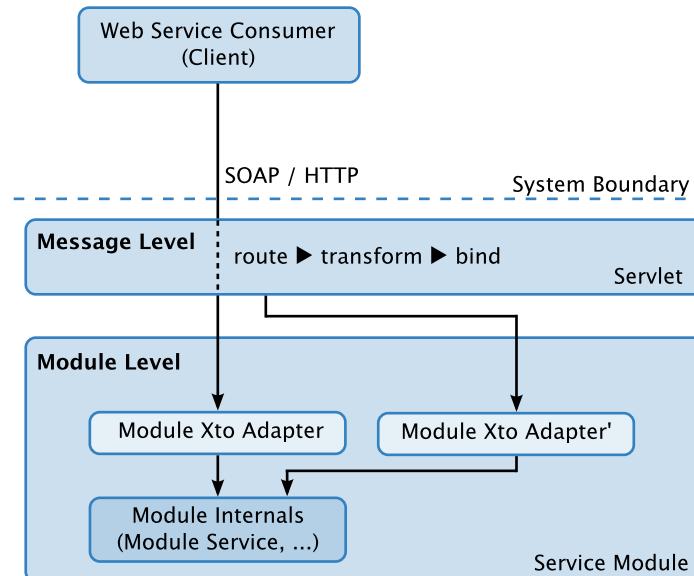


Figure 9: Web Service Backwards Compatibility Infrastructure

On the message level, the web service servlet transforms SOAP messages to be backwards compatible, whereas service modules provide backwards compatibility with adapters on the module level. Unfortunately, the second approach has several drawbacks. First, It requires a duplication of configuration and adapter classes on the level of the module. Second, it may also require additional transformations on the message level. Therefore, this approach should only be used, if backwards compatibility can not be maintained by transforming SOAP messages.

The chapter [Web Service Backwards Compatibility](#) on page 94 addresses the web service backwards compatibility on the message level in greater detail and discusses how a developer is able to build a backwards compatible module.

3.3.2 Upgrading

While the last section covered the life cycle of applications based on the eHealth Framework, the following explains the life cycle of the underlying data that is stored in relational databases. The data structures of the database and its entities in eHF depend mainly on the defined domain logic. If the domain logic of a module is changed, it is very likely that the corresponding database structure will have to be changed as well. However, while the software should be as flexible and change compliant as possible, the consistency of the data and its structure has to be maintained between different releases under all circumstances.

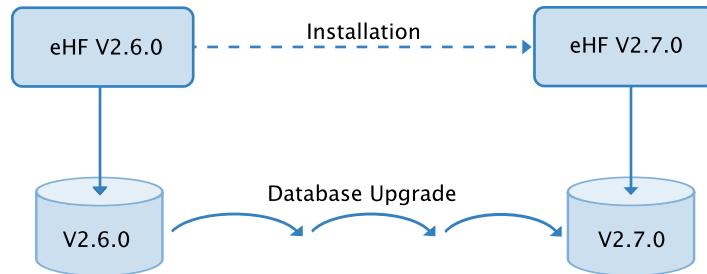


Figure 10: Conceptional Upgrade Process

A database upgrade usually includes different tasks. For instance, typical tasks of an upgrade process are the modification of schema and structure elements, the updating of records and data fields, and the validation of the data integrity. Therefore, it is best to divide the different tasks in several steps as shown in [Figure 10](#). One advantage of this method is that each step is checked separately making it easier to identify potential error causes. However, the application should only be deployed after the upgrade process is completely finished.

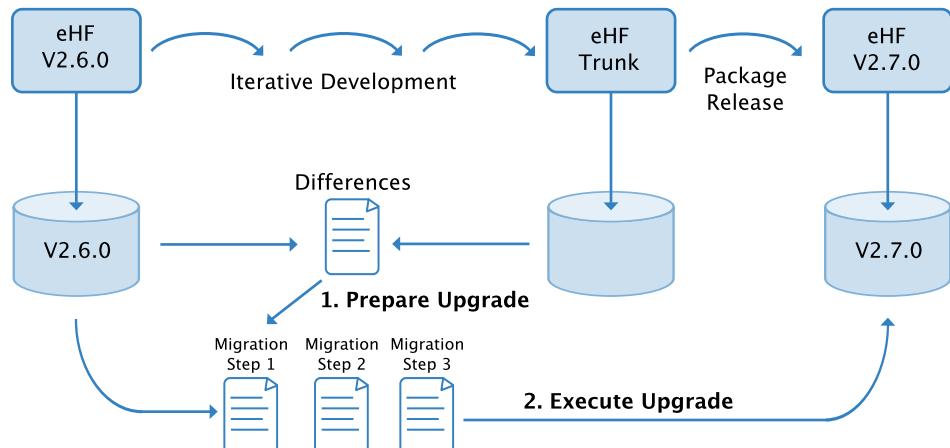


Figure 11: eHF Upgrade Process

[Figure 11](#) shows the upgrade process concept that is used in eHF. The upgrade process is divided into two areas, the preparation and the execution. In this example, the application V2.6.0 is the latest released application. While the version 2.6.0 remains stable, the development of new features or change requests proceeds on the current version (*Trunk*) that will lead to changes in the database. At a certain point in time a new version of the module will need to be released. First, the changes between the database schema of version 2.6.0 and the current version will be collected with the help of a difference analysis tool. Then, each of these changes are separated and configured in migration steps. Finally, the migration steps are packaged into the application assembly and the upgrade preparation is finished. The package release 2.7.0 contains both the eHF application and

the configured migration steps so the upgrade can be executed at a later point in time in the test and production environments.

3.3.3 Continuous Integration

Continuous Integration (CI) describes various practices in software engineering helping to speed up the delivery of the product by decreasing integration times. This is accomplished by the team members when they frequently integrate their work — either multiple times a day or at least once a day. Each integration is verified by an automated build (including tests) to detect integration errors as quickly as possible. Primarily CI is about communication between team members. If you want to read more about the technologies and approaches in Continuous Integration please refer to the [article by Martin Fowler](#). ➔

Build Strategies

In order to encourage frequent builds to test integration, it is very important to keep builds fast. The key to fast builds and fast feedback for the developer is identifying two different types of builds. The *Commit Build* and the *Integration Build*.

The *Commit Build* is triggered every time a commit on a module is performed. It runs on a build server, which ensures that the module compiles using the latest sources and integrates on a binary level. Not only is code compilation part of the Commit Build, but also unit test execution, code analysis and report generation. This type of build is very fast.

The *Integration Build* is slightly more complex, it tests all the modules together that have been assembled into an application. The application is deployed into a production-like environment and an automated test suite is executed against it. The Integration Build runs nightly or on demand. Both build types are configured in one Continuous Integration server per code line.

Communication

Having two different type of builds ensures that the feedback about the quality of code changes is always provided within at least one day and even faster on the module level. Feedback is provided by the Continuous Integration Server. Information generated from the nightly tests, continuous tests, code coverage reports, and Checkstyle reports is aggregated by a custom tool and published to a Wiki via RSS Feeds. This facilitates communication within the team.

To read more about how continuous integration is practiced in eHF see [Continuous Integration](#) on page 86

Part II - Architecture

4 Conceptual Architecture

In this chapter the architectural principles and concepts are detailed. The main purpose is to describe the architecture on an abstract level, not to include technology and implementation details.

How the architecture is realized is the subject of the chapter [Architecture and Life Cycle Realization](#) on page 34. However, it is strongly recommended to read this chapter before diving into the eHF-specific implementation of the architecture.

4.1 Primary Objectives

The eHF architecture, design and concepts are driven by the following objectives. These objectives are largely derived from non-functional requirements applications in health care have to meet. They are further ordered in terms of priority to emphasize the explicit needs in the demanding health care environment.

Application Security

Enterprise-class authentication and authorization mechanisms ensure a very high level of security for sensitive medical and personal data in health records. The access of users to individual data objects is restricted according to the granted access rights.

Data Security

In addition to security on the application level the protection of data on the transport and database level is of central importance. The eHF provides sophisticated concepts on how to protect data from unauthorized access and theft.

Extensibility and Integration

The eHF architecture supports fast and efficient development of new modules, services, and applications. This development can be done by ICW or partners. The architecture requires to support straightforward integration of eHF services into heterogeneous existing health care environments and higher-level business processes.

Modularity

Modularity supports the distributed and loosely-coupled development of application modules by ICW and partners. The eHF architecture encourages and promotes the reuse of modules, components and services. This leads to a significant increase in speed and flexibility during the development process. Complexity, overall costs, and time-to-market for new or adapted functionalities are reduced.

Layered Architecture

To separate vertical concerns, different abstraction layers are required. These layers must be well-defined and access to other layers of the system must be narrowed and controlled, isolating technical details of adjacent layers. Therefore, bypassing layers is generally not allowed. A layered architecture enables a higher flexibility concerning improvements or replacements of technologies used in an individual layer without affecting other layers.

Separation of Concerns

In addition to architectural layers that isolate concerns vertically, general cross-cutting concerns are addressed using aspect-oriented programming (AOP) methods. In other words, business logic should be strictly separated from concerns, such as security logic

or transaction management. Architectures supporting AOP allow for better reusability of application components and easier maintainability.

Scalability and High Availability

The architecture scales from small user numbers to several million users. A system can be deployed to support high availability. This means that the system is not dependent on any single point of failure and that maintenance of the application will not affect availability.

Testability

The architecture supports fully automated functional testing on different levels (unit, module, service, application, solution). Furthermore, the architecture enables automated regression-, compatibility and upgrade tests.

It is a fundamental prerequisite for testing that the APIs in an application are well-defined, documented and exposed very consciously.

Portability

The architecture of the eHF allows for the smooth porting between different Java EE containers. Portability is further promoted by an orientation towards Plain Old Java Objects (POJO) programming models.

Standard Technology

The eHF is based on open source frameworks which are widely accepted and can therefore benefit from the experience and skills of a constantly growing community. This attracts partners who are familiar with these technologies and flattens what would otherwise be a steep learning-curve.

Serviceability and Maintainability

Software in a production environment has to satisfy strong service level requirements. The eHF defines standard procedures and tools for operating, maintaining and managing an eHF-based application.

4.2 Modularization

Modularity, extensibility, a layered architecture, separation of concerns, testability and portability are strong objectives, which significantly influence an architecture. There are various levels on which to approach these objectives and the concept of modularization is the most coarse-grained level to start with. A module isolates a functional concern and focuses on a specific domain. A module is connected to others on a distinct architectural layer and can be tested by consuming the functionality exposed by the module. A modularized application is by definition extensible as you can simply add an additional module to it.

Apart from the fact that modularization is the concept of partitioning your system into functional units it has further advantages and benefits over a monolithic approach, which is its opposite.

Clarity of Overall System Design

When organizing all your functionality into modules your overall system design is much cleaner and more structured than in other approaches. The modules and their functional responsibility is easy to grasp and to communicate. Using a combination or sequence of modules you can easily illustrate functional use cases as they are processed in the system.

Interface-based Access

Modularization - if manifested consistently and forcefully - only allows access to functionality using defined interfaces. A consumer of a module's functionality cannot access internal

parts of the module. He is forced to make use of the selected and exposed functionality through interfaces the module defines at its boundaries.

For this to work a clear definition of the module boundaries and a technical enforcement is required. Otherwise violations are not prohibited and will be considered available. The eHF architecture therefore provides tools to define the boundaries of a module and to enforce this boundary is not violated.

Isolation for Evolution

By consciously defining the boundaries of a module the internal implementation details can be hidden. Hiding in itself cannot be regarded as an advantage here, but the fact that you can constantly evolve the internals of your module - without having to worry about destroying an undefined contract with a consumer - keeps you flexible.

In this context, a principle of Domain Driven Design may also help to understand the benefits. In Domain Driven Design it is regarded as essential that you constantly evolve your understanding of the domain and simultaneously your code. Modularization and the concept of a defined public API help you to understand and eventually not to break your contract, while still being able to change the internals of your module.

Isolation for Integrity

In order to implement a certain kind of functionality usually several components act in concert with each other and a request has to follow a certain path through these components in order to produce a consistent and valid result. Bypassing parts of the execution (for example, accessing it at a deeper level than anticipated) may endanger the integrity of the result and ultimately the system.

Modularization and enforcement of a public API prevent such a violation. The functional responsibility of the module cannot be compromised.

Public API Rules

Some rules can be extracted and derived with respect to the public API:

- It is prohibited to violate the public API of other modules.
- It is prohibited to bypass the public API of other modules.
- Direct access to the database schema of another module is prohibited.
- Any assumptions on a technical level that are not part of the public API of the module are not allowed.

Such rules and their consequences may appear artificial in some cases. For example, the limitation to access all data via the API, as mentioned above, does not seem to be appropriate. However, in the health care domain several access control and data protection requirements have to be considered. The eHF is positioned in this domain and aims to enable strict authorization and protection rules. This results in the above rules, as data is only accessible via the public API of the module that implements the domain specific protection mechanisms.

Despite what seems to be a hard limitation, such a paradigm offers various advantages at the level of data protection and for software life cycle management.

As part of the eHF Development Environment the definition and enforcement of the public API of modules is supported.

4.3 Componentization

While modularization is a coarse-grained approach to partitioning your system into functional units, componentization is another level of separating functional concerns.

In the eHF architecture components are defined as elements a module is composed of. Components normally belong to a distinct layer of a layered architecture.

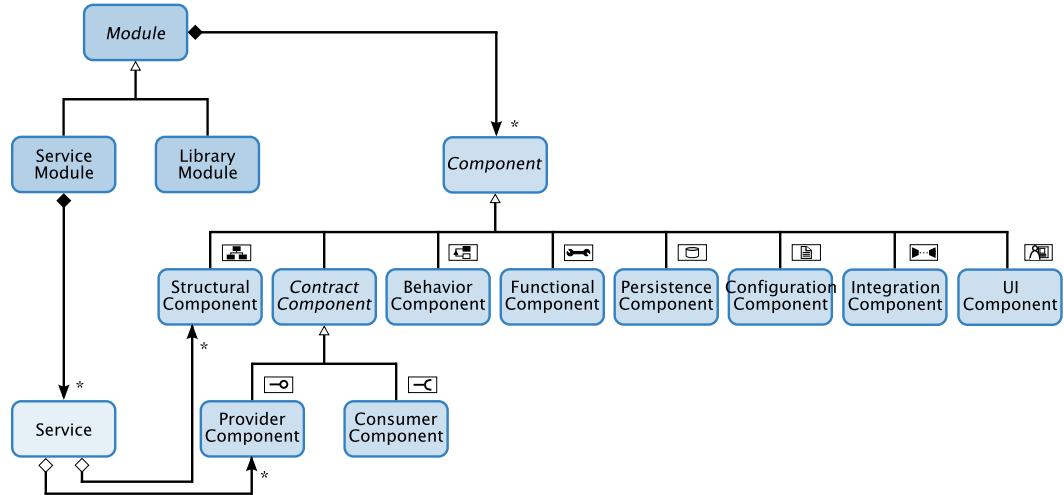
Please note that the term component is highly overloaded in software development. The definition used in the scope of the eHF refers to functional units of a fine granularity. In many contexts the term can mean very coarse-grained functional units (which we would refer to as subsystems) or even bits and pieces of hardware.

The following table is a classification of components as defined in the context of the eHF.

	Structural Component: All kind of data structures are considered structural components.
	Behavior Component: Abstract base classes which define generic behavior and can be implemented from other modules.
	Configuration Component: Configuration files or configuration fragments of any kind.
	Functional Component: Self-contained units (utilities) which may be re-used from other modules if exposed.
	Persistence Component: DAO implementations or other persistence layer elements.
	Contract Component - Provider: Defines a contract.
	Contract Component - Consumer: Defines a required contract. This can be also understood as extension point or service provider interface (SPI) consumer.
	Integration Component: Component providing integration portions. Integration components can be combined to address an integration scenario.

4.4 Module and Component Meta Model

The eHF architecture provides functional separation using a decomposition in modules and components. In order to illustrate the relationship between modules and components the following meta model was derived. The meta model and the pictograms that are used provide a powerful tool to describe the responsibilities, boundaries, relationships and interactions between modules.

**Figure 12:** eHF Module and Component Meta Model

[Figure 12](#) shows the eHF module meta model. According to the figure a module consists of several components. Modules are either library modules or service modules.

Library Modules

A library module provides an API or a set of abstract base classes with well-defined extension points that provide a common implementation basis for other modules. A library module can define service interfaces specified in the context of an API which can be implemented. It also offers APIs encapsulating access to basic library functions.

Library modules do not require a database and do not expose an external (web) API. Inside the eHF, library modules can be found in the core, security, and infrastructure domains. For example, library modules are responsible for common tasks such as authentication, messaging, or transformation.

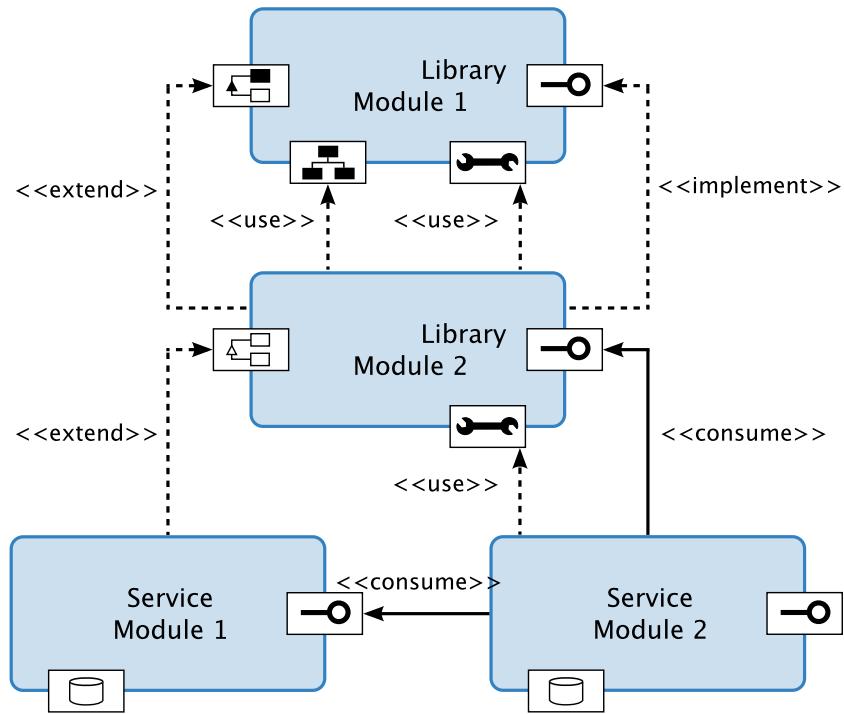
Service Modules

Service modules are self-contained and isolated functional units. They are more complex components which encapsulate a specific localized domain and may expose their functionality externally as a web service or internally to other modules or any other software. A service consists of a provider component and structural components (parameter and result types).

Service modules can use functionality from both library modules and other service modules. They can implement business logic, which requires services of other service modules. Thus a complex business process provided as a single service can actually be a collaboration of several services from different service modules.

Module Interdependencies

[Figure 13](#) shows the two module types and some possible interdependencies. Modules can extend other modules (behavior components), can use other ones (structural or functional components), can implement or consume them (consumer component) or can provide functionality to other ones (provider component). However, there is a constraint that service modules cannot be extended.

**Figure 13:** Module Types and Dependencies

Modules within the eHF either interact synchronously via procedure calls (request-response) or asynchronously via one-way messages. Message-based interactions are primarily used to communicate application events. Messages are sent via messaging services provided by the eHF Application Platform.

4.5 Application Meta Model

Basically an application consists of a set of service modules, library modules and third-party modules that are composed using additional configuration components. This is illustrated in [Figure 14](#), the meta model of an application. The configuration components specify how the modules interact with each other. Furthermore, application specific behavior of the modules can be defined on this level.

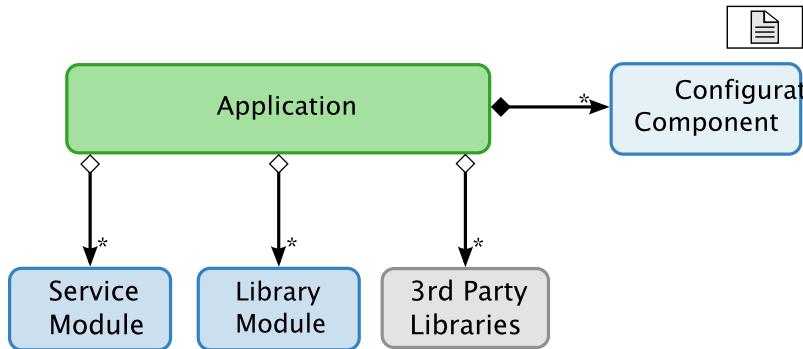


Figure 14: Meta Model of an eHF-based Application

The eHF Development Environment provides tools to define such an application. In this context we call a project to define an application assembly. The result of an assembly build or release is a distributable artifact that contains all the elements of the application.

Figure 15 shows a schematic instance of an application. The application exposes the web service provided by the modules. The web service of *eHF Service Module n* is specific to the application as it implements the desired service that the application requires.

eHF Service Module 2 consumes the service of a third-party module or service. The diagram also indicates that *eHF Service Module n* does not have any persistence. This normally means that it is only a facade to existing modules with the only focus being to implement an application-specific use case. The logic that is required is represented by the wrench pictogram.

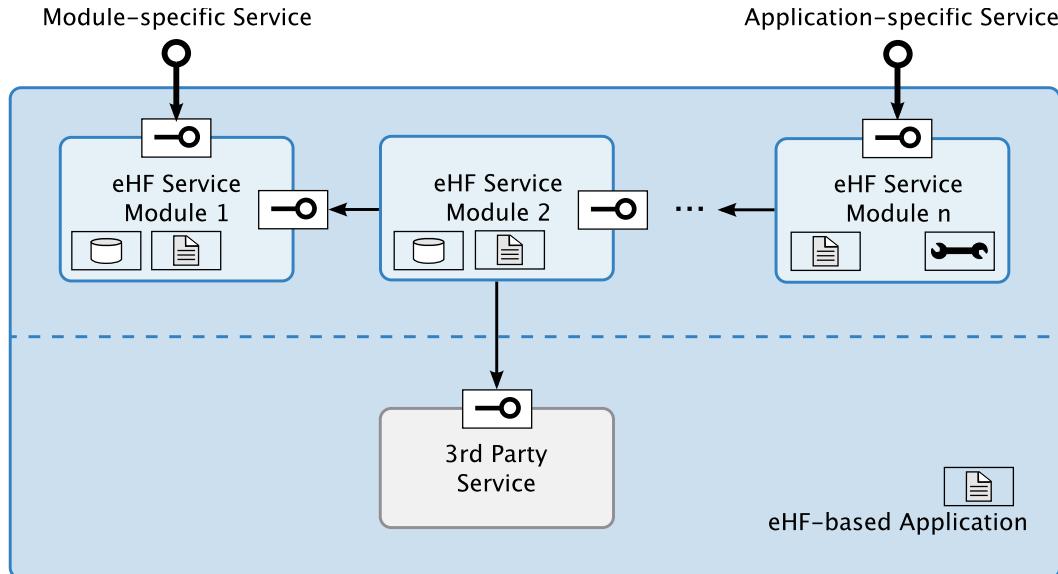


Figure 15: Example of an eHF-based Application Using the Component Pictograms

4.6 Architectural Layers

Next to modularization, layering constitutes the second cornerstone of the eHealth Framework's (eHF) architecture. While modularization focuses on functional, vertical (de-)composition, layering enforces a responsibility-driven, horizontal structuring of the software system. A module typically stretches across many, but not necessarily all layers.

Layering is one of the most common techniques used for structuring complex software systems. Each individual aspect of the software system is conceptually assigned to one of these layers. The dependencies between the different components are organized in such a way that only parts on higher layers may use parts on the same or lower layers. These layers have well-defined interfaces and controllable access to other layers of the system, thereby isolating technical details of adjacent layers. This layered architecture also allows for the possibility to upgrade or replace one or more tiers independently as requirements or technology change.

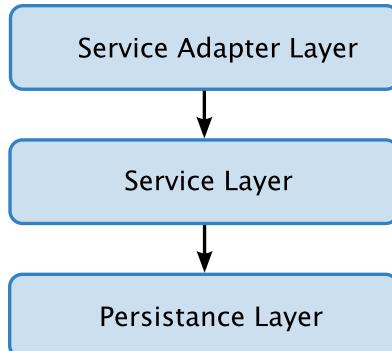


Figure 16: eHF Layers Overview

Conceptually the eHealth Framework's service stack consists of three layers; persistence, service and service adapter (see [Figure 16](#)). In the persistence layer data is stored and retrieved. It keeps data neutral and independent from the business logic found in the service layer, which is exposed to clients by the service adapter layer.

Persistence Layer

For the persistence layer the eHealth Framework uses the Data Access Object (DAO) Design Pattern to build a Data Access Layer (DAL). The service layer uses this DAL to access and manipulate stored data objects without having to deal with the complexities of database access. The DAOs define the interface between the service layer and the concrete data sources. They hide persistence technology details and decouple the business layer from changes in the persistence layer.

Service Layer

The service layer is an architectural layer that separates the business logic from other components like the persistence layer or presentation layer. It aggregates the business logic in a central place, which promotes consistent behavior across different applications. In addition it hides complex logic from application clients ensuring an easier, faster and less error-prone development process.

The service layer accepts requests from the service adapter layer and interacts with the persistence layer in order to map the business objects to the database.

Service Adapter Layer

The service adapter layer provides well-defined interfaces for integration. It decouples the service layer from the presentation layer or from other applications that are using the eHF services.

4.7 Integration

This section explains some of the concepts and terminology that are fundamental to IPF and Camel. The usage of IPF inside of eHF offers a flexible and consistent approach to integrate with different applications on different platforms and protocols.

Services and Endpoints

In order to integrate eHF based applications with other applications it is essential that a service is provided and exposed to the outside. The term service is very abstract and contains no information about the mechanics of how the service is exposed. In general, a service refers to a well-defined function of a service provider that is universally available and responds to requests from service consumers.

Another term that is closely related to a service is the concept of an endpoint. While the term endpoint is used in different contexts slightly different, all meanings have in common that they refer to a software entity that can be addressed. In Camel, the term endpoint is used in the area of routes and route definitions. A route starts with an incoming endpoint and may use several outward facing endpoints consuming either external functions or further internal routes. That is, a route is addressable through an endpoint. Camel provides out-of-the-box support for endpoints implemented with different communication technologies. In context of using Camel in eHF, a service refers to a collection of endpoints that have a manifestation towards the outside of an application.

Routes

Routes are the elementary building blocks of any integration solution implemented with IPF inside of eHF. A route definition contains the wiring between endpoints and processors to form routing rules. Any eHF module is able to define its own routes. In order to be globally addressable, the routes have to be contributed to a central Camel context. A Camel context represents the Camel runtime system. Typically, there is one globally configured Camel context to which the modules can contribute their route definitions.

Routing rules perform a certain task at which some tasks can recur in different use cases. One obvious example of route recurrence occurs in situations where services expose their functionality through multiple protocols. In this case, it is often possible to divide the route logic into a technical and domain-specific part. The technical route logic is responsible to adapt a certain protocol. The domain-specific logic contains logic that is tied to the domain. Instead of mixing domain-specific and technical routing logic it is recommended to separate the logic. The main benefit is that the domain-specific routing logic can be reused once the domain logic is factored out into its own route. When connected routes are splitted into multiple parts, the connections can be rebuilt by using local endpoints. A local endpoint in Camel is an endpoint that allows the direct invocation of a consumer endpoint in a route from a producer. This kind of modularization works only under the assumption that routes are registered within the same Camel context.

5 Architecture and Life Cycle Realization

The chapters on [Central Concepts](#) on page 8 and [Conceptual Architecture](#) on page 25 set the tone, scope, and requirements for the eHealth Framework (eHF). They were deliberately kept technology and implementation-neutral.

This chapter explains how the eHF realizes the proposed architecture, which core technologies are employed, and how it manages the software life cycle.

5.1 Platform Architecture

The eHealth Framework (eHF) as a platform binds together various modules to cohesively form one application, called *assembly* in eHF. These modules can be (and usually are) both eHF's own framework modules and user-implemented, custom application modules. The platform's only job is to ensure that all modules fit together nicely and operate according to their declared contract.

The eHF employs the Spring Inversion of Control (IoC) container to harbor all modules and uses Spring's dependency injection (DI) mechanisms to bind them together. These two core parts of the Spring Framework are the only concrete technologies imposed on module developers. They are otherwise free to choose implementation techniques as they please. It is due to this fundamental dependency on the Spring framework, however, that any eHF developer should have a good understanding of the capabilities of Spring IoC and DI. It is key to successful development with the eHF platform. See <http://www.springframework.org> ▾ for more information.

As in a typical Spring-based application, each module of an eHF-based application defines a number of Spring beans that fulfill the module's purpose. But whereas standard Spring maintains a global application context, with every bean being visible and accessible, eHF strictly enforces module boundaries by introducing module-specific Spring contexts. That is, every module is free to declare arbitrary Spring beans, but all of them are internal to the module and not visible for consumption by other modules. Beans that are considered public must be declared and exposed explicitly in the module's context. In the same manner, if a module consumes beans of other modules, these must explicitly be imported.

To illustrate this concept, let us take a look at a module's Spring context declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ehf="http://www.intercomponentware.com/schema/ehf-commons"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.intercomponentware.com/schema/ehf-commons
           http://www.intercomponentware.com/schema/ehf-commons/ehf-commons.xsd">
    <ehf:module id="moduleId">
        <ehf:configLocations>
            <ehf:value>
                classpath:/META-INF/ehf-module-runtime-context.xml
            </ehf:value>
            <ehf:value>...</ehf:value>
        </ehf:configLocations>

        <ehf:export>
            <ehf:bean id="myBean">
                <ehf:interface name="java.util.List"/>
                <ehf:interface name="java.lang.Iterable"/>
            </ehf:bean>
        </ehf:export>
    </ehf:module>
</beans>
```

```

</ehf:bean>
<ehf:bean id="otherBean" interface="..." />
<ehf:bean ... />
</ehf:export>

<ehf:import>
<ehf:pattern>transactionManager</ehf:pattern>
<ehf:pattern>...</ehf:pattern>
</ehf:import>
</ehf:module>
</beans>

```

The eHealth Framework defines its own namespace for the configuration of a module's Spring context. Items under the `ehf:configLocations` tag point to regular Spring configuration files, where a module's beans are declared. A subset of these beans are listed under the `ehf:export` tag, which makes them, and only them, the public service API of a module. Should the module consume beans of other modules, these must be listed under the `ehf:import` tag. At runtime, the eHF platform instantiates specialized Spring context instances that make sure that all modules contained in an assembly adhere to their declared module boundaries.

These simple, but effective measures built on top of the powerful Spring platform ensure that many of eHF's goals and concepts are achieved and implemented: Use of POJOs and IoC, modularity with public contract enforcement, and testability are all enabled by this setup. Moreover, consequent use of Spring-managed beans offers easy introduction of cross-cutting concerns by applying advice through Spring's AOP facilities.

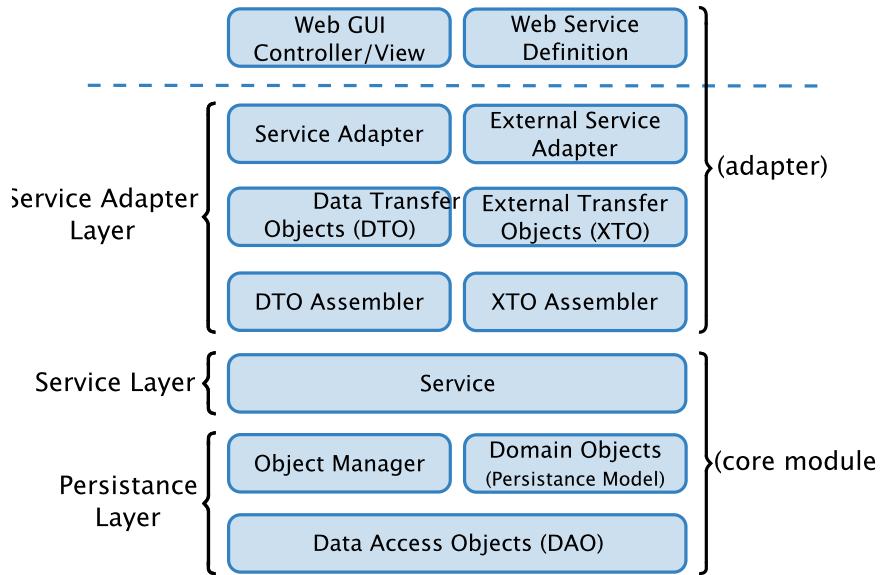
5.2 Standard Service Module Architecture

As a module developer, you are more or less free in your choices of how you implement your module. However, you will have to keep in mind the overall architectural layers (see [Architectural Layers](#) on page 31), a module's decomposition into components (see [Module and Component Meta Model](#) on page 28), and the provisioning or consumption of services.

While there are many ways to fulfill these requirements, the eHealth Framework (eHF) proposes a standard service module architecture to guide you in implementing a module. It is highly recommended that you make use of this standard architecture, since then you will benefit the most from the tools and libraries provided by the eHF. Following this standard path also helps you avoid mistakes, since many detailed design decisions have already been made with all architectural and non-functional objectives in mind. And finally, all modules shipped with the eHealth Framework follow this standard architecture. So when taking a look into these modules, you will have no trouble understanding their code.

With all this being said, please note that the architecture presented in this chapter applies in its entirety to service modules only. Since library modules must not provide persistence or services of their own, these parts of the architecture are not directly applicable to them. However, they may, and typically do implement certain parts of the architecture for consumption and re-use by service modules.

"Zooming in" on the high-level three-layer architecture from chapter [Architectural Layers](#) on page 31, the eHF standard service module architecture presents itself in the following way:

**Figure 17:** Standard Service Module Architecture

The following sections describe the architectural layers in more detail. Note that the eHF Commons module (see [Commons](#) on page 121) provides abstract and base implementations for almost all parts of the architecture. In most cases a module merely has to specialize these parts and complement them by module-specific code.

5.2.1 Persistence Layer

The persistence layer is built upon and around a module's domain objects. They represent its actual, persistent data model. Together with the Object Manager, they act as an abstraction layer to internal data sources, which are accessed using Data Access Objects (DAOs). The standard service module architecture employs Hibernate (see <http://www.hibernate.org>) as the de-facto standard Java persistence solution. However, module developers are free to choose any other JEE-compliant persistence approach instead.

Domain Objects

The domain objects are at the heart of any service module. They represent the data structures that the business logic is built upon, and thus act as an interface between the persistence layer and the service layer. The domain objects themselves, however, are mostly devoid of any program logic. Functional responsibilities are carried out by other components.

Data Access Objects

Data Access Objects (DAOs) are responsible for all persistence oriented operations - creating, updating, deleting, and retrieving domain objects. They provide access to a particular data resource using an abstract interface, thereby decoupling the resource's API from the business logic, while at the same time providing specific operations without exposing details of the database.

Object Manager

The object manager deals with persisting object graphs (in contrast to single objects). It offers methods to check the integrity and the overall consistency of a set of domain objects. It is a very central access point for the service layer and is described in more detail under [Object Manager](#) on page 121.

5.2.2 Service Layer

The service layer implements the business logic of an eHealth Framework service module. It operates on the domain objects, and uses the data access objects and the object manager for persistence. This is where the actual functionality of a module's business cases are implemented.

The service layer is considered a private artifact of a module. Instead, select parts of it are publicly exposed through the service adapter layer.

5.2.3 Service Adapter Layer

The service adapter layer exposes a module's public API to clients (for example web GUIs) and external applications. It hides the internal implementation of the service layer through adapter interfaces, and represents domain objects with specialized transfer objects.

There are two types of service adapters. One provides the internal API of a module. This is typically used by the presentation layer or other modules that are collocated with your own module in the same VM. The other adapter provides a web service API for consumption by remote clients and other applications. Each kind of adapter comes with its own set of transfer objects.

Apart from technical details, the main difference between these two adapters is the scope and extent of their API. While the internal service adapter may provide fine-grained access to the domain objects, the external service adapter is restricted to coarse-grained, compound business operations in order to minimize the number of costly remote service invocations.

Service Adapter

The service adapter represents the API of a module. The internal service adapter consists of one or more Spring beans that are exposed in a module's Spring context. They can be consumed by any other eHF module within the same global application (assembly) context.

The external service adapter is a web service API for the module. In the standard service module architecture, the Apache Axis framework (see <http://ws.apache.org/axis/>) is used to expose the service.

Transfer Objects

A transfer object is a container for data exchange between client or external applications and the services on the service layer. It may represent data from different business components and therefore may enable the number of remote method calls over the network to be minimized. Both adapters always operate on transfer objects and not directly on the domain objects. The service adapter for internal access operates with data transfer objects (DTO) and the external service adapter with the external transfer objects (XTO).

Assembler

The assembler is responsible for converting domain objects to transfer objects and vice versa and thus decouples the service layer from the service adapter layer. Since the data structures of the domain and transfer objects may not always be identical, the assembler may have to map the data between these two structures. This allows for optimized views of the domain model that are better suited for remote communication. Each service adapter comes with its own assembler - the DTO assembler for the internal service adapter, and the XTO assembler for the external service adapter.

5.2.4 Services

The eHealth Framework (eHF) architecture mandates that modules expose their capabilities as services. In the standard service module architecture, there are two variants of these services: object (CRUD) services and module services. The former are bound to one domain class and typically perform create, read, update, and delete (hence CRUD) operations, although they can be extended by custom services. The latter are bound to the module as a whole and provide coarse-grained business/domain operations. There is only one such service per module. This chapter takes a more detailed look at both kinds of services.

5.2.4.1 CRUD Services

Every domain object can expose a CrudService that basically reflects the features of its data access object (DAO) for creating, reading, updating and deleting instances of that domain object. Moreover, a CrudService in the standard service module architecture also exposes methods for adding and deleting instances of associated domain objects. Finally, a CrudService may also provide custom, domain object-related methods.

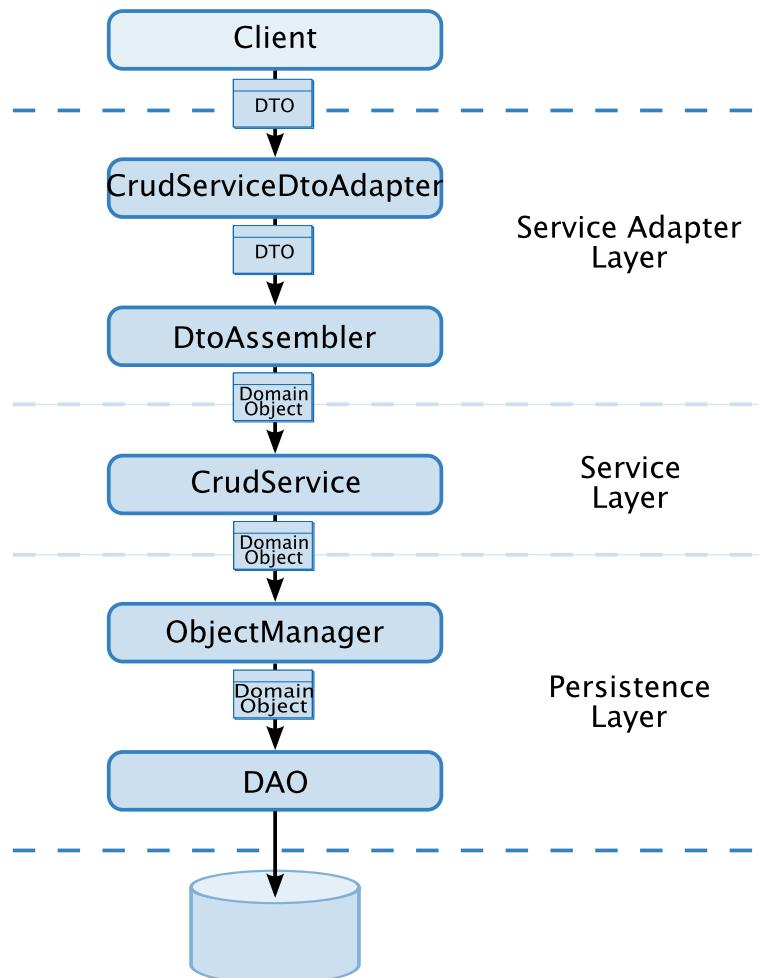


Figure 18: CRUD Service Stack

Figure 18 shows the architectural stack of a CRUD Service. Calls to the adapter layer are forwarded to the **CrudService** implementation in the service layer, with the object assembler converting transfer objects to domain objects (and reverse for the operation's return value). The **CrudService** uses the **ObjectManager** of the persistence layer for performing persistence operations. This, in turn, will use the domain object's DAO to access

the data source. Throughout this process, transactional aspects are applied on the adapter level, and security aspects operate on the service and domain object level.

It is important to note that there is no external service adapter for CRUD services. This is because these services would be too fine-grained for remote calls. It would increase the number of calls and thus have a negative impact on the overall system performance. CRUD Services therefore only have an internal service adapter which is called CrudServiceDtoAdapter.

For a typical domain object, the CRUD service stack in the standard service module architecture consists of the following classes in their respective layers. Note that most of these classes are represented by both an interface and an actual implementation class.

- Service adapter layer
 - DTO service adapter
- Service layer
 - Service
- Persistence layer
 - Object manager
 - DAO
 - Domain object

5.2.4.2 Module Services

Module services are coarse-grained services that typically operate on a set or a graph of domain objects. They typically implement complex business logic operations in a module's domain. As opposed to CRUD services, module services are exposed by both an internal and an external API, thus allowing web service access for remote clients.

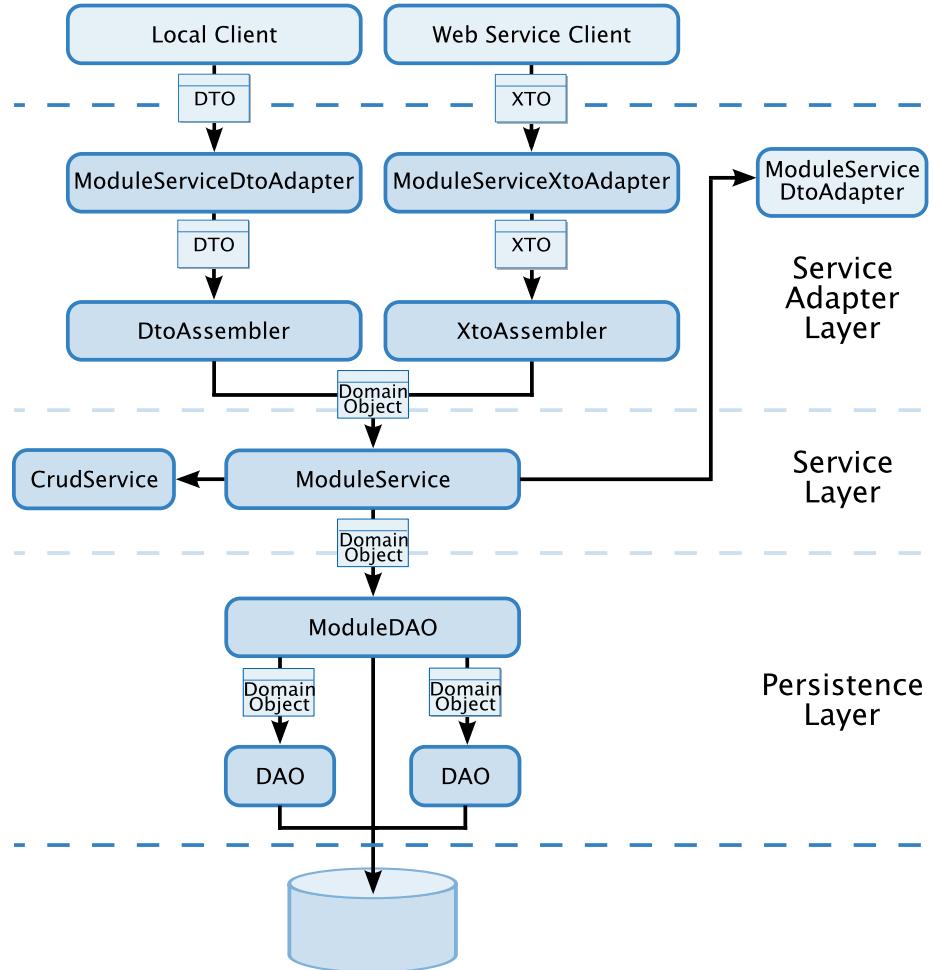


Figure 19: Module Service Stack

Figure 19 shows the stack of a module service within the standard service module architecture. Incoming calls on the adapter layer (either internal or external) are routed into the service layer, with the object assembler (DTO assembler or XTO assembler respectively) taking care of transfer-to-domain object conversion.

The `ModuleService` implementation may directly use any CRUD services inside its own module. Other module's services must be invoked through their `DtoAdapter` API. The module service may employ its own `ModuleDao` for database operations. This, in turn, may want to use its module's domain object DAOs in some cases. As with CRUD services, transactional and security aspects are applied on various levels.

5.3 Event processing

This section outlines the different components in eHF that constitute the event processing infrastructure and how you can make use of it.

An application usually needs to inform interested parties about certain processes with the help of events. This provides a loose coupling between the component that generates these events and the components that are interested.

The infrastructure for event processing in eHF is based on IPF. The definition of the processing chains is done in the form of an IPF route definition. This enables a very

flexible way to process events and integrate different event consumers and producers. The infrastructure has three responsibilities:

1. Processing of application events and providing extension points where eHF components interested in these application events can hook in (like the eHF Audit or a component which gathers statistical data).
2. Publishing application events to external interested components (like an external ATNA audit repository).
3. Integrating components and their events that are not compliant to the eHF event processing infrastructure (like an IHE XDS Actor that is only capable of sending ATNA events).

This is illustrated in [Figure 20](#) which shows all three use cases combined.

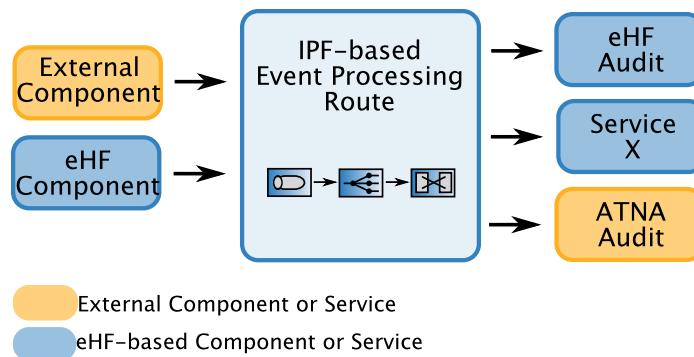


Figure 20: Different event processing usecases combined

The central element of this infrastructure is the IPF event processing route (see `EventProcessingRouteBuilder.java` in your assembly for this). By default you can find the starting point of the process chain here, which is by convention the endpoint `direct:platformEventProcessingEndpoint`. You have to add the building blocks of your application's event processing. It is important to emphasize that each application has its own needs and therefore you have to complete the default definition of this processing chain according to those needs yourself.

The default definition of the `EventProcessingRouteBuilder` looks like this:

```

1 public class EventProcessingRouteBuilder extends SpringRouteBuilder {
2
3     @Override
4     public void configure() throws Exception {
5         errorHandler(loggingErrorHandler());
6
7         from("direct:platformEventProcessingEndpoint");
8     }
9 }
```

This basically just provides the endpoint of the route. Actually this is only a starting point as up to now nothing has happened. What could a complete implementation of a route look like?

As an example we use writing to eHF Audit as one of the use cases to illustrate how you can build your event processing with the help of IPF and the underlying Apache Camel technology.

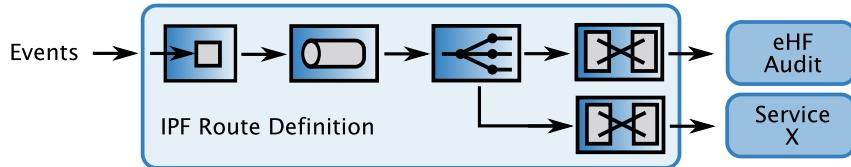


Figure 21: Concept of a route that provides application auditing

Conceptually an event processing route that would write into audit would look like [Figure 21](#). Events come in via the endpoint, are placed into a queue and from there they are distributed to different recipients. Afterwards the events are transformed and then sent to the audit module. The recipient list in the middle of the route allows you to hook in other components that are interested in the different events. In this example we use an imaginary service X.

The route definition for this concept would then look similar to this:

```

1  @Override
2  public void configure() throws Exception {
3      errorHandler(loggingErrorHandler());
4
5      from("direct:platformEventProcessingEndpoint")
6          .to("seda:eventQueue?concurrentConsumers=10"
7              &waitForTaskToComplete=Never");
8
9      from("seda:eventQueue?concurrentConsumers=10&waitForTaskToComplete=Never")
10         .to("direct:auditTransform", "direct:serviceXTransform");
11
12     from("direct:auditTransform")
13         .process(transformationAdapterProcessor)
14         .to("direct:audit");
15
16     from("direct:serviceXTransform")
17         .process(serviceXTransformationProcessor)
18         .process(serviceXProcessor);
19 }
  
```

In line 6 the SEDA component from Apache Camel Core is used as a queue. This is a lightweight in-memory queue. For reliable messaging you could use the JMS or ActiveMQ component instead (see the documentation of Apache Camel for more details).

In lines 9 to 10 the events are read from the queue and distributed to all endpoints mentioned in to-clause. It represents the recipient list from the conceptual structure of the route above. This is the extention point to add more components that want to be notified of the events.



Tip: For more information on how to build routes see the IPF and Apache Camel documentation.

In order to send events from an eHF-based module to this route you can use a platform bean called `eventMessageSender` for your convenience. This bean implements the interface `MessageSender` and places your events into the event processing route.

5.4 Software Life Cycle Support

The chapters [Platform Architecture](#) on page 34 and [Standard Service Module Architecture](#) on page 35 have shown how the eHealth Framework implements a powerful and flexible runtime architecture that meets the goals and requirements defined in chapters [Central Concepts](#) on page 8 and [Conceptual Architecture](#) on page 25. However, these requirements mandate that the eHF is to be an holistic approach for developing and maintaining health care solutions, including software life cycle support. These life cycle issues are the focus of this chapter.

Glancing at the diagrams in chapter [Standard Service Module Architecture](#) on page 35 it is obvious that a standard service module may consist of a considerable amount of code. Most of this code however, apart from the domain model definition and the implementation of a module's services, is concerned with infrastructure and architectural concerns. eHF therefore applies domain-driven design (DDD) and model-driven software development (MDSD) principles to let the developer focus on the domain-specific parts of a module.

The eHealth Framework contains a generator based on the openArchitectureWare (oAW, see <http://www.openarchitectureware.org>) framework that is capable of generating basically all components of the architecture diagram shown in chapter [Standard Service Module Architecture](#) on page 35. Only the service layer implementation is left for the developer, albeit it is also possible to implement other (usually generated) parts of the module manually.

The generator receives a domain model as input. That model is a standard UML2 model, enriched by an eHF-specific profile. In theory, any UML2-compliant modeling tool is capable of providing such a model. eHF currently supports MagicDraw and the TopCased editor for this purpose.

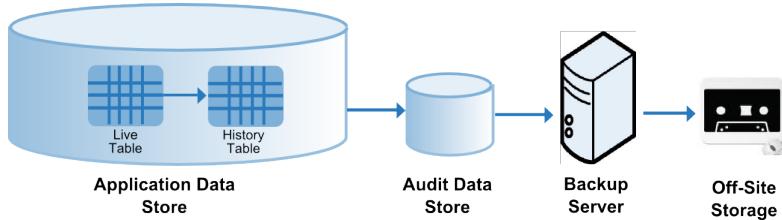
For typical development life cycle tasks, the eHF provides a sophisticated build infrastructure based on Maven (see <http://maven.apache.org>). Each module and assembly is defined as a separate project, with Maven managing all dependencies and performing all everyday build steps. This includes code generation, test case execution, database management, and packaging and configuration of distributable artifacts (typically eHF-based applications).

Once an application is delivered, tedious and error-prone installation steps at the customer's production site are simplified by a number of installation scripts that are based on Ant (see <http://ant.apache.org>). Ant is much leaner than Maven and its usage avoids the necessity of installing a (possibly heavyweight) Maven infrastructure at the customer's site. The scripts also support update and upgrade procedures.

All in all, these measures cover the entire software development life cycle of state-of-the-art health care solutions.

5.5 Information Life Cycle Support

All information has a lifecycle. It begins when information is acquired and ends when it is no longer needed and can be deleted. During the lifetime, data is moved between higher performance (e.g. production database) and lower performance storage medium (e.g. tape library) depending on its access rates (e.g. frequently or rarely accessed) as shown in [Figure 22](#).

**Figure 22:** Information Lifecycle

Health applications are concerned with sensitive data. Governmental and industrial regulations mandate building an audit trail which keeps a complete history of all changes made to the data. An audit trail allows for tracking who changes which data, when and in which way (e.g. creation, modification and deletion). In eHF, an audit trail can be constructed using historical data in history tables and audit entries in audit tables.

Backups are primarily designed to restore an entire database. It is best practices to schedule backups at regular intervals and ship backups off-site to prevent data losses in case of hardware failure, human error, natural disasters, etc. Often a backup is also scheduled before a planned database or application upgrade. In case the database is corrupted during the upgrade, it can be rolled back to the time before the data was modified. A backup consists of all data in the production database including live tables, history tables and audit tables.

Enterprise databases may grow with high pace over time. Overloaded databases degrade application performance and availability. However, enterprise data should not be destroyed arbitrarily. Instead, they must be retained for years while being easily accessible to comply with regulatory retention requirements. Database archiving allows for removing rarely accessed data from production databases and storing them on a variety of storage medium (e.g. tape drives) while providing easy access. Regularly scheduled database archiving frees up significant amounts of disk capacity and processing power to improve application performance and availability.

5.5.1 Building an Audit Trail

A history table is required for each table for which audit trails are needed. In eHF, the generator automatically generates database scripts to create database schema for domain objects specified in the model. By default, the generator generates database schema only for live tables. If explicitly specified, the generated database schema also includes definitions for the corresponding history tables. The `maven.ehf.db.schema.generate.history.tables` property is used to specify whether history tables are needed. By default, the property is set to false and thus no history table is generated.

For each domain object, its unique identifier serves as the primary key in the live table storing its current status. In the history table there can be multiple rows storing the complete change history of a single domain object. Apart from this difference, the history table has the same structure as the associated live table.



Note: From time to time the structure of a live table can change. Upgrade scripts (e.g. consisting of a sequence of `alter table` statements) must be created manually to upgrade the existing database using the eHF upgrade process. Keep in mind to upgrade the associated history table accordingly.

The generator creates BEFORE DELETE OR UPDATE triggers to automatically insert data into history tables before they are updated or deleted in the live table. As illustrated in [Figure 23](#), before the data of version vt1 is updated to version vt2 in the live table, version vt1 is inserted to the history table. In this way, history tables do not duplicate data in live tables. All current data are stored only in live tables, while history tables store only historical data.

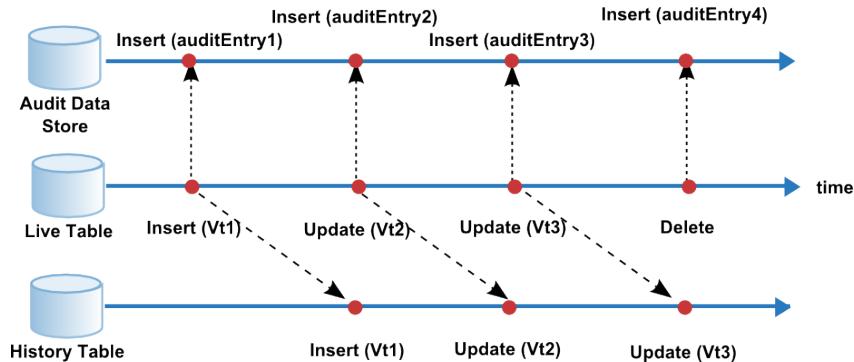


Figure 23: Building an Audit Trail with History Table and Audit Entries

Each eHF domain object has the `creatorId`, `createDate`, `changerId` and `changeDate` attributes. They are persisted in the database. Thus, history tables keep the information about who changes which data and when.

Apart from persisting historical data, eHF supports auditing CRUD operations on domain objects and application-specific events (e.g. authorization events or transaction rollback events). The action performed on the data and the prevailing contextual information are recorded as audit entries in audit tables.

Audit entries and history tables must be used together to construct the complete change and access history.

In eHF, the following eHF Modules have history tables:

- eHF Authorization: Permission information is subject to historize. A typical usage scenario is to show permissions for a user at a given time.
- eHF Record Medical: Changes to personal health records are subject to both historization and auditing. A typical usage scenario is to verify whether user B was authorized to perform certain modifications on records owned by user A according to his/her permissions at a given time.



Note: eHF Record Admin does not use history tables, since demographic data shall not be historized.

5.5.2 Deletion and Retention Requirements

Deletion and retention requirements are specific for each country. In Germany, the following regulations apply for the deletion and retention of history and audit data:

- Audit entries and history data must be retained for the reclamation period of 6 months in the production database, before they are moved to backup medium.
- Once a customer cancels his/her contract, there is no legal background for the storage of his/her data. Hence, the data in live tables must be deleted immediately according to data privacy requirements. However, data in archives must be retained for 3 years starting from the time of cancellation.

Once application data are encrypted, the encryption parameters used to encrypt them (including verification data such as certificates and passwords) are liable to retention requirements as well.

5.5.3 Securing the Audit Trail

Securing an audit trail includes the encryption of history tables, audit tables, backups and archives.

Using application-level encryption (ALE), application data which is classified as sensitive are already encrypted in the application before they are inserted to live tables. Once the data in live tables is encrypted, it is encrypted in history tables as well.

If an existing database needs to be upgraded to initially enable ALE, to re-encrypt or decrypt particular columns as a consequence of changes in the domain model or cryptographic parameters, the upgrade process provided by the eHF Key Tools ensures that data in both live and history tables is upgraded (i.e. encrypted, re-encrypted or decrypted).

Using Oracle's TDE, sensitive data is specified in a TDE configuration file. This configuration is used by the eHF Key Tools to active TDE for the associated columns at build time. The database transparently encrypts the data. If a live table has a history table, it must be explicitly specified in the configuration to ensure the encryption of historical data in the history table.

Certain properties of an audit entry are encrypted in order to prevent data correlation attacks. In particular, the information about who made the change and which object has been changed is encrypted. Additionally, each audit entry is associated with one or multiple attributes, each being a name and value pair. The values are encrypted as well.

The security of database backups and archives is extremely important, because they typically contain sensitive data which is stored in one localized, compact and portable medium. Database vendors (e.g. Oracle) provides secure backup mechanisms using backup encryption. In eHF, thanks to application-level encryption, sensitive data is already encrypted before being backed up.

In case of LifeSensor, an ICW product based on eHF, backups are further encrypted for confidentiality by the application hoster.

Encryption parameters must also be included in the backup. The prerequisite for the security of backups and archives is the separation of keys and the data encrypted with these parameters.

5.5.4 Backup Encryption Parameters

With application-level encryption, eHF allows using dedicated keys to encrypt different data (e.g. data of different classifications, or data from different modules). The eHF Encryption Module persists information about how particular data is encrypted as receipts in the database. Each receipt consists of a reference to the encrypted data, a reference to the encryption key, and further encryption parameters such as the initialization vector and the cryptographic algorithm etc. A decryption is not possible without the matching receipt.

Using encryption, the task of protecting a large amount of sensitive data is reduced to the protection of a comparably small set of encryption keys. The security of the keys depends on two factors, namely where are the keys stored and who has access to them. In eHF, we use a chain of keys to protect the encryption keys:

- The Content Encryption Keys (CEK) which encrypt the payload data are kept in a password-protected keystore residing in the file system. The keystore is part of a key package which is an installation-specific folder consisting of important files for the encryption.
- The keystore file is protected by a password which is randomly generated by the master key provider. The password is referred to as the Primary Master Key.
- Each CEK in the keystore is further encrypted with a Key Encryption Key (KEK).
- As part of the key package, KEKs are further encrypted with a Secondary Master Key randomly generated by the master key provider. In a running application, KEKs are stored in the database of the eHF Encryption Module.

Best practices recommend separating application data from keys in files and databases, and distributing the responsibility among several administrators (see [Figure 24](#)). However, due to cost factors one single person often takes over the responsibility of several administrators, or intentionally separated data may be brought together in backups and archives.

The Master Key Provider has been introduced to solve this problem. It is hosted on a separate, dedicated system by a trusted third party other than the application hoster. Hence, the master keys are separated from the content and key encryption keys. Without the master keys, neither the keys nor the data can be decrypted. An application interacts with the Master Key Provider with certificate-based mutual authentication.

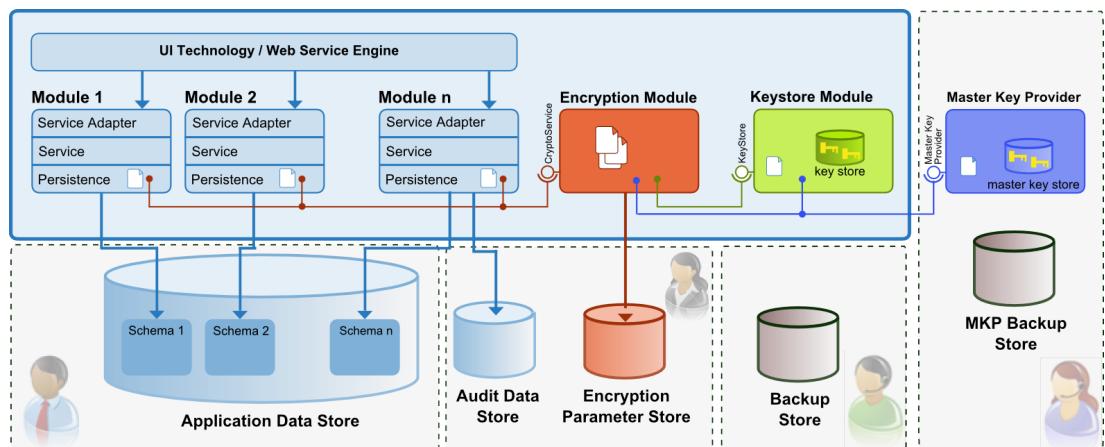


Figure 24: Separation of Responsibility

Due to the separation between application and Master Key Provider, backup of encryption parameters must be performed on both sides:

At the application side, the backup administrator must ensure the backup of

- the key package consisting of the keystore and KEKs,
- database consisting of encryption parameters, and
- the certificates used to interact with the Master Key Provider. In eHF, the application stores its private key in a keystore and the certificate of the Master Key Provider in a truststore. The passwords to access the keystore and truststore must also be backed up.

At the Master Key Provider side, the backup administrator must ensure the backup of

- the key repository consisting of the primary and secondary master keys,

- the passwords protecting the key repository,
- the certificates used to protect interactions with the client. The passwords to access the keystore (keeping the private key of the master key provider) and truststore (keeping the trusted client certificates) must also be backed up.

5.5.5 Accessing the Audit Trail

In general, the application (e.g. LifeSensor) is not meant to access the audit trail. Especially, the application is neither allowed to alter nor delete data in history and audit tables. This is ensured by the fact that eHF Modules do not provide a public API for accessing history tables, and eHF Audit does not provide web services to applications.

The following describes cases of authorized accesses to audit trails.

Audit the online history tables

Authorized auditors shall be allowed to query the online history tables along with audit tables using certain search criteria.

A dedicated audit process (similar to the support process for 3rd level support) allowing authorized auditors to submit queries to the encrypted database and view results in plaintext is required. The audit process directly accesses the database instead of through the application.

Delete data from online history tables after cancellation of contract

This case is currently not supported. History tables are retained for 6 months before they are moved to the archive which is retained for another 3 years.

Audit using the archive

This case is currently not supported. Like auditing the online history tables, a dedicated audit process directly accessing the database is required.

Delete history data from archive

Although requested by data privacy requirements, deletion from archives might be technically not feasible. For example, an archived data object may be stored on write-once media, along with other records that are not subject to deletion.

5.5.6 Long Term Archiving of Sensitive Data

Long term preservation of digital information is centered on the term "time": the lifetime of data requested by legal retention requirements, the life span of hardware and software used to store and process the data, the validity periods of cryptographic keys and certificates protecting the data, and the suitability periods of cryptographic algorithms for hashing, digital signatures and encryption. As computing and storage technologies advance, hardware used to store the data may become obsolete. Data formats and software used to interpret and present the data may cease to exist. Cryptographic keys may be comprised or certificates can expire or be revoked. Cryptographic algorithms used to prove the integrity and authenticity of data or to protect their confidentiality may become unsuitable due to advances in cryptanalysis and computational capabilities.

5.5.6.1 Requirements

The requirements for long term archiving (see [RFC 4810](#) and HKLW2009) are briefly described in the following:

- *Integrity*: Archived data objects must not be maliciously or accidentally altered, deleted or overwritten. The integrity of data retained must be ensured in a verifiable way, i.e. modifications to an archived data object must be detected.
- *Authenticity*: The authenticity of data is defined as the ability to establish its identity and validate its integrity. The identity of a digital record consists of the name of a file, originator, the creation time etc. The authenticity of data retained must be ensured in a verifiable way. Digital signatures of archived data objects can be used to prove both their integrity, authenticity and non-repudiation.
- *Confidentiality*: Any confidential information stored must be protected against unauthorized access by means of access control and encryption.
- *Availability*: The archived data must always be accessible by authorized entities. As system failures, natural disasters or human errors can not be excluded, a certain degree of redundancy is required.
- *Accessibility*: The archived data must be interpretable over very long periods of time, although the hardware and/or software originally creating them may become obsolete. Periodically migration of data to the successor technology and emulation of the original operation environment are means to ensure accessibility.
- *Legal compliance*: Legal compliance requires that data must be stored in a way such that the applicable legal requirements are satisfied. For example, if qualified electronic signatures according to the German Signature Act are archived, an integrity conservation mechanism with qualified timestamps must be used.

When archiving sensitive health data, the integrity and confidentiality of archives are of particular importance. The German Federal Office for Information Security (BSI) has developed a Technical Directive for trustworthy long term archiving. However, the directive focuses only on integrity and authenticity without providing guidelines for confidentiality.

5.5.6.2 Protecting integrity of archives

Protecting integrity of archives with timestamps

Timestamps can be used to protect integrity and authenticity of data. RFC 3161 specifies a [Time-Stamp Protocol \(TSP\)](#). It utilizes a Time Stamping Authority (TSA) to creates timestamp tokens which indicate that a datum existed (i.e. was created or last modified) at a particular point in time. The TSA signs each timestamp token using the private key of a key pair generated exclusively for this purpose.

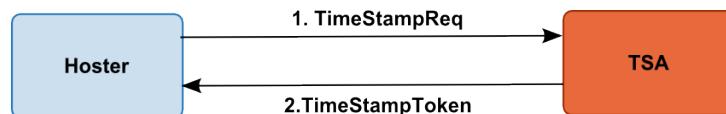


Figure 25: Time-Stamp Protocol

Figure 25 shows the interactions between an application hoster and a TSA participating in a Time-Stamp Protocol as specified by RFC 3161. The hoster requests a time stamp for the archive whose integrity is to be protected. The messages `TimeStampReq` and `TimeStampToken` are defined as follows:

1. $\text{TimeStampReq} = \{\text{hashAlg}, \text{H}(\text{archive}), \text{nonce}, \dots\}$
2. $\text{TimeStampToken} = \{(\text{hashAlg}, \text{H}(\text{archive}), \text{nonce}, \text{time}, \text{tsa}, \dots), \text{Sig}_{\text{tsa}}(\text{hashAlg}, \text{H}(\text{archive}), \text{nonce}, \text{time}, \text{tsa}, \dots)\}$

The request `TimeStampReq` consists of the identifier of the one-way and collision resistant hash function, the hash value of the archive, and a nonce to prevent replay attacks. Instead of the whole archive, its hash value `H(archive)` will be timestamped.

Upon receiving the request, the TSA appends the time `time` and its own identifier `tsa`, and signs the message with its private key. Whenever the archive is modified, the hash value included in the signed timestamp token will not correspond to the hash value of the modified archive. Thus, a modification can be detected.

Timestamps rely upon mechanisms that are subject to the same problems of long term archiving of signatures. The private key of the TSA can be comprised, the associated certificate can be revoked or expire, and the public key algorithm can become unsuitable over time. To counter this problem, timestamps must be renewed by obtaining a new timestamp that covers the original data and its previous timestamps.

[RFC 4998 Evidence Record Syntax](#) specifies the renewal process. An evidence is referred to as a piece of information that may be used to demonstrate the validity of an archived data object. In particular, RFC 4998 supports *archive timestamps* which can cover a single data object or a group of data objects. Groups of data objects are addressed using hash trees described by [MERK1980]. The leaves of the hash tree are hash values of the data objects in a group. A timestamp is requested only for the root of the hash tree. In the following, we only focus on single data objects. RFC 4998 distinguishes between timestamp renewal and hash tree renewal.

A *timestamp renewal* is necessary, if the private key of the TSA has been compromised, the public key certificate has been expired, or if the public key algorithm or its parameters used for the generation of the timestamps are no longer secure. An archive timestamp allows the verification of the existence of several data objects at a certain time. The following shows the archive timestamps ats_i at time i and ats_{i+1} at time $i+1$ respectively. TST is the function which creates a timestamp token as specified in RFC 3161 for the given data object or group of data objects. With a timestamp renewal, the archive itself does not have to be accessed.

- $ats_i = \{TST(H(archive))\}$
- $ats_{i+1} = \{TST(H(ats_i))\}$

The process of timestamp renewal generates an archive timestamp chain. It is a time-ordered sequence of archive timestamps, where each timestamp preserves integrity of the previous archive timestamp. In an archive timestamp chain, the same hash algorithm is used to create all timestamps.

A *hash tree renewal* is required if the hash algorithm used to build the hash tree loses its security properties. A new hash algorithm H_{new} must be used for the renewed timestamp. The following shows the archive time stamp at $i+2$, when the previous hash function H used to create the archive time stamps ats_i and ats_{i+1} becomes weak. The new timestamp generated with H_{new} includes both previous archive timestamps.

$$ats_{i+2} = TST(H_{new}(ats_0 || \dots || ats_i || ats_{i+1}))$$

Critics on signature renewal

The above signature-based approach has been implemented by the [ArchiSafe project](#) and [Open Text](#). There is no further known implementation. The significant cost of signing large amount of data and regular signature renewal must be justified.

Audit-proof archiving provides a less expensive alternative which stores data on WORM (Write Once Read Multiple) media.

Protecting semantic integrity

Migration activities on a regular basis are required for accessibility of long term archives. For example, a trustworthy administrator translates an Oracle 10g backup created at time i to an Oracle 11g backup at time $i+1$ (see [Figure 26](#)). When properly performed, the semantic integrity of the archive is preserved. Technically, the semantic equality of both archives is not always verifiable. Hence, migration activities must be carefully documented.

Over time, early backups (e.g. $\text{backup}(i)$) will become inaccessible. After a successful migration to a new generation of hardware/software they don't have to be retained. The timestamp at time $i+1$ shall cover both the translated backup ($\text{backup}(i+1)$) and the documentation of migration ($\text{metaInfo}(i+1)$).

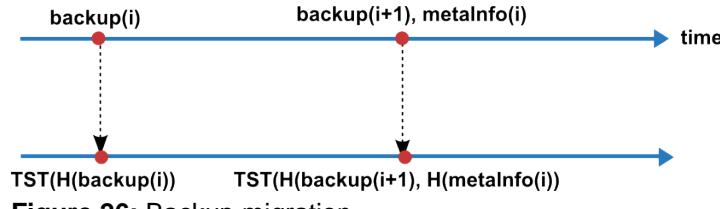


Figure 26: Backup migration

5.5.6.3 Protecting confidentiality of archives

Encryption-based methods

In eHF, application-level encryption provides initial protection of sensitive data. However, keys used for encryption can be compromised or the algorithms used for encryption can become weak. Data in the production database must be re-encrypted. Due to the potentially huge amount of data in an archive, a re-encryption of archive by the application is not an acceptable solution. Thus, we recommend an *over encryption* of the complete archive as illustrated in [Figure 27](#).

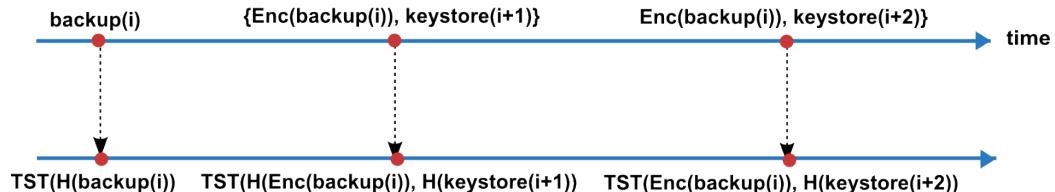


Figure 27: Backup Encryption

At time $i+1$, the original $\text{backup}(i)$ is encrypted using a key stored in the $\text{keystore}(i+1)$. We recommend deleting the original $\text{backup}(i)$, since its disclosure would disclose the sensitive data stored inside. At time $i+2$, the backup at time $i+1$ is decrypted and subsequently encrypted with a new key in $\text{keystore}(i+2)$.

If a signature-based approach is used to ensure integrity, the timestamp at time $i+1$ covers both the encrypted backup and the encryption key (which is a case of protecting integrity of a group of data objects).

Secret sharing based methods

Shamir's secret sharing algorithm provides an alternative to encryption-based methods for confidentiality. The idea is to divide the archive into N shares in such a way that the archive can be reconstructed from any K shares. Even complete knowledge of $K-1$ shares does not reveal any information about the archive. A secret sharing approach consists of the following steps:

1. Produce shares of the archive,
2. Distribute shares to different sites for storage, and
3. Retrieve K shares to reconstruct the archive.

A secret sharing approach tackles both the confidentiality and the availability requirement. Since any K out of N shares are required for reconstruction, a certain level of redundancy is

achieved. The larger N is, the more costly archiving is; the larger K is, the more impervious the system is to malicious attacks on data privacy, but the more costly data retrieval will be.

Both encryption and Shamir's secret sharing algorithm are computational intensive. However, the secret sharing algorithm provides information theoretically provable security, i.e. the security does not degrade over time.

5.5.6.4 Recommendations

Independently whether you look for an external Long Term Archive (LTA) service or implement your own LTA service, this section makes a few recommendations on activities of such a service:

- A LTA may establish authentication and authorization policies in order to decide whether an entity is allowed to submit, retrieve or delete archived objects.
- A LTA must periodically perform maintenance activities such as monitoring the accessibility of archive data and translating it from one format to another or migrating data from one storage medium to its successor.
- If cryptographic algorithms are used to protect the integrity and/or confidentiality of archives, a LTA must periodically perform activities that consider the strength of relevant cryptographic algorithms and life span of certificates, and possibly renew the associated evidence or perform a new encryption.

6 Conventions

Conventions and guidelines are a controversial topic. There is much debate about the sense and nonsense of conventions, and it is undoubtedly hard to have the right amount of guidelines for any particular project. Still, it is worth trying.

The bottom line is that conventions create some sort of uniformity across various parts of a large system. This uniformity helps you and others in understanding the system. Moreover, assuming that guidelines are correct, they can prevent you from doing things wrong.

In that spirit, the eHealth Framework (eHF) attempts to define a reasonable, yet not too rigid set of conventions, which are described in the following sections. Few of these are actually endorsed inside the framework and can theoretically be changed when writing your own modules or eHF applications. However, sticking to these guidelines is highly recommended.

6.1 Naming Schemes

Naming schemes, or naming conventions, help applications to be more understandable by making them easier to read. They can give information about the purpose, function or scope of an artifact or identifier – for instance, whether it is an API, runtime, or configuration artifact. This can be helpful in understanding the code. Furthermore, well-chosen naming conventions aid even the casual user in navigating larger structures. They help you to recognize patterns and to memorize the locations of artifacts.

The eHealth Framework (eHF) uses and reuses standard naming conventions whenever possible in order to not reinvent the wheel. This includes for instance standard [Java conventions](#) for formatting and naming source code artifacts, or [Maven 2 conventions](#) for structuring a module. It is assumed that you are familiar with these conventions, and that is why only naming schemes that are particular to eHF are being described. The aim is to better understand the overall existing module structures, to easily locate existing artifacts and their content, and to create new artifacts consistent with the naming schemes and conventions.

To better understand the eHF naming schemes, it is important to first understand their origins. Three basic framework components drive and define implicitly the naming schemes: the eHF Module-Template, the Build Plug-in (see [Build Plug-in](#) on page 70), and the eHF Generator (see [Generator](#) on page 88). The eHF Module-Template defines the structure of a module, which in principal, could be changed after its initial creation. The Build Plug-in requires certain artifacts in all modules to make a successful build. You can customize these artifact names. However, such changes are seldom necessary. The eHF Generator represents the backbone of many eHF concepts and influences the structure and naming conventions of generated artifacts such as Java code, resources, Spring configuration files, and web service deployment descriptors.

The eHF does not force you to use the naming conventions. Particularly, if you create new modules, or customize existing modules, you are not required to use the eHF naming schemes. You can override almost all default properties of directory and filenames. However, they are somewhat useful and highly recommended. Developers who are familiar with the eHF can profit from the uniform view and the advantage of better understanding as mentioned previously.

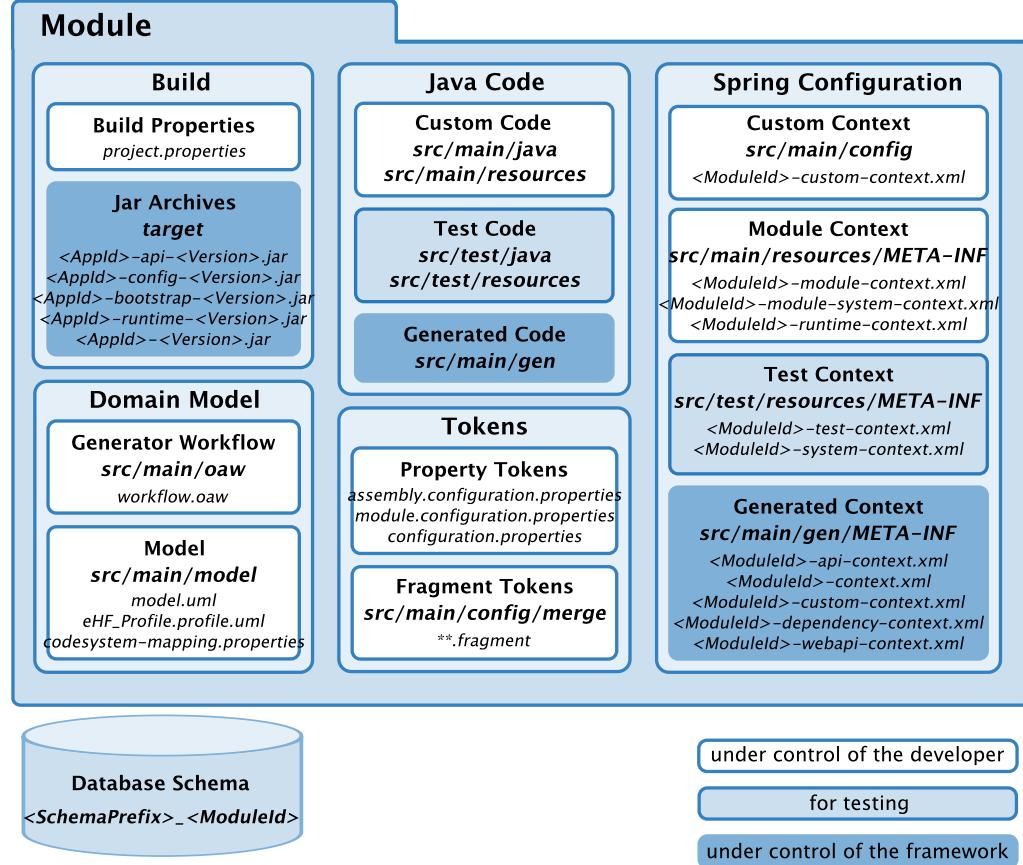


Figure 28: Overview of the Structure and Artifacts of a Module

Figure 28 provides an overview of the module parts, in which naming conventions occur. You can see their locations and their most important files. We will discuss them in the following sections.

6.1.1 Creating Modules

You most commonly create a new module with the help of the [Maven 1.x genapp plug-in](#) and the eHF Module-Template. The template uses the Maven 2 conventions for [structuring modules](#). That is why you find directories `/src/main`, `/src/test`, and `target` on the first directory level. The eHF configured template expects some inputs while generating. Table 4 lists these variables.

Variable	Default	Example	Description
<code>project root directory</code>	-	<code>C:\ICW_IDE\workspace\ehf-document</code>	The subdirectory in which the module is created
<code>application ID</code>	<code>ehf-module</code>	<code>ehf-document</code>	Artifact ID of Maven
<code>application group ID</code>	<code>ehf</code>	<code>ehf</code>	Artifact group ID of Maven
<code>application name</code>	<code>eHF Module</code>	<code>eHF Document</code>	Artifact name of Maven

Variable	Default	Example	Description
<i>application package</i>	com.icw.ehf	com.icw.ehf.document	Java package name
<i>module ID</i>	module	document	eHF module name
<i>schema prefix</i>	EHF	EHF	Database schema prefix

Table4. Naming Schemes for Creating new Modules

The *application ID* and the *application group ID* variables follow the Maven naming conventions of [artifacts](#). The *application package* is a valid Java package name. The *module ID* is an eHF specific ID, which has to be globally unique. That means, it is unique in the namespace of a packaged assembly. Additionally, the ID must not contain whitespaces or other special characters. The *module ID* is often used as a property token or as a part of a filename (for example Spring context definition files).

6.1.2 Building

When you create a module, you can configure the module-specific properties of the eHF Build Plug-in. Most of these have reasonable default values. However, you can override all properties in the module's `project.properties` file.

When building a module, the goal is to produce a distributable artifact in the form of a JAR file. To provide a clear distinction and in order to ensure that clients can only use public classes, the eHF provides the ability to split up the build results into multiple artifacts for one module (see [Table 5](#)).

Filename	Location	Description
<code>project.properties</code>	/	Controls the build process and the Maven plug-ins
<code><ApplicationId>-api-<Version>.jar</code>	/target	This artifact represents the public API of a module and contains all Java classes which should be provided for usage by other modules.
<code><ApplicationId>-config-<Version>.jar</code>	/target	The configuration package contains all configuration files necessary to use the module.
<code><ApplicationId>-bootstrap-<Version>.jar</code>	/target	This package contains all data files which are provided by a module, for example bootstrap and import data.
<code><ApplicationId>-runtime-<Version>.jar</code>	/target	The runtime artifact contains all classes which are needed to use the full functionality of a module. The classes of the API package are also included in this file.
<code><ApplicationId>-<Version>.jar</code>	/target	This includes the entire module with all configuration, class and data files. This file is deprecated.

Table5. Naming Schemes of the eHF Build Plug-in

Generally, all built JAR files follow the naming scheme and contain the corresponding *application ID* and version number in the filename (for example `ehf-document-api-2.6.0.jar`). Since the *module ID* has to be unique, the JAR archives also have unique names.

6.1.3 Domain Modeling

The openArchitectureWare (oAW) workflow engine is a declarative configurable generator engine. It provides a simple XML-based configuration language. The workflow file `workflow.oaw` located in the directory `/src/main/oaw` specifies the generator input. The directory `/src/main/model` contains the UML definitions `model.uml` and `eHF_Profile.profile.uml` used by the code generator. Both are referenced by the workflow file.

If the model uses code systems (see [Codes, Code Sets and Code Systems](#) on page 14), you have to provide the code system mapping file `codesystem-mapping.properties` in the directory `/src/main/model`. [Table 6](#) summarizes the artifacts required for modeling.

Filename	Location	Description
<code>workflow.oaw</code>	<code>/src/main/oaw</code>	The oAW workflow configuration used for the generation process
<code>model.uml</code>	<code>/src/main/model</code>	The UML domain model
<code>eHF_Profile.profile.uml</code>	<code>/src/main/model</code>	eHF UML profile
<code>codesystem-mapping.properties</code>	<code>/src/main/model</code>	Code system mappings used by the UML domain model

Table6. Naming Schemes for Modeling

6.1.4 Writing Java Code

The eHF Module-Template for applications and the eHF generator drive the package structure of Java code. The main package of a module is a configured input variable of the eHF Module-Template (refer to [Table 4](#)).

All custom created packages have to be located below the *application package*. Additionally, the eHF Generator creates sub-packages for data access objects, security interceptors, service adapters, and transfer objects as listed in [Table 7](#). If you customize the generated, moveable Java code, you have to maintain the structure of these sub-packages.

Sub package name	Description
<code>*.dao</code>	Data Access Objects (DAO)
<code>*.security</code>	Security Interceptors
<code>*.service</code>	Service Adapter, CRUD-Services
<code>*.service.adapter</code>	Data Transfer Objects (DTO)
<code>*.webapi.service.adapter</code>	External Transfer Objects (XTO)

Table7. Naming Schemes of Module Packages

For other Java artifacts (such as classes, methods, attributes, etc.), it is recommended to follow the [Sun Java naming conventions](#). On the whole, this will lead to uniform readable source code and improve the understandability.

6.1.5 Tokens

The eHF Build Plug-in copies configuration files and replaces specific tagged text. For this purpose, it uses two types of tokens:

- one for parameterizing single values (property tokens)
- one for merging complete fragments of configuration files (fragment tokens)

as shown in [Table 8](#).

Pattern	Example	Description
@@<Property>@@	@@<module.schema.name>@@	The property token will be replaced with the specific value of a property.
@@@<FragmentFile>@@@	@@@module.context.import.fragment@@@	The fragment token will be replaced with the complete content of the fragment files.

Table8. Naming Schemes of Property and Fragment Tokens

The actual values for the property tokens are defined in several property files in the top level directory of a module. The fragment files contain composite definitions. They are located in subdirectories of `/src/main/config/merge` and have the filename extension `.fragment`. [Table 9](#) lists all property configuration files and merge directories. In the case of an assembly module, the fragment files of all application modules are merged.

Filename	Location	Description
configuration.properties	/	Properties for the environment with default values
assembly.configuration.properties	/	Properties used for assembly builds
module.configuration.properties	/	Properties used for module builds
*.fragment	/src/main/config/merge/assembly	Fragment files used for assembly builds
*.fragment	/src/main/config/merge/module	Fragment files used for module builds

Table9. Naming Schemes for Properties and Configuration Fragments

6.1.6 Configuring Spring

Spring plays a central role in the infrastructure of eHF modules. With the eHF and the modularization concept, the Spring configuration has been split up into various files as listed

in [Table 10](#). The configuration files have to be unique across all modules. For this reason, they have the prefix <ModuleId> as part of the naming scheme.

Filename	Location	Description
<ModuleId>-config-context.xml	/src/main/config	Config Module Context
<ModuleId>-module-context.xml	/src/main/resources/META-INF	Main Module Context
<ModuleId>-module-system-context.xml	/src/main/resources/META-INF	Module System Context
<ModuleId>-module-runtime-context.xml	/src/main/resources/META-INF	Runtime Context
<ModuleId>-test-context.xml	/src/test/resources/META-INF	Test Context
<ModuleId>-system-context.xml	/src/test/resources/META-INF	Test System Context
<ModuleId>-context.xml	/src/main/gen/META-INF	Internal Context
<ModuleId>-api-context.xml	/src/main/gen/META-INF	Module API Context
<ModuleId>-webapi-context.xml	/src/main/gen/META-INF	Module Web-API Context
<ModuleId>-dependency-context.xml	/src/main/gen/META-INF	Module Dependency Context

Table10. Naming Schemes of Spring Context Files

6.1.7 Database

Maven creates the database and its tables as part of the build process by executing the following SQL scripts:

- `init.sql` (creates the database schema),
- `schema-create.sql` (creates database tables in the schema),
- `schema-custom.sql` (allows custom SQL statements like creation of indices).

The property `module.schema.name` in the file `module.configuration.properties` defines the name of the database schema. The default schema name is `<SchemaPrefix>_<ModuleId>` where `schema prefix` and `module ID` are the input variables of [Table 4](#).

For each database supported backend, there is a subdirectory `<Dialect>`, which can be found below the folder `/src/main/config/db`. The eHF supports currently HSQL and Oracle. By default, eHF configures a HSQL database in `configuration.properties`:

```
connection.dialect=org.hibernate.dialect.HSQLDialect
connection.driver=org.hsqldb.jdbcDriver
connection.url=jdbc:hsqldb:hsq1://localhost:9999/testdb
```

After the scripts executes, it is still necessary to run the bootstrapping and import process to ensure that all necessary data have been imported into the database. The XML files `ehf-assembly-bootstrap.xml` and `ehf-assembly-import.xml` specify the bootstrap and import data respectively. [Table 11](#) provides a list of the database-related files.

Filename	Location	Description
init.sql (default)	src/main/gen/META-INF/db/<Dialect>/	Creates the default database schema
init.sql (customized)	/src/main/config/db/<Dialect>	Creates a customized database schema
schema-create.sql	/src/main/config/db/<Dialect>	Creates the database tables
schema-custom.sql	/src/main/config/db/<Dialect>	Allows custom SQL
create-datasource-user.sql	/configuration/database/scripts/oracle<version>/	Creates an eHF database user and applies privileges for this user
create-functions.sql	/configuration/database/scripts/oracle<version>/	SQL Creation Procedure for granting privileges for an eHF module user
drop-functions.sql	/configuration/database/scripts/oracle<version>/	SQL Drop Procedure for dropping the creation procedure for privileges
ehf-assembly-bootstrap.xml	/src/main/resources/META-INF	Data that is imported in production environment
bootstrap.policy	/ehf-assembly/bootstrap	Bootstrap policy file for granting permission to the database tables
ehf-assembly-import.xml	/src/main/resources/META-INF	Data that is imported only in test environment

Table11. Naming Schemes for Database-related Files**Note:**

There are two possible ways to use the `init.sql` file: the first table row shows the default path to the `init.sql` file with its default settings.

The entry in the second row shows, where to store the `init.sql` file when custom settings should be used. You have to edit the `init.sql` and change the path as described above. The customized `init.sql` is chosen and overwrites the default one.

6.2 Logging

One of the driving concepts behind the eHealth Framework is maintainability of eHF-based products. A crucial aspect for ease-of-maintenance is the amount and the quality of log information produced by the system. This chapter outlines some general guidelines about what logging is, what it is not, and how to do it. These guidelines are mandatory for modules contained within the eHealth Framework's domain. Modules built with eHF in the scope of a product, however, are advised to follow the same guidelines for reasons of uniformity and, again, maintainability.

Logging in the context of eHF is supposed to provide vital information that enables the maintainer of a running software system to diagnose the product in case of a failure or other unexpected behavior. The support team will also need this information to locate and fix defects.

Knowing what to log is one thing, but you should also be aware of what *not* to log. Logging is not auditing. The latter represents a business concept, typically connected with legal requirements, and is covered in [Audit](#) on page [...](#). Logging is not tracing either. Fine-grained messages detailing each and every method invocation are not helpful, clutter the LOG file and obscure the really important messages. And finally, logging is also not intended for debugging purposes. If you want to find out what happens in your module, set a breakpoint and use a debugger.

Reviewing these dos and don'ts, it should be clear by now that logging benefits from a less-is-more approach. Use it spare and wise.

On the technical side, all logging within eHF is done through the [Apache Commons Logging](#) library. The following is a list of detailed, practical conventions that will help you log correctly:

- Never use `System.out` or `System.err`. This includes all code - even test code. In server environments these logs are lost or end up in completely different LOG files than the rest.
- Never use the `printStackTrace()` method on `Throwables`. In server environments these logs are lost or end up in completely different LOG files than the rest.
- Never use the `dumpStackTrace()` methods on the `Thread` class. Did we mention before that in server environments these logs are lost or end up in completely different LOG files than the rest?
- Use a static final log variable named `LOG` and always provide the class the variable is located in.

```
public class ClassWithLogging {
    private static final Log LOG = LogFactory.getLog(ClassWithLogging.class);
    ...
}
```

- The value of a variable should be put in square brackets separated from the log text by an additional whitespace.

```
LOG.debug("Results size: [" + results.size() + "]");
```

This way the reader can clearly distinguish between the static message and the dynamic computed values.

- No sensitive domain information must be logged into the system (for example complete patients or diagnoses). If possible, constrain log messages to IDs of the domain objects.

```
LOG.debug("Found patient: [" + patient.getId() + "]");
```

- Logging should be conditional if the log message is expensive to be produced. "Expensive" includes method calls beyond simple getters and in particular database access, reflection, and implicit calls of `toString()`.



Note: This applies both to DEBUG and INFO level. In case a piece of code will be used with very high frequency conditional logging should always be used.

```
if (LOG.isDebugEnabled()) { LOG.debug("Retrieved all 10000 search
    results: [" + searchResults + "]");}
```

- Static text logging should not be conditional, because the if-statements takes more time to process than instantiating the string.

In order to determine the log level, you can follow these guidelines:

- The error reporting uses log levels ERROR, WARNING exclusively. This is the DEFAULT log level in productive systems.
- The log level FATAL is *not* used (only in reserved cases), the ERROR log level should be used instead.
- Major business processing steps the application undertakes are logged in the log level INFO. The logging in this level must be both informative and short.
- The log level DEBUG is used to provide additional information relevant for developers.
- Logging policy. In order to make the LOG file informative and readable, lower system layers should produce detailed log output when the system is debugged, but less output when it's finally deployed (and is set to the log level WARN). In detail:
 - The Persistence Layer usually is based on third-party components, it's not known when and what it logs.
 - The Data Access Layer takes care of persistence and logs only in DEBUG.
 - The Service Layer is responsible for business logic only. It logs in INFO and DEBUG. See (1).
 - The Presentation Layer, Integration Layer, other service layer clients can log in any log level, including "exceptional" ERROR, WARNING, since the exceptions are caught and finally handled here.

6.3 Testing Conventions

In eHF, we follow an agile development approach with emphasis on Test-Driven Development (TDD) techniques. So each new functionality has to be covered by a respective test to ensure proper implementation. For development of eHF components, we define a set of conventions and best practices which help in writing meaningful and easily maintainable test cases which guarantee a high eHF code base quality.

Test Framework

Unit tests in eHF are written in JUnit. We suggest writing unit tests against JUnit version 4.4 or higher to leverage new features available with version 4. For more information about JUnit, we refer to <http://junit.org/>. Older, existing unit tests could be implemented using previous releases of JUnit. A migration of these test cases is not necessary since JUnit 4.4 is fully backwards compatible with JUnit 3.x releases.

Test Classification

In eHF, we see three groups of test cases with different responsibilities and increasing granularity:

1. **Unit Test** A unit test provides sufficient test cases for the least known artifact in eHF, a unit. Typically, a unit is a class or a set of classes which fulfill a certain concern. Library modules units should be fully covered by unit tests to ensure correct functionality, even in corner cases.
2. eHF demands a high level of reusability through modularization. Due to this, the next more coarse grained unit is called a module. Each module provides a well-defined set of services in form of its public API. These services implement business-driven use cases. These services should be tested thoroughly by **module tests**. Typically, each module which provides services relies on other services contributes by other modules.

For module tests, such services need to be mocked. Conventions for mocking platform services will be addressed later in this chapter.

3. In eHF, modules are aggregated in an assembly to provide an eHF-based product. In each shippable product, **integration tests** should be run successfully to guarantee correct behavior. In contrast to module tests, integration tests run with unmocked platform services to allow conclusion about integration with other modules. Typically, they are executed in an assembly environment.

Test Metrics

To measure the quality of eHF tests, we use the OpenSource code coverage analysis tool named Cobertura (see <http://cobertura.sourceforge.net/>). It is able to calculate both line and branch coverage for each eHF module. Cobertura is applied in form of a Maven plugin and calculates the metrics during build time. For eHF modules, we define the following code coverage thresholds:

- 70% of line coverage
- 50% of branch coverage

Cobertura provides us with always up-to-date information about the current code coverage in each eHF module. Furthermore, we are able to identify untested parts of a module and are able to see code places inside a module which are not covered by business use cases. In case of service modules, such places are often candidates to be removed. If exposed services do not run through these identified code parts, they tend to be obsolete.

Hint: Artifacts produced by the eHF generator are not instrumented by Cobertura. They will not be regarded by Cobertura since module developers cannot influence generated code.

Test organization

In eHF we follow Maven conventions and place tests in the `src/test/java` folder within an eHF project. We observe the rule organizing unit tests using the same package hierarchy as for the testee unit itself. The test folder is as well analyzed by Checkstyle to conform our conventions. This fact enforces us to provide JavaDoc for unit tests as well as for the sources. Since unit tests are seen as documentation, it is significant to provide proper documentation for them as well.

Necessary test resources are held in `src/test/resources`. This allows the provision of additional test fragments without them being packaged in resulting JAR archives.

Spring test context files are as well regarded as test resources and thus are held in `src/test/resources`. The amount of test context files should be minimized in order to keep the codebase maintainable. Therefore, each module should only provide a single test context. Per convention, this is located in `src/test/resources/META-INF/ehf-<ModuleId>-test-context.xml` and should be referenced by each Spring-based unit test.

Integration tests follow the same organizational conventions as unit and module tests. The only difference is that these tests are executed in an assembly environment where mock implementations are replaced with the productive service implementations.

Mocking

Unit and module tests should always run isolated of any environment and test the implementation of the focussed unit respectively module service. To decouple the testee from its environment, mock objects are used. Such mock objects mimic a certain interface by implementing a dummy implementation. In eHF, we support the following Mock Frameworks:

- **EasyMock:** Provides support for dynamic mock objects for interfaces. Such mock objects are generated by using Java's proxy mechanism. More information about EasyMock can be found under
- **Spring Test:** Provides convenient static mock objects for testing implementations around JNDI, the Servlet and Portlet API

Beside the supported mock frameworks, eHF modules as well provide mock implementations for most of the DTO service adapters. They implement the same interface under which the productive implementations are exposed. So, instead of injecting the real implementations, the mock implementations could be taken in a test scenario.

Test Tools

eHF comes with the test-tools project which enables you to write tests in a more convenient way. Currently, it supports your test activities in the following areas:

- Concurrency tests: Abstract test cases which provide you with the ability to run your test code concurrently. You can specify the number of parallel threads as well as the Runnable which should be executed. This is especially helpful against common concurrency problems which can lead to deadlocks or race conditions for example.
- Automated accessor tests: The Java Beans convention dictates the existance of accessor methods for each field of a bean in form of getters and setters. Correct accessor implementations could be tested automatically by using the test-tools.

Best Practices for writing tests in eHF

In this section, we want to list all of the best practices we identified in the past. These practices incorporate in our test activities and should be regarded whenever tests are written.

Use service mock implementations in module and unit tests. When modules depend on other modules, this dependency should be resolved at test time by injecting mock services rather than the real implementations which come from the dependent module. This keeps your test configuration clear and avoids unnecessary bootstrap effort and cuts dependencies during test time. Mock objects and mock service implementations help you in stating your expectations towards the utilization of external interfaces. Inter-module communication happens via the DTO layer, so modules typically communicate via module service DTO adapters with each other. In general, eHF service modules provide abstract mock implementations of the DTO adapter service interface. You could easily extend and customize this implementation for your tests. For your orientation: all module service DTO adapters extend the interface `com.icw.ehf.commons.service.ModuleServiceAdapter`

Use the system context to mimic platform context for module and unit tests. Module and unit tests are not executed in an integrated environment. For that reason, the system context needs to be imitated. Such a system context provides globally used beans (e.g. Hibernate session factory or transaction manager) and service beans contributed by other modules (e.g. security service). In a test environment, such beans a module is depending on need to be replaced by mock counterparts to isolate tests. This is the purpose of the system context, which is located at `src/test/resources/META-INF/ehf-<ModuleId>-test-context.xml` per convention. The system context is only used for tests and is not part of the packaged JARs. System contexts contain property tokens (e.g. for database connection properties) which are resolved at test context initialization time. The property values are taken from the `configuration.properties` file located at the top level project folder. It is recommended to separate the system context and the test context because of the different concerns. However, it is a recommended technique to import the system context into the test context to get the necessary platform beans needed by the module.

Use test support classes whenever applicable. For common test cases there already exist a lot of generic test classes which can be reused to avoid code duplication throughout tests. Following sources should be consulted before writing own test helper respectively test support classes:

- **Spring Test:** This library provides support for tests written in conjunction with the Spring Framework. It provides support for dealing with Spring contexts during test. For example, you can use an annotation-driven approach to target to your test context

and populate fields by the Spring autowiring mechanism to easily get access to your Spring-maintained beans:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:/META-INF/ehf-<ModuleId>-test-
context.xml"})
public class Test {

    @Autowired
    protected SecurityService securityService;

    @Test
    public void test() {
        assertNotNull(securityService);
    }
}
```

Spring test support test-tools

Use highest JUnit version whenever possible. Test cases always should leverage the newest version of JUnit available. Exceptions could be made for existing legacy tests, which are implemented against older versions. Nevertheless, it is recommended to lift them as well to the highest version to harmonize the test base since migration effort is very low. Note: Currently, eHF uses the Maven Test Plug-in, which runs with earlier versions < 4.x of JUnit. Therefore, you have to declare the static method `suite()` which returns an adapted test case the plug-in can handle:

```
public static junit.framework.Test suite() {
    return new JUnit4TestAdapter(<Name of the Test class>.class);
}
```

Part III - Development Environment

The eHealth Framework provides both a runtime environment for health care solutions and an application development platform that can be used for the development of new as well as existing modules. The eHealth Framework supports and accompanies the full software development life cycle to produce lean and high quality health care applications.

Developers can turn to the provided development support and use it to enhance their productivity. The eHealth Framework supports agile and iterative development methodologies. When this approach is applied consistently, it will result in the creation of robust and sophisticated health care applications. The modules that are provided by the eHealth Framework have been developed using the eHealth Framework development environment.

7 ICW Eclipse Development Environment

An integrated development environment (IDE) is a software application that provides comprehensive facilities to software developers. Usually, an IDE integrates a source code editor, a compiler (and/or interpreter), build automation tools and a debugger.

Advantages of using integrated development environments

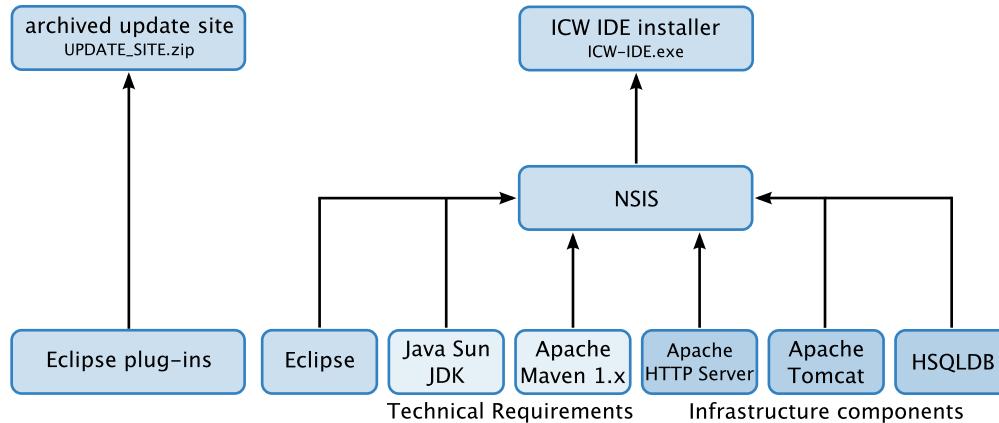
The usage of integrated development environments in a project or by an entire organization can yield important advantages over conventional editors or heterogeneous development environments.

- Using a standardized IDE may boost the developers' **productivity** significantly. It can be extremely beneficial to have a seamless integration of essential system and software development functions (such as build tools) in one integrated development environment.
- As far as **usability** is concerned, new users can familiarize themselves with new IDE plug-ins quite easily via an overall consistent "look and feel". I.e. the appearance of the IDE and the underlying workflow concepts are generally consistent in all IDE plug-ins.
- A standardized IDE can contribute substantially to **adherence with corporate coding standards**. Coding standards are written procedures describing coding or programming style conventions. They specify rules governing the use of individual constructs provided by the programming language. They include naming, formatting, and documentation requirements , which help prevent programming errors, control complexity and promote understandability of the source code. ICW's quality management board has introduced corporate Java formatting rules and a generally agreed Checkstyle configuration. The ICW-IDE embraces and enforces these company-wide standards - as ICW's Java formatting rules and Checkstyle rules are already enabled in a pre-configured Eclipse workspace.
- An IDE may also facilitate **team work** (e.g. via built-in support for versioning systems like SVN or bug tracking systems such as JIRA).

Eclipse and the ICW IDE

The ICW IDE package is currently only available for Microsoft Windows, because all of the eHF developers' working stations are running on Windows.

The ICW IDE is distributed in the form of two files: an installer and an archived update site for Eclipse. The main installables (such as Eclipse, the Sun JDK, Maven, the Apache web server, Tomcat and HSQLDB) get installed and pre-configured via a [NSIS-based Windows installer](#) ➤.

**Figure 29:** Composition of the ICW IDE

The ICW IDE bundle is mainly based on [Eclipse](#) ↗ an open source Java-based IDE. In fact, Eclipse constitutes a whole software platform comprising extensible application frameworks, tools and a runtime library for software development and management.

As Eclipse was mainly written in Java, it is also primarily - but not exclusively - targeted at Java developers. However, support for numerous other programming languages such as C++, Python, Perl, Ruby, Groovy, Fortran, Cobol, PHP, JSP/Servlet, J2EE as well as for OOD/OOP design tools can be enabled by installing specific Eclipse plug-ins. As for the eHF, the primary programming language is Java. Shell scripts are also used (e.g. for executing upgrade procedures). Perl is primarily used in the context of build and deployment management. Furthermore, the eHF also makes extensive use of OOP/OOD tools as well as of tools for model-driven software development.

The ICW IDE combines the most recent version of Eclipse with a set of well aligned [Eclipse plug-ins](#) (delivered in the form of a stand-alone Eclipse update site). The selection of plug-ins has been based on a requirements analysis involving the eHF developers themselves as well as other software developers who are building applications on top of the eHF. Currently the following plug-ins are contained in the archived update site:

- **AspectJ DevelopmentTools (AJDT):** The [AspectJ Development Tools \(AJDT\)](#) ↗ project provides Eclipse platform based tool support for AOSD with AspectJ.
- **Eclipse Checkstyle plug-in :** The [Checkstyle plug-in](#) ↗ performs static code analysis, and generates a report of how well it adheres with predetermined code-style policies - using Checkstyle. The Checkstyle plug-in has been configured to validate the source code against the ICW-wide coding standards.
- **Eclipse Modeling Framework (EMF):** [EMF](#) ↗ is a modeling framework and code generation facility for building tools and other applications based on a structured data model.
- **FindBugs plug-in:** [FindBugs](#) ↗ is a plug-in which uses static analysis to look for bugs in Java code.
- **Graphical Editing Framework:** The [Graphical Editing Framework \(GEF\)](#) ↗ allows developers to take an existing application model and quickly create a rich graphical editor. The GEF is used as a prerequisite for other Eclipse plug-ins.
- **Hibernate Tools:** [Hibernate Tools](#) ↗ is a toolset for Hibernate3, implemented as an integrated suite of Eclipse plug-ins, together with a unified Ant task for integration into the build cycle. Hibernate is used extensively by the eHF, because it permits persistence of Java objects in relational databases.
- **J2EE Standard Tools (JST):** The scope of the [J2EE Standard Tools](#) ↗ subproject is the support of J2EE programming. This includes the support of APIs covered by the

J2EE1.4 specifications (e.g. JSP, Servlets, EJBs, JCA, JMS, JNDI, JDBC, Java Web Services, JAX* and related JSRs).

- **Maven:** The [Maven plug-in](#) provides the ability to generate Eclipse project files (.classpath and .project files), as well as configuring Eclipse to use Maven as an external tool. Maven is used for building the various eHF modules and for deploying the eHF assembly.
- **Mevenide:** [Mevenide](#) aims to integrate Maven (<http://maven.apache.org>) into popular IDEs. As of today Eclipse, Netbeans and Jbuilder are supported. Eclipse plug-in provides support for: Launching Maven from within the IDE, editing POM through a multipart editor, editing Jelly files, browsing and searching repositories, synchronizing POMs and many other things.
- **Mylyn:** [Eclipse Mylyn](#) (formerly known as Mylar) is a task-focused UI that reduces information overload and makes multi-tasking easy. It does this by making tasks a first class part of Eclipse, and integrating rich and offline editing for repositories such as Bugzilla, Trac, and JIRA.
- **oAW:** The [openArchitectureWare](#) project provides a suite of tools and components assisting with model driven development built upon a modular MDA/MDD generator framework implemented in JAVA supporting arbitrary import (design) formats, meta models, and output (code) formats. In the context of the eHF, oAW is used by the generator module (i.e. for model-driven software development).
- **Orangevolt XSLT editor:** [Orangevolt](#) EclipseXSLT provides XSLT support to the Eclipse platform. It is the Eclipse-based successor of the Java/swing based ROXES XmlWrite XSLT editing environment. It provides many great enhancements when working with XML/XSLT. The Orangevolt XSLT editor can also be used as an editor for XML based documentation platforms such as DITA.
- **Quantum database editor:** [Quantum](#) is a tool to run SQL queries against a database. It allows a user to bookmark databases and dynamically loads the JDBC driver. It also allows you to view the tables, views and sequences in a tree format.
- **Spring IDE:** The [Spring IDE](#) is a graphical user interface for the configuration files used by the Spring Framework. It's built as a set of plug-ins for the Eclipse platform. The Spring Framework is a central backbone of eHF Modules. While eHF Modules can be deployed in any environment (application server, web container, standalone) Spring provides the necessary hooks and benefits of the runtime environment.
- **Subversive:** [Subversive](#) is an Eclipse plug-in that adds Subversion (SVN) integration into the Eclipse IDE.
- **W3C Sac Feature:** A plug-in for W3C Sac (Simple API for CSS)
- **Web Standard Tools (WST):** The [Eclipse Web Tools Platform \(WTP\)](#) Project provides APIs for J2EE and web-centric application development. WST has not been added to the Eclipse update site because of the eHF itself but because it is used by web applications which are built on top of the eHF (such as LifeSensor).
- **XML Schema Infoset Model (XSD):** The [XML Schema Infoset Model](#) is a reference library that provides an API for use with any code that examines, creates or modifies W3C XML Schema - standalone or as part of other artifacts, such as XForms or WSDL documents.

Apart from Eclipse, the pre-configured Eclipse workspace and the update site, the ICW IDE package also features a selection of third party components that facilitate the initial setup of the developers' working environment.

These third party components can be classified into two categories: technical prerequisites for Eclipse and optional infrastructure components.

Technical prerequisites : The Sun JDK is a prerequisite for building Java projects with Eclipse. As Apache Maven is used for building and deploying the eHF modules, it is also considered a prerequisite.

- The [Sun JDK](#) is a prerequisite for developing Java projects with Eclipse. The ICW IDE installer contains the specific version of the Sun JDK which was used as a reference for QA tests.
- [Apache Maven](#) is a software tool for Java project management.
- The [Maven runner](#) is an alternative tool by Nicolay Petrov to further facilitate Maven 1.x builds.

Infrastructure components: The following packages are not strictly necessary for building a Java application with Eclipse. However, they facilitate the setup of a local infrastructure for deploying and testing the eHF. Hence, the following tools can be considered as standard equipment for eHF developers and testing engineers.

- The [Apache HTTP server](#) is the most widely used web server software on the Internet today.
- [Apache Tomcat](#) is a servlet container developed by the Apache Software Foundation (ASF). Tomcat implements the Java Servlet and the JavaServer Pages (JSP) specifications from Sun Microsystems, and provides a "pure Java" HTTP web server environment for Java code to run.
- [HSQLDB](#) (Hyperthreaded Structured Query Language Database) is a relational database management system written in Java.

Installation of the ICW IDE

Please refer to the installation manual for the IDE, available from Subversion or the official download site of the ICW IDE.

8 Build Plug-in

Building modules and assemblies for the eHealth Framework is a non-trivial task that involves many steps and is bound to be error-prone if done manually. The eHF therefore provides a Maven plug-in that automates the build process. This chapter describes how the plug-in can be utilized. A basic understanding of Maven is required.

The plug-in is named `ehf-build-maven-plugin` and contributes a number of tags and goals to the build process. If you create your projects from an eHF template, a dependency declaration like the following will already be included in your `project.xml`:

```
<dependency>
  <groupId>ehf</groupId>
  <artifactId>ehf-build-maven-plugin</artifactId>
  <version>2.10.3</version>
  <type>plugin</type>
</dependency>
```

Moreover, the eHF project templates also generate `maven.xml` files. These provide public goals as entry points into the build process. As a developer, you will typically invoke the `dev:build` goal, while the automated integration builds will call the `masterbuild:all` goal. Both make use of the goals provided by the eHF Maven build plug-in.

The Build Plug-in fulfills the following responsibilities during the build process:

- Invoke the generator before compilation.
- Include AspectJ in the compilation process.
- Modify the classpath so that modules are compiled only against the public API of other modules.
- Copy resource and configuration files to their target locations.
- Merge configuration file fragments to fragment tokens.
- Replace property tokens in configuration files.
- Split a module into distinct artifacts (API, runtime, bootstrap, etc.).
- Start and stop the database.
- Create database schema(s).
- Bootstrap and import data.

All of the above tasks are controlled by a number of configuration properties. Most of these have reasonable default values. Others are typically set by eHF's project templates. You can override all properties in your project's `project.properties` file. Properties are your major and yet easiest influence on the build process. A comprehensive list of properties used by the eHF Build Plug-in is given in the appendix under [Build Plug-in Configuration](#) on page 342

You will typically interact with the eHF build plug-in in three ways: when building a module, when building an assembly, and when working with the database. The usage scenarios are covered in the following sections.

8.1 Module Build

Building a module is one of the most essential tasks when working with the eHealth Framework. There are many steps involved and most of the time you will want to execute them all by calling the `dev:build` goal.

This results in the standard build sequence described (in order) in the following paragraphs, each having the Maven goal name of the respective step as its headline. You can invoke most of these goals directly to execute only portions of the build sequence.

A Maven build creates many artifacts in the `target` folder of a module. In order to perform a clean build the standard Maven `clean` can be executed to purge the previous build's results.

8.2 Assembly Build

The assembly build produces a WAR file that can be deployed into an application server. The process has two slightly distinct variants depending on the value of the `maven.ehf.build.release` property. You typically decide which one is used by calling either the `dev:build` or the `ehf:release` goal, both of which are usually generated by eHF's assembly project template. The goal `dev:build` on assembly level will produce a WAR file that can be deployed in a web application container. This is intended to support the developer. The `ehf:release` performs additional steps and provides a distributable artifact that can be shipped to a client for the installation of the software.

Assembly Dependency Properties

When adding a dependency declaration for an eHF based module artifact, to an eHF based Assembly's `project.xml` you can define numerous properties within the declaration, that then influence how the build plugin processes these dependency artifacts.

An example of a dependency declaration that includes two of these properties (`ehf-module` and `ehf-persistent`) is shown here:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-audit-config</artifactId>
    <version>SNAPSHOT</version>
    <properties>
        <ehf-module>audit</ehf-module>
        <ehf-persistent>true</ehf-persistent>
    </properties>
    <type>jar</type>
</dependency>
```

The following list details the available properties and explains when they should be used.

ehf-module

- Lets the build plugin know that it needs to pull configuration files from this artifact i.e. database schema create scripts, Spring configuration etc, for use in the assembly.
- Should generally only be used for configuration artifacts, for example, `ehf-audit-config`.
- The build process will pull the relevant files out of the `config` artifact and copy them (or include them via token replacement), in the corresponding assembly folder (or file) at the appropriate time. Thereby making the required part of the individual module's configuration available to the whole assembly where required.
- The value of this property is also used for various output messages when working on the module during the build, it should generally have the same value as the `module.name` property contained in the `project.properties` file of the dependency's own project.

ehf-persistent

- Details that this artifact includes database relevant information, i.e. schema create scripts.
- Again it is generally only required for configuration artifacts.
- If this boolean property is not defined, but the `ehf-module` property is defined, then a default value of `true` is taken.

- The database scripts will be taken from the artifact and used during either the build or install process to create the required database schema for your module in your running database at the appropriate time.

ehf-artifact

- Details the "type" of artifact that this dependency references.
- Currently can only be used to mark an artifact dependency as a documentation artifact (containing DITA based documentation files) - only value that is currently valid is documentation, any other value will for the time being be ignored and will therefore have no effect.
- All files within the referenced artifact will be copied to the path defined by the property `ehf-documentation-path` under `target/dita`.

ehf-documentation-path

- Location under `target/dita` where the referenced artifact's contents should be copied to.

These last two properties, `ehf-artifact` and `ehf-documentation-path` are really only of relevance for DITA based documentation (<http://dita.xml.org/>). Based on the files that are copied using the previous two properties, PDF files will be generated from the dita files contained within. The following is an example of a dependency definition that uses these properties:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-reference-documentation-doc</artifactId>
    <version>SNAPSHOT</version>
    <properties>
        <war.bundle>false</war.bundle>
        <ehf-artifact>documentation</ehf-artifact>
        <ehf-documentation-path>reference</ehf-documentation-path>
    </properties>
    <type>jar</type>
</dependency>
```

This means that all files from this artifact will be copied to `target/dita/reference`, and the eHF Reference Documentation PDF file will then be produced based on these files.

war.bundle

- Is a maven specific property, that when set to `true` means the dependency artifact will be included in the `/WEB-INF/lib` folder of the resultant `war` file of the assembly's build process.

category

- Causes the dependency artifact to be copied to given folder in the resultant release zip of the assembly's build process.
- Valid values for this property are actually determined by the assembly itself through the `maven.releasedistribution.categories` property in the `project.properties` file of the assembly.

The following is an example of a dependency that uses both `war.bundle` and `category`:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-commons-runtime</artifactId>
    <version>SNAPSHOT</version>
    <properties>
        <war.bundle>true</war.bundle>
        <category>lib</category>
    </properties>
    <type>jar</type>
```

```
</dependency>
```

This means that the `ehf-commons-runtime.jar` file will be included in the `/WEB-INF/lib` folder of the assembly's resultant `.war`, and will be copied to the `lib` folder of the release zip file.

There are four further properties that you might see used when defining a dependency to specifically the `ehf-build-tools` in an assembly. These are, `release`, `releaseIncludePattern`, `releaseExcludePattern` and `releaseTarget`.

These properties are very specific for the `ehf-build-tools` dependency in an eHF assembly, and basically cause all the relevant ant scripts, required for the install process, to be copied to the release's `bin` folder.

8.3 Property and Fragment Tokens

The Build Plug-in copies configuration files to the target folder and replaces specific tagged text. For this purpose, it uses two types of tokens: *property tokens* and *fragment tokens*. Property tokens are for parameterizing single values, while fragment tokens are for merging complete fragments of configuration files as shown in [Table 12](#).

Pattern	Example	Description
<code>@@<Property>@@</code>	<code>@@module.schema.name@@</code>	The property token will be replaced with the specific value of a property.
<code>@@@<FragmentFile>@@@</code>	<code>@@@module.context.import.fragment@@@</code>	The fragment token will be replaced with the complete content of the fragment files.

Table12. Property and Fragment Tokens

The actual values for the property tokens are defined in several property files in the top level directory of a module. The fragment files contain composite definitions and are located in subdirectories of `/src/main/config/merge` of a module or assembly (see [Naming Schemes](#) on page 57).

Fragment tokens support the merging of configuration fragments into generator templates and generated configuration artifacts. Furthermore, each module can contribute defined configuration fragments to the configuration of an assembly.

A further explanation and a complete list of provided fragment tokens is given in the Appendix. Please consult [Fragment Tokens](#) on page 352.

8.4 Database Management

The eHealth Framework's Maven Build Plug-in provides limited access to the database by making available the following public goals:

start-database

Starts an HSQL database instance in the foreground. Please note that this is slightly different from the (internal) goal used during the test cycle which runs the database in the background.

stop-database

Stops the HSQL database instance that was started with the start-database goal.

ehf:bootstrap

Imports bootstrap data from the location specified by the maven.ehf.db.initialize.bootstrap.pattern property into the database.

ehf:import

Imports import data from the location specified by the maven.ehf.db.initialize.import.pattern property into the database.

ehf:content-import

Imports content data from the location specified by the maven.ehf.db.initialize.import.content.pattern property into the database.

9 Customization Plug-in

Applications built with the eHealth Framework (eHF) are usually packaged as an assembly into a release artifact. These artifacts are meant to be distributed to the production site, where they can be installed using the [Installation Ant Scripts](#) on page 81.

Often it is however necessary that a customization is applied to the artifact before it can be delivered to the on-site administrator for installation. Typically, this involves localization or branding of the application. Ultimately such a procedure produces one or more variants of the original application release artifact.

The eHF Customization Plug-in provides an easy and configurable way to customize distributable artifacts. Typically, these artifacts are eHF module or assembly archives. In general the eHF Customization Plug-in is capable of manipulating any arbitrary (nested) ZIP or JAR archives.

The eHF Customization Plug-in works in the following way:

1. It unpacks the artifact(s) to be customized into a temporary location.
2. It processes the given customization rules. This may involve:
 - Copying additional (or replacing) files into the temporary location.
 - Deleting files from the temporary location.
 - Adding or overwriting properties in configuration files.
 - Transforming XML files.
3. It re-packages the modified files from the temporary location into the customized version of the distributable artifact.

The following sections describe this process in more detail.

9.1 Defining a Customization Project

Customizations must be defined in a (Maven) project. The easiest way to create such a customization project is to use the `ehf-customization-template` project template. If you do not have access to this template, you will have to add the following dependency to your project's `project.xml`:

```
<dependency>
  <groupId>ehf</groupId>
  <artifactId>ehf-customization-maven-plugin</artifactId>
  <version>2.10.3</version>
  <type>plugin</type>
</dependency>
```

9.2 Defining the Targets for Customization

The eHF Customization Plug-in will determine the the artifact(s) which are to be customized over the `customization.targets` property in `customization.properties` file in each of the specified customization folders. Therefore it is required so that all wanted archive paths are via `customization.targets` reachable (see [Customization Actions](#) on page 77).

Please note that the customization folders are applied in the order determined by the `customization.targets` property.

You can specify helper properties for later use:

```
# convenience property to ease configuration
ehf.module.version=2.10.3
```

The root archive(s) are using specialized properties in your project dependencies specified in the `project.xml`:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf</artifactId>
    <version>2.10.3-bin</version>
    <properties>
        <ehf-customization-target>true</ehf-customization-target>
    </properties>
    <type>zip</type>
</dependency>
```

The additional property `ehf-customization-target` indicates that the given plug-in is the target of the customization. This strategy is very useful if you manage your distributable artifacts in your Maven repository.

Alternatively the plug-in supports additionally that the target is located in your working directory. In order to identify it as customization target the following property has to be specified in the `project.properties` file:

```
# working dir customization target
maven.ehf.customization.target=ehf-2.10.3-bin
```

9.3 Defining Customization Rules

Customizations are defined in one or more `customization.properties` files. These property files contain the specification of what needs to be done. They can make use of standard actions provided by the eHF Customization Plug-in. It is currently not possible to define custom actions.

`customization.properties` files can reside anywhere in your customization project. While the eHF Customization Plug-in does not rely on any particular folder structure, it is recommended to organize the project by distinct topics, like branding or localization topics. Input files referenced by your `customization.properties` file, however, must reside in a directory named `files` next to the `customization.properties` file. A typical customization project layout looks like this:

```
localization/
  files/
    my-custom-file.txt
  customization.properties
branding/
  files/
    some-other-file.properties
  customization.properties
```

Instead of defining customization definitions in the current project, they can also be included from other projects by defining specially qualified dependencies:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-example-customization</artifactId>
    <version>1.0.0</version>
    <properties>
        <ehf-customization>true</ehf-customization>
    </properties>
    <type>jar</type>
```

```
</dependency>
```

The `ehf-customization` property signals that this project is to be considered an input path for customization definitions. This mechanism is very useful if you want to re-use general purpose customizations.

9.3.1 Customization Actions

Each `customization.properties` file has common properties to specify meta data for the customization:

```
#description of what the customization is about
customization.description=eHF Document Bootstrap Data Customization

# comma separated list of targets
# (patterns - i.e. wildcards for versions - are supported)
customization.targets=ehf-*-bin.zip/ehf-*-bin/lib/ehf-document-bootstrap-*.jar
```

The `customization.targets` restricts the scope of the customization actions to the given location(s). Note that the paths given here also specify the the archive(s) so the artifact(s) which are to be which are to be customized (see [Defining the Targets for Customization](#) on page 75).

In the following sections, each of the manipulation actions and the additional properties to control the actions are specified.

Note: the order in which the actions are listed here is the order in which the customization plug-in executes each action.

Delete

```
# specify delete patterns (relative to the target)
customization.delete.include=**/*-cfg.xml
customization.delete.exclude=-nothing-
```

Deletes all files in the target folder that matches the specified pattern.

Copy

```
# specify copy patterns (relative to the 'files' folder)
customization.copy.include=**/*
customization.copy.exclude=-nothing-
```

Performs a copy of files specified by the pattern properties. This feature can be used for adding or replacing files in the customization target. By default, existing files are overwritten by the copy operation. You can set the `customization.copy.overwrite` property to false in order to avoid this behavior.

Add/Overwrite Properties

```
# property manipulation patterns
customization.property.include=**/*.properties
customization.property.exclude=-nothing-
```

Merges property files as they can be found in the source and target folder matching the provided pattern. Whereas the action can be used to add and overwrite properties, it is not possible to remove properties.

Transform

```
# patterns specifying the XSLT to apply to the target files
customization.transform.include=**/*.xsl
```

```
customization.transform.exclude=-nothing-
```

Applies an XSLT transformations matching the pattern. The XLST paths have to match the target files paths plus a .xsl suffix.

9.4 Applying Customizations

Once you have specified the customization target and the customization rules, you need to define the ehf.maven.customization.sequence property that determines the order in which customizations are applied:

```
ehf.maven.customization.sequence=\
    localization-DE/ehf-document-bootstrap,\ 
    branding-CUSTOMER/ehf-document-runtime,\ 
    branding-CUSTOMER/ehf/artifact,\ 
    branding-CUSTOMER/ehf
```

Note that *-wildcards are supported, too.

A file named `customization.properties` has to be provided in each of the specified folders. Then you can run the `customize:all` goal. It applies all customizations in the given order. The goal can be decomposed in the three goals below:

1. `customize:prepare` to extract the target archives as specified in the `customization.targets` property.
2. `customize:execute` to actually perform the customizations as specified in the `ehf.maven.customization.sequence` property.
3. `customize:finalize` to repackage the archives in the reverse order as they were unpacked. The resulting ZIP file is packaged in the target folder and renamed according to the property `maven.ehf.customization.tag`. For example, if the tag is specified as `maven.ehf.customization.tag=CUSTOMER-LANG`, the goal will generate a customized `ehf-SNAPSHOT-CUSTOMER-LANG.zip` in the target folder.

While you may call these goals separately, it will almost always be more convenient to use the `customize:all` goal to accomplish everything in one go.

Note:



In the past it was necessary to specify in addition to the sequence of customizations a property named `ehf.maven.customization.archive.sequence`. This property specified the sequence in which (nested) archives were unpacked and repacked in the `customize:prepare` and the `customize:finalize` goals.

This property has however created some irritation. It was hard to maintain and therefore error prone. With eHF 2.10 the property is now automatically derived based on the customization sequence and the target folders in the customizations.

9.5 Extensions

The eHF Customization Plug-in provides a sophisticated tooling for modifying any arbitrary archive. In the context of eHF applications, however, one recurring customization task is the

extension of an assembly by additional modules. This is a rather cumbersome effort when using the above mentioned customization actions exclusively. The eHF Customization Plug-in therefore offers a special way to support this particular use-case.

Adding a Runtime Artifact

In order to add an artifact you have to specify the following dependency:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-example-runtime</artifactId>
    <version>1.0.0</version>
    <properties>
        <ehf-artifact>runtime</ehf-artifact>
    </properties>
    <type>jar</type>
</dependency>
```

This dependency with the property `ehf-artifact` set to `runtime` indicates the plug-in to construct and apply a customization to include the runtime artifact in the target archive.

This extension can be used to include any runtime artifact in the customized application. This can either be an arbitrary third-party artifact or a regular eHF-based module runtime artifact.

In order to specify the destination of the runtime artifacts the property `customize.product.war.lib` must be specified.

Adding an eHF Configuration Artifact

In order to add an configuration artifact you have to specify the following dependency:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-example-config</artifactId>
    <version>1.0.0</version>
    <properties>
        <ehf-artifact>config</ehf-artifact>
    </properties>
    <type>jar</type>
</dependency>
```

This dependency with the property `ehf-artifact` set to `config` indicates the plug-in to construct and apply a customization to include the configuration contained in the artifact into the target archive.

This extension can be used to include an eHF configuration artifact to be included in the customized application. In combination with adding a runtime artifact this extension can be used to add a eHF Service Module at customization time

In order to specify the destination of the customization the property `customize.product.modules` must be specified and point to the folder where all module configurations are kept in the target archive.

Adding an eHF Web Artifact

In order to add a web artifact you have to specify the following dependency:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-example-web</artifactId>
    <version>1.0.0</version>
    <properties>
        <ehf-artifact>web</ehf-artifact>
    </properties>
    <type>jar</type>
```

```
</dependency>
```

This dependency with the property `ehf-artifact` set to `web` indicates the plug-in to construct and apply a customization to include the web artifacts contained in the artifact into the target archive.

This extension can be used to include an eHF web artifact to be included in the customized application.

In order to specify the destination of the web artifact's content the property `customize.product.war` must be specified and point to the folder reassembling the root of the war file inside the archive.

Adding a Categorized Artifact

Apart from runtime and configuration modules, the eHF Customization Plug-in supports adding arbitrary (Maven) artifacts to the customization target. This is done by assigning so-called *categories* to such artifacts and by mapping categories to locations in the customization target.

In order to categorize an artifact you have to specify the following dependency:

```
<dependency>
  <groupId>ehf</groupId>
  <artifactId>ehf-example-test</artifactId>
  <version>1.0.0</version>
  <properties>
    <category>test</category>
  </properties>
  <type>jar</type>
</dependency>
```

Additionally you have to specify the `maven.releasedistribution.categories` property in your `project.properties` to map a category onto a path in the customization target. Multiple mappings are comma-separated. For example:

```
maven.releasedistribution.categories=test:${maven.ehf.customization.target}.zip/
${maven.ehf.customization.target}/lib
```

The above example instructs the eHF Customization Plug-in to include the `ehf-example-test` artifact in the `lib` folder of the customized application.

One possible use of this functionality is to contribute your "test data" to be imported during deployment of an application.

10 Installation Ant Scripts

The eHF Build Plug-in produces so-called distributable artifacts. These artifacts contain the complete distribution of a product. Using the eHF Customization Plug-in such artifacts can be customized by applying modifications or even functional extensions to a distributable artifact.

However a distributable artifact has the purpose of being delivered to the customer for configuring the product to its environment and to eventually install the software. This non-trivial procedure is supported by a set of Ant scripts that are delivered as part of the distributable artifact.

This chapter describes the main functionalities of the scripts and illustrates how it can be used.

10.1 Prerequisites

The installation scripts are an optional element of the assembly. There is an easy way to supplement them (and potentially other scripts) in the distributable artifact. The scripts are currently provided by the ehf-build-tools project. To include the scripts in the release just provide the following configuration in your assembly's `project.xml`:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-build-tools</artifactId>
    <version>SNAPSHOT</version>
    <properties>
        <category>lib</category>
        <release>ant</release>
        <releaseIncludePattern>**/*</releaseIncludePattern>
        <releaseExcludePattern/>
        <releaseTarget>install</releaseTarget>
    </properties>
</dependency>
```

The details of the used properties are described as part of the eHF Build Plug-in documentation.

Using this extension to the `project.xml` the install scripts will be contributed to the folder `install` in the distributable artifact.

10.2 Script Targets

After unpacking the distributable artifact and adapting the `configuration.properties`, the installation procedure can be executed using the following command:

```
ant -f install/install.xml <targetname> -D<parameter> ...
```

The subsequent sections describe, which target names and parameters are possible.

Configuring the Distribution

After the changes have been made in the `configuration.properties` file, these properties have to be propagated. The following targets take over this responsibility.

Target Name	Description
-------------	-------------

configure:all	<p>Applies the configuration from the configuration.properties file to all elements of the distributable artifacts that are involved in the installation procedure.</p> <p>This target is a prerequisite to all other goals and must be executed every time the configuration.properties are changed.</p> <p>Please note that the configure:all target implicitly calls configure:artifacts.</p>
configure:assembly	<p>Configures the product assembly. Independent of the artifacts this is required for steps like initial installation, upgrades and other activities that are not part of the deployable artifacts.</p> <p>Please note that the configure:all target implicitly calls configure:assembly.</p>
configure:artifacts	<p>Configures the deployable artifacts that are part of the distribution. This target is implicitly called by configure:all.</p> <p>Please note that the configure:artifact target creates configured artifacts in the root directory of the distribution. The artifacts are ready for deployment in the appropriate container.</p> <p>The deployment to the container is currently not part of the installation procedure as it strongly depends on the customer infrastructure.</p>
configure:clean	<p>Cleans the configuration done previously (inside the .install folder).</p> <p>Please note that the configure:clean target implicitly calls configure:assembly.</p>
configure:initialize	<p>Initializes the application. In this step further files (like a key store for encryption) can be initialized.</p> <p>Please note that the configure:all target implicitly calls configure:initialize.</p>

Setting Up the Database

The availability of a database is a prerequisite for ehf-based products. The following targets focus on setting up the database as required for the product to work as anticipated.

Target Name	Description
-------------	-------------

database:prepare	<p>Connects to the database to initialize the schemas and create the required tables for all the modules that are part of the distribution assembly.</p> <p>Since eHF 2.9 this goal supports schema pre- and suffixing. This feature is largely module and assembly specific. Also the properties to facilitate schema pre- and suffixing are defined by the application and may vary.</p>
database:privileges	<p>Connects to the database to create an eHF database user. It then creates users for every eHF module and grants privileges.</p> <p>After this target has been executed the database is ready to be used.</p> <p>Since eHF 2.9 this goal supports schema pre- and suffixing.</p>
database:bootstrap	<p>Connects to the database to import bootstrap data. The bootstrap data is the minimum set of data that is required for the product to function.</p> <p>Since eHF 2.9 this goal supports schema pre- and suffixing.</p>
database:import-content	Apart from the bootstrap data bigger content can be imported or refreshed using this target.
database:import-testdata	Imports test data into the database. This target should not be used on a production, but rather only on test systems.

Updating / Patching Data in the Database

In case you are installing a new patch-level update (that is a change in the third digit of the version number) of a product that is already operational you may need to perform an update of the bootstrapped data. No structural database changes are applied in this procedure. The following goals support this.

Target Name	Description	Parameters
update:execute	Executes a sequence of bootstrap activities to update the data in the database. The sequence can be selected by providing the <code>version</code> parameter.	<code>version</code> : A version key that is supplied by the provider of the distribution. The version key is internally mapped to a sequence of bootstrap configurations dedicated to a specific software update.
patch:execute	Executes a sequence of bootstrap activities to patch the data in the database. The sequence can be selected by providing the <code>version</code> parameter.	<code>version</code> : A version key that is supplied by the provider of the distribution. The version key is internally mapped to a sequence of bootstrap configurations dedicated to a specific software update.

		configuration dedicated to a specific software patch.
--	--	---

Upgrading Data in the Database

In case you are installing a new minor or major version of a product that is already operational you typically have to perform an upgrade of the underlying data, possibly including updates to the database structure. The following targets support these activities.

Target Name	Description	Parameters
upgrade:configure	Configures the input used by the upgrade procedure.	None.
upgrade:execute	Executes a single upgrade step. The upgrade step must be provided as parameter.	task: the task to execute. Your product provider will deliver the list of upgrade tasks together with the product's documentation. Please note that the upgrade supports local and remote tasks. Local tasks have to be prefixed with 'local:' while remote tasks have to be prefixed with 'remote:'.
upgrade:prepare-remote-host	In case remote tasks are part of the upgrade procedure the remote host has to be prepared using this target. The preparation mainly consists of a transfer of script files to the remote host where the database is located.	None.
upgrade:finalize-remote-host	After all upgrade steps have been executed the remote host can be restored by calling this target. The scripts that were copied by the upgrade:prepare-remote-host task are removed.	None.
upgrade:all	Runs a fully automated upgrade procedure. This target is considered a development target and enables continuous integration of the upgrade procedure. In production this target should be used with care.	None.

Encrypting the Database

Applications in health care have to satisfy high data security demands. eHF provides many components to meet these demands. Database-level encryption is one of the measures

that is supported by eHF to protect sensitive data. The following goals support the activation of database level encryption.

Please note that this feature is currently only supported in conjunction with an Oracle database that supports Oracle TDE and Oracle DMBS_CRYPTO.

Target Name	Description	Parameters
encrypt:configure	Configures the input used by the encryption procedure.	None.
encrypt:execute	Executes a single encryption step. The encryption step must be provided as parameter.	task: the task to execute. Your product provider will deliver the list of upgrade tasks together with the product's documentation. Please note that encryption supports local and remote tasks. Local tasks have to be prefixed with 'local:' while remote tasks have to be prefixed with 'remote:'.
encrypt:prepare-remote-host	In case remote tasks are part of the encryption procedure the remote host has to be prepared using this target. The preparation mainly consists of a transfer of script files to the remote host where the database is located.	None.
encrypt:finalize-remote-host	After all encryption steps have been executed the remote host can be restored by calling this target. The scripts that were copied by the encryption:prepare-remote-host task are removed.	None.
encrypt:all	Runs a fully automated encryption procedure. This target is considered a development target and enables continuous integration of the encryption procedure. In production this target should be used with care.	None.

11 Continuous Integration

The eHF Continuous Integration infrastructure is made up of three main concepts: Build Server, Deploy Server, and eHF Routines.

11.1 Build Server

The Build Server is a dedicated machine that does nothing else than continuously build and integrate the software modules upon changes in the source code. It also runs nightly test deployments and nightly tests against deployed applications. This is taken care of by a continuous integration software called Hudson (for more information please refer to <https://hudson.dev.java.net/>).

11.2 Deploy Server

A deployment is a complete installation of the eHF on a dedicated deploy server. The only software that is assumed to be installed on the deploy server is Java and an Oracle database, in particular a deployment involves the following steps:

1. Installation of infrastructure components (Apache Tomcat, Apache HTTPD)
2. Setup of the database (creation of database schema and tables)
3. Creation and deployment of the eHF WAR (building an eHF Assembly)

There are three different types of deployments:

Deployment Type	Description
Standard Deployment	Normal deployment with the steps described above
Cipher Deployment	Normal deployment followed by the encryption of the database
Upgrade Deployment	<ol style="list-style-type: none"> 1. deployment of selected previous eHF release (eHF WAR and according database), 2. insert of some test data into the database, 3. execution of an upgrade to the current eHF database schema 4. running the test client of the previous eHF release against the upgraded database

Table13. Deployment Types

Please note that the upgrade deployment already contains some simple data consistency checks.

11.3 eHF Routines

The eHF Routines encapsulate all steps that are needed to:

- deploy an eHF
- run an upgrade on an eHF installation

- run test (WebService test with the eHF WS Testclient)

For having a look at the source code please check out <https://cvs.de.icw.int/subversion/bas/production/ehf/test/routines/trunk/routines>.

11.4 Continuous Integration Infrastructure in Action

This section describes how the three concepts fit together in the different use cases. The deployment is triggered by the **Build Server** (by Hudson). Hudson uses the **eHF Routines** to run the deployment. The **eHF Routines** encapsulate all the steps needed to setup an eHF on the **Deploy Server**.

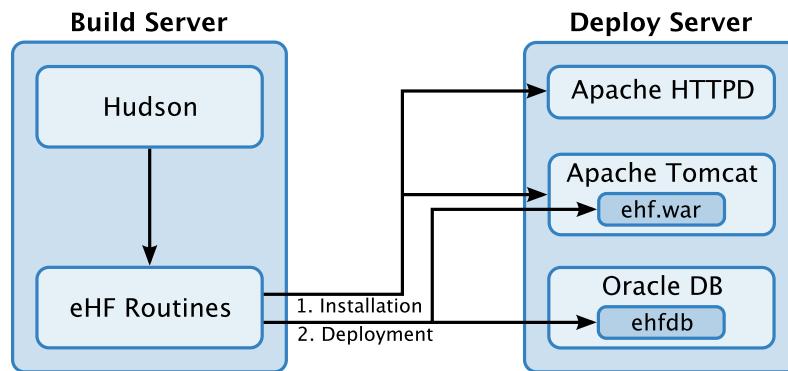


Figure 30: Deployment

The WS Test is also triggered by the **Build Server** (by Hudson), which uses the **eHF Routines** to launch the WS Test (using the eHF WS Testclient) against the same **Deploy Server** eHF was deployed on earlier.

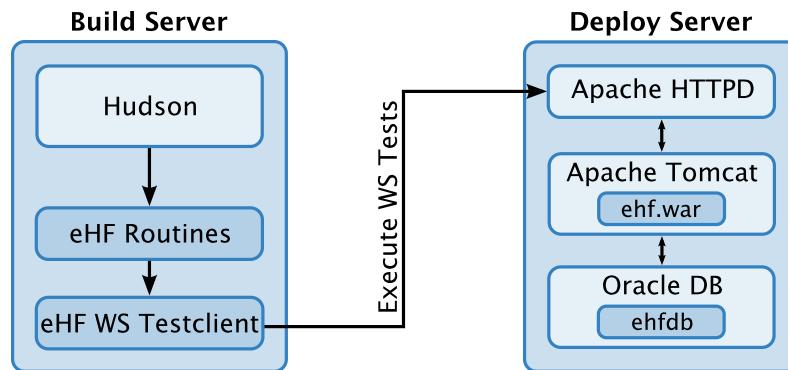


Figure 31: Test Automation as part of Continuous Integration

12 Generator

The eHF Generator represents the backbone of many eHF concepts and drives the evolution of the overall eHF architecture by using the Model-driven software development (MDSD) approach.

MDSD is a development technique that allows you to automatically generate executable software from UML models. The eHF Generator automates the creation of eHF-based modules including the full persistence data model, the appropriate service layers and integration layers. Custom service methods can be added when required. The generator approach abstracts from the technical details of the architecture, emphasizing instead on the domain model, the accompanying business logic and services. The nature of this chapter is to explain what kind of features the eHF Generator offers, how it is used, and how it can be extended.

12.1 Characteristics of the Generator

The eHF Generator is comparable to a compiler. The compiler usually reads a source artifact of a higher abstraction level and creates a compiled artifact of a lower abstraction level. You can see a very simple graphical representation of the compiler approach in [Figure 32](#).



Figure 32: Principle of a Compiler

In Model-driven software development, the generator represents the compiler. The model is the corresponding source artifact that can be described in UML or other languages. The generator automatically performs a series of transformations. The resulting source code, configuration files and descriptors represent the compiled artifacts.

12.1.1 openArchitectureWare

[openArchitectureWare](#) (oAW) is an open source generator framework. Using the above compile analogy, oAW defines the *compiler* in detail. The adaptation of the tool is possible in a variety of manners, including the adaptability to any kind of modeling tool, the extensibility in terms of custom models, parsers, meta model classes, as well as templates. [Figure 33](#) shows the oAW architecture.

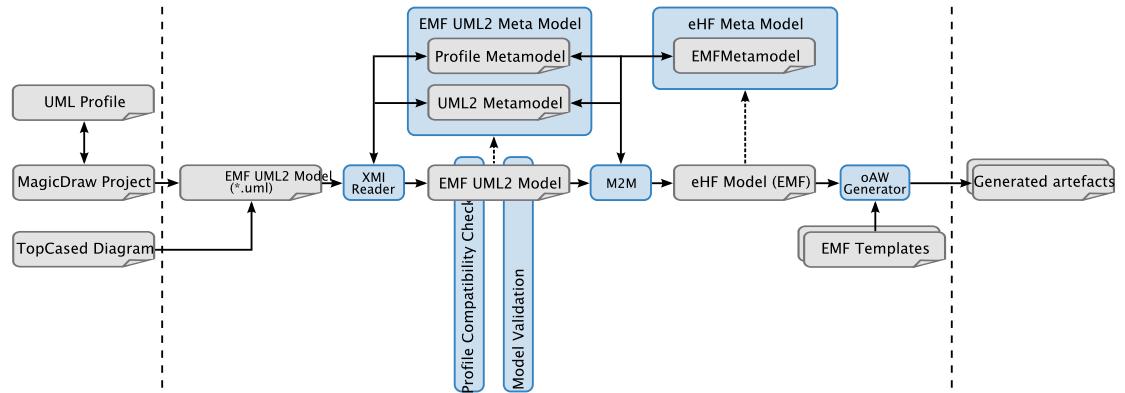


Figure 33: Architecture of openArchitectureWare

The represented meta-model instance is created from the UML model. The openArchitectureWare is able to instantiate any kind of model with a suitable set of parser and instantiator. The generator creates the generated code using templates based on the Java meta-model instance. Depending on the templates, any kind of text output can be generated.

12.1.2 Xpand Templates

The special language Xpand is used by openArchitectureWare to define templates that control the output of the code generation.

Templates are saved in files with the extension .xpt and have to be located on the Java classpath of the eHF Generator. The following is an example of the general structure of a template:

```
«EXTENSION extensions::Functions»

«IMPORT com::icw::ehf::generator::metamodel»

«DEFINE javaClass FOR Entity
    «FILE javaFilename()»
        package «javaPackage()»;

        public class <<className>> {
            //implementation
        }
    «ENDFILE»
«ENDDEFINE»
```

A template consists of any number of IMPORT and EXTENSION statements followed by one or more DEFINE blocks (called definitions). If your template contains an IMPORT statement, you can use the unqualified names of all types and template files contained in that namespace. This is similar to a Java import statement `import com.icw.ehf.generator.metamodel.*`. Extensions provide a way of defining additional features of meta classes. The example uses the extension file `extensions/Functions.xpt`. Please note that extension files also have to reside on the Java classpath.

The concept of Xpand is centered on the DEFINE block. This is the smallest identifiable unit in a template file. The tag consists of a name, an optional comma separated parameter list and the name of the meta model class for which the template is defined. The body of a template can contain a sequence of other statements including any text.

In addition to the basic statements, Xpand supports a full parametric polymorphism. This means that it is possible to have two templates with the same name that define two meta

classes, which are inherited from the same super class. When the template is called for the super class, Xpand will use the corresponding subclass template. In case a subclass template is not available, the super class template will be used.

Please refer to the [Xpand Language Reference](#) for more information on Xpand Templates.

12.1.3 Object Constraint Language

A UML diagram is typically not precise enough to provide all the relevant aspects of a specification. There is a need to describe additional constraints about the objects in the model. In a requirement specification such constraints are often described in natural language. However, this will always result in ambiguities. The Object Constraint Language (OCL) is meant to resolve these ambiguities. It is a formal language that remains easy to read and write.

The OCL can be used to accomplish several tasks. In the eHF Generator module, it is used to specify invariants for validating domain objects. In particular, it is not possible to save a domain object to the persistent store that does not pass the validation of the defined constraints.

The following example shows an OCL invariant constraint of the domain object `Allergy` of the module eHF Medical Record:

```
self.effectiveTime.beginDate.canonicDate < ${currentTime}
```

The constraint is defined as an annotation of the domain object. `self` refers to an instance of the domain object, `effectiveTime` is a field of this instance, and `currentTime` is an OCL function that returns the current time in milliseconds.

The common approach to OCL expressions is that they are evaluated during the generation of code artifacts. An important feature of the eHF is that OCL expressions are evaluated at runtime. This enables the specification of additional OCL constraints after the generator run is completed and the eHealth Framework modules are deployed. Thus, the application developer can subsequently add further constraints.

12.1.4 Generated Artifacts of the Architectual Layers

The eHF has adopted a domain object centered approach. It focuses on the business logic instead of the infrastructure code. The eHF Generator supports this by generating the infrastructure code based on the modeled domain objects. Specifically, the eHF Generator creates implementations for the following architectural layers of an eHF module:

Core Module

- EJB 3.0 Entity Beans representing the persistent domain objects
- AspectJ security aspects
- Domain Object Meta Data
- DAO Interfaces and HibernateDao implementations
- ObjectManagers supplementing the functionality of the persistence layer
- Typed CRUD (Create – Read - Update – Delete) interfaces and CRUD service implementations
- Spring and Hibernate configuration files

Internal API

- Data Transfer Objects (DTOs)
- Transfer Object Assemblers that convert the domain objects from/to DTOs

External API

- External Data Transfer Objects (XTOs)
- Transfer Object Assemblers that convert the domain objects from/to XTOs

- Web Service API

Additions

- Eclipse Modeling Framework (EMF) based adapters to the domain model for evaluation of OCL expressions.

12.2 Configuration of the Generator

The openArchitectureWare workflow engine is a declarative configurable generator engine. Beside the domain model, the generation output is controlled by the generator invocation and module specific properties. Both will be introduced in the following.

12.2.1 Specifying the Generator Invocation

openArchitectureWare provides a simple XML-based configuration language. The workflow file (with extension .oaw) specifies the generator input. The following listing shows an example:

```
<?xml version="1.0" encoding="windows-1252"?>
<workflow>

    <!-- default value for basedir -->
    <property name="basedir" value=". "/>

    <!-- invoke the generator (Root::Root) -->
    <cartridge file="oaw/ehf-generator.oaw">
        <model value="${basedir}/src/main/model/model.xml" />
        <srcGenPath value="${srcGenPath}" />
        <srcMainPath value="${srcMainPath}" />
        <templateEntry value="Root::ObjectModel" />
        <codeSystemMappingFile
            value="${basedir}/src/main/model/codesystem-
            mapping.properties" />
        <globalCodeSystemMappingFile value="${basedir}/
            src/main/model/global-ehf-mappings.properties" />
        <moduleConfigFile value="${basedir}/module.configuration.properties" />
    </cartridge>

    <!-- invoke the generator to generate EMF adapter classes -->
    <cartridge file="oaw/ehf-generator.oaw">
        <model value="${basedir}/src/main/model/model.xml" />
        <srcGenPath value="${srcGenPath}" />
        <srcMainPath value="${srcMainPath}" />
        <templateEntry value="Root::Emf" />
        <moduleConfigFile value="${basedir}/module.configuration.properties" />
    </cartridge>

</workflow>
```

The above workflow invokes the generator twice. The first step generates the complete module infrastructure. The second step adds the generation of EMF classes. The main difference between the two invocation steps is the property *templateEntry*, which defines the entry point in the generator templates. [Table 14](#) summarizes the template entry points currently offered along with their description. Please note that there are several other template entry points. However, they may be subject to change and are therefore not regarded as public.

Entry Point	Description
Root::Root	Generates the complete module (except EMF) from persistence to adapter, web user interface and web service classes, configuration, etc.

Entry Point	Description
Root::FullModule	Same as <i>Root::Root</i> , but includes EMF. This is the recommended entry point that matches most of the use cases of the eHF Generator.
Root::CoreModule	Generates only the core module as described in the introduction. No adapters, user interface classes and web service implementations are provided.
Root::Emf	Generates EMF adapter classes based on the domain objects. The EMF classes can be used for checking OCL constraints.

Table14. Publicly available Template Entry Points

If the generated model uses code systems (see [Code Systems](#) on page 14), code system mapping files have to be provided. They describe the mapping from code system names to code system identifiers. The generator workflow configuration hereby distinguishes between local and global code system mapping files. In the latter case, you are able to provide multiple comma separated global mapping files. For example one for each product line defining code systems.

12.2.2 Defining the Template Version

The eHF Generator provides an infrastructure for versioned templates. This infrastructure enables the the templates to be changed without breaking existing, already generated modules. The modules need to be explicitly upgraded to use a newer template version.

In order to use a specific template you have to ensure that the workflow file contains the line:

```
[...]
<moduleConfigFile value="${basedir}/module.configuration.properties"/>
[...]
```

This is the prerequisite for using versioned templates. Now you can control the version of the template to be used by adding the following lines to the `module.configuration.properties` file:

```
ehf.generator.dao.version=2.0
ehf.generator.auditable.version=2.0
ehf.generator.webapi.version=2.1
ehf.generator.persistence.jointable.version=2.9
ehf.generator.persistence.convention.version=2.1
ehf.generator.persistence.aggregate.version=2.0
ehf.generator.security.annotations.version=2.7
ehf.generator.input.validation.version=2.9
ehf.generator.persistence.inheritance.annotations.version=2.9
ehf.generator.service.version=2.9
```

Please consult the [Appendix](#) on page 325 for details on the available versioned templates.

12.2.3 Defining the Module Prefix

In addition to versioning, you are able to define a module prefix for each module via the `module.configuration.properties` file.

```
# define module prefix
ehf.generator.module.prefix=record-admin
```

The given module prefix will be attached to all generated Spring and web service configuration file names. This allows custom definition of your configuration files. If this property is not set, the model name of your UML model is taken as the default module prefix.

12.3 Extension Points of the generated Artifacts

The eHF Generator generates source code and configurations files. In order to be able to extend the generated artifacts, several concepts and extension points have been identified during the design of the generator. The following will describe those concepts in detail.

12.3.1 Moveable generated Code

A common generator throws away and re-generates all artifacts on each individual generator run. One of the important features of the eHF Generator is that some files are generated once and then left under control of the developer. To keep these files separate from the throw-away artifacts, they are generated into the appropriate locations in your project. One-time generated Java source code is located under `src/main`, while one-time generated configuration is located under `src/config`. Throw-away artifacts are generated under `src/gen`.

In general, all code/configuration that is not generated in `src/gen` is under the control of the developer. None of the files in `src/gen` should be modified, since the changes will be lost during the next generator run.

The classes which are generated once is determined by the generator and can be influenced for selected classes (see [Appendix](#) on page 325).

12.3.2 Extending the generated Spring Configuration

The generator creates a `<modulename>-custom-context.xml` in the `src/config` folder. Here you are able to provide additional Spring bean definitions and tweak the throw-away configuration parts.

Since Spring is vastly used to wire the modules layers together a special customization approach is taken to modify and extend the generated Spring configuration. The eHF Commons Project (TODO REF) (which is the basis for all generated modules) provides a special Spring postprocessor. This postprocessor can be used to further modify the existing spring configuration or to inject an additional service (i.e. an external service) in your module service:

```
<bean id="moduleServiceServices"
    class="com.icw.ehf.commons.spring.PropertySetBeanFactoryPostProcessor">
    <property name="targetObject" ref="recordModuleServiceTarget" />
    <property name="values">
        <map>
            <entry key="personService"
                  value-ref="usermgntPersonService" />
        </map>
    </property>
</bean>
```

13 Life Cycle Support

13.1 Web Service Backwards Compatibility

[Backwards Compatibility](#) on page 20 covered the basic concepts of versioning schemes. This section goes into the implementation details of web service backwards compatibility and explains how you can specifically create backwards compatible modules.

The [WSDL 1.1 specification](#) does not address the issue of web service versioning. Once you expose a web service to the outside world and have to do subsequent modifications or extensions, your web service becomes backwards incompatible. For example, even very small changes such as adding a new attribute will cause the web service to be incompatible. Consequently, whenever a new web service is released, all existing client applications have to be updated immediately.

To overcome this, the eHF provides the message-based backwards compatibility layer for web services that is illustrated in [Figure 34](#).

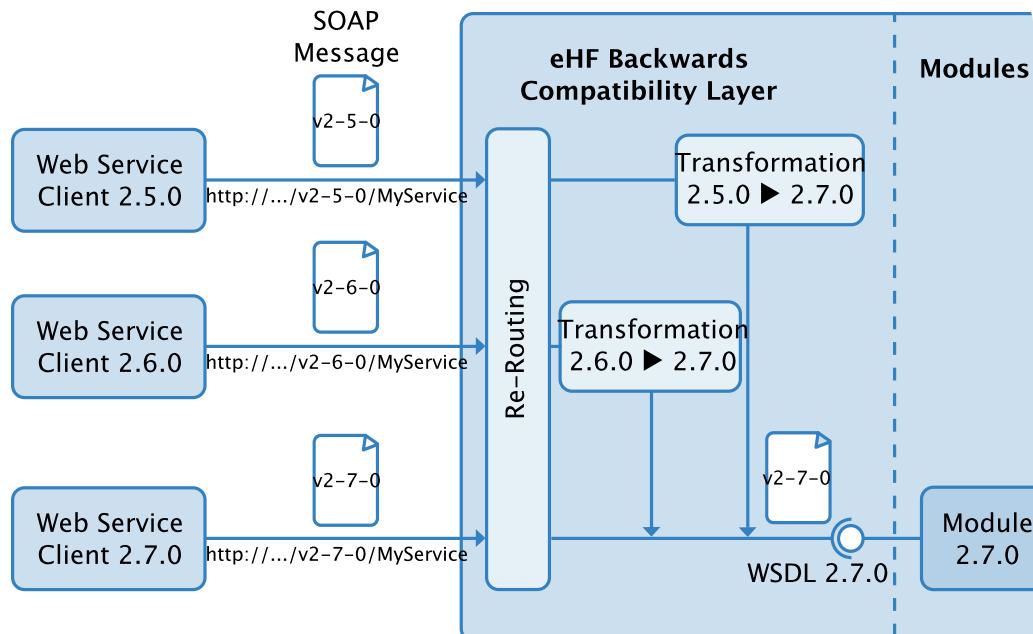


Figure 34: Principle of the eHF Backwards Compatibility Layer of Web Services

The eHF applies the concept of versioning schemes to SOAP messages to ensure backwards compatibility. The version numbers are contained in the caller URL of the web service and in the namespace of the SOAP message. A typical URL for an eHF web service, for instance, is <http://myserver/ehf/services/v2-7-0/DocumentWebService> and the corresponding namespace is <http://ehf.icw.com/document/v2-7-0/service>.

The eHF backwards compatibility layer has the responsibility of re-routing web service requests and transforming SOAP messages, before the web service receives the SOAP request. If the SOAP message is of the same version as the installed web service, the eHF backwards compatibility layer simply routes the message to the web service.

If a new version of the eHF is delivered, the old version of the web service is replaced with a new one that also contains a new version number. However, the web service client still sends SOAP message requests with the old namespace to the old URL. After that, the following will happen. The eHF backwards compatibility layer:

1. determines the version of the calling client according to the version in the called URL (that is, a 2.5.0 client will call a v2-5-0 web service),
2. checks that the version in the calling URL matches the version of the SOAP message namespace,
3. transforms the request to the latest available version (that is, from v2-5-0 to v2-7-0) before routing the request to the web service,
4. applies the configured request transformations before routing the request to the web service,
5. processes the request using the latest web service implementation,
6. then applies the configured response transformations respectively (that is, from v2-7-0 to v2-5-0) before answering the call.

At the same time request messages are being transformed from the lower version number to the latest available version number, the response messages are being transformed from the highest version number into the version number of the calling client.

Adding a SOAP Message Transformation

So, how exactly does it work? What do you need to configure in order for it to work properly? A servlet re-routes the SOAP messages to the web service. The transformation mechanism is based on the [Extensible Stylesheet Language](#) (XSL). These transformations modify the requests from and responses to a web service client that has been created against an older version of the web service. If you develop your own modules with web service support, you first have to configure the file `axis.compatibility.fragment` in the directory `/src/main/config/merge/assembly`. This file is merged into the file `/src/main/config/axis-service-config.xml` of the assembly module (see also [Tokens](#) on page 57). The file starts with the URL path to the web service (`servicePath`) including the property token for the current version number that is replaced at build time and follows with the transformations for upgrading the SOAP messages to the current WSDL. The structure of the file is as follows:

```

<service>
  <servicePath>/@{webapi.version}@/MyService</servicePath>
  <transformations>

    <transformation>
      <servicePathPattern>/v2\-\d\-[0-9]*\d\-/MyService</servicePathPattern>
      <requestTransformation>v2.5.0-v2.6.0-request.xslt</requestTransformation>
      <responseTransformation>v2.5.0-v2.6.0-response.xslt</responseTransformation>
    </transformation>

    <transformation>
      <servicePathPattern>/v2\-\d\-[0-9]*\d\-/MyService</servicePathPattern>
      <requestTransformation>v2.6.0-v2.7.0-request.xslt</requestTransformation>
      <responseTransformation>v2.6.0-v2.7.0-response.xslt</responseTransformation>
    </transformation>

    [...]
  </transformations>
</service>

```

For each backwards compatible version, you have to provide a transformation definition composed of:

servicePathPattern

This is the relative path to the called URL of the backwards compatible web service. If the called URL matches the regular expression, the transformation is executed for the request and response messages.

requestTransformation / responseTransformation

This is the path to the corresponding style sheet. Each file contains a style sheet transformation of the request or response from the web service interface defined by one version of the service to that of another subsequent version. The files are usually located in the directory `/src/main/resources/META-INF/compatibility` and are referenced relative to the classpath (for example `META-INF/compatibility/<ModuleId>-v2.6.0-v2.7.0-response.xsslt`). Please note that you can also use file references such as `file:///.../<ModuleId>-v2.6.0-v2.7.0-response.xsslt`. In this case, the style sheets are not cached and can be modified without restarting the container.

The following code shows an example of style sheet transformation for a SOAP response message. The current web service has a new attribute `filename`. The template (1) deletes the new attribute from the SOAP message and (2) copies all other elements of the SOAP message.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <xsl:output method="xml" encoding="UTF-8" />

  <!-- (1) remove the new filename attribute -->
  <xsl:template match="*:filename" />

  <!-- (2) This rule copies all elements except those
       with special template matches defined. -->
  <xsl:template match="@* | node()">
    <xsl:copy>
      <xsl:apply-templates select="@*| node() | text()"/>
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Chaining of SOAP Message Transformations

For clients who are using a version that is more than one version older than the current web service it is necessary to execute multiple transformations one after the other. This *chaining* is done by including in each XSLT the subsequent one, if it exists.

Regarding the example of [Figure 34](#), the transformation from 2.5.0 to 2.6.0 includes the transformation style sheet from 2.6.0 to 2.7.0:

```
<xsl:include href="/META-INF/compatibility/<ModuleId>-v2.6.0-v2.7.0
              -response.xsslt"/>
```

These includes have to be repeated until the last transformation style sheet.

At the end of the latest transformation, you will find the "identity transform" rule, that is the one with `match="@* | node()"`. This rule must be defined for every transformation chain, but it must only be defined once, otherwise unwanted warnings are generated. Therefore, it should always be moved to the latest XSLT file when adding a new one. Please pay attention to the fact that XPATH 2.0 / XSLT 2.0 is required for the wildcards of the match attribute.

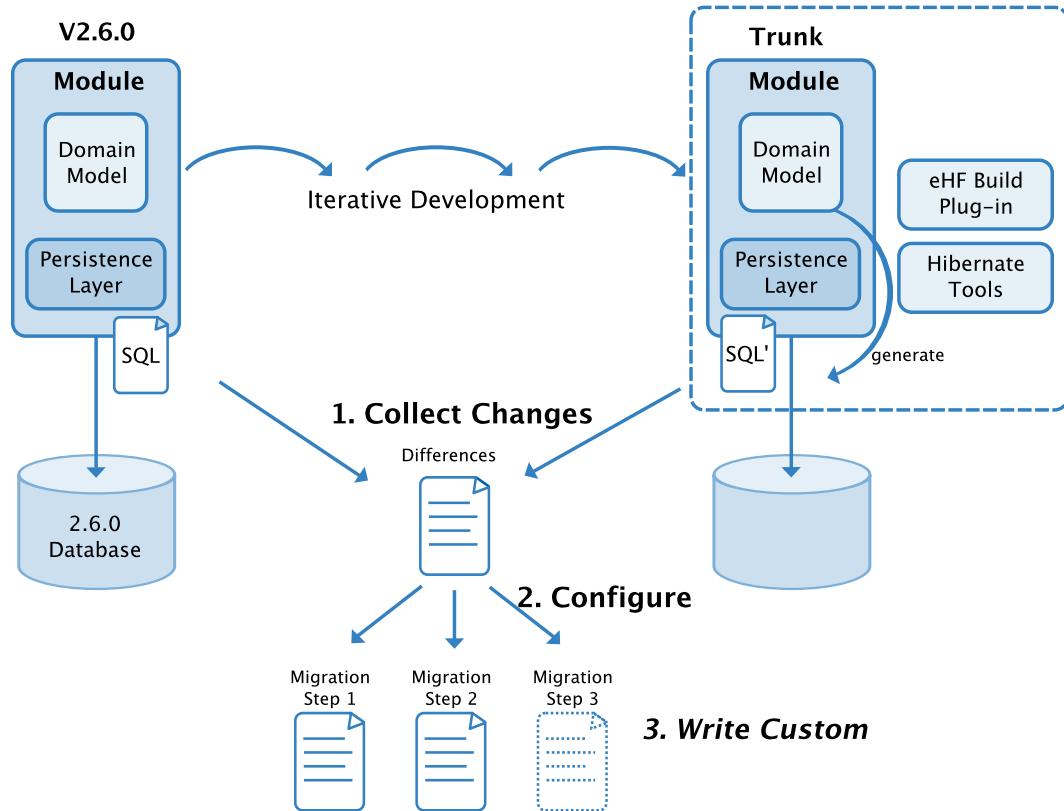
13.2 Database Upgrade

[Upgrading](#) on page 21 covered the basic concept of the upgrade process. The upgrade process consists of several steps that are divided into *upgrade preparation* and *upgrade execution*. The upgrade preparation requires the collecting of the database changes since the last release, the configuring of necessary migration steps, and possibly writing custom migration steps for special purposes. The upgrade execution requires the configuration of the environment dependent properties and the execution of the migration steps. While an upgrade is prepared in the local development environment, the upgrade itself will usually be executed time-delayed in the staging environment. The following describes both in detail.

13.2.1 Upgrade Preparation

Among other things, a service module contains a domain model and the binding to the corresponding database schema through the persistence layer. Since the eHF applies Model-driven software development (see [Model-Driven Software Development](#) on page 12), the changes of the domain model are updated automatically to the persistence layer. The eHF Build-Plug-in (see [eHF Build Plug-in](#) on page 70) generates with the help of the Hibernate Tools (<http://www.hibernate.org/255.html>) a database creation SQL file. The database schema is dropped and newly created. Therefore, the database structure is usually up-to-date and synchronized with the domain model in development environment.

In the example of [Figure 35](#), the module V2.6.0 is the latest released module. While the version 2.6.0 remains stable, the development of new features or change requests proceeds on the current version, called *Trunk*. In particular, this causes continuous changes of the database. At a later point in time, a new version of the module will need to be released and the upgrade has to be prepared.

**Figure 35:** Upgrade Preparation

Collecting Changes

The first step is to collect all database changes with the help of a difference analysis tool before packaging the release. One advantage of this procedure is that the developers do not have to permanently have the upgrading issue in mind. Instead, they can first concentrate on new features of the current version and refactorings. Afterwards, at a later point in time, they can focus explicitly on the needed migration steps that are the result of the changed version of a module.

One technique for collecting the database changes is to analyze the generated SQL files:

1. Check out the two versions of the eHF module in question (that is, V2.6.0 and *Trunk*).
2. Build each version (using *maven dev:build*).
3. Compare the files `init.sql`, `schema-create.sql`, and `schema-custom.sql` in the directory `src/main/gen/META-INF/db/<DB-dialect>` from the two versions.

`init.sql` creates the initial schema/user in the database, `schema-create.sql` creates the content of the schema (for example tables, triggers, indexes), and `schema-custom.sql` allows custom SQL statements. The analyzing of the differences between the SQL scripts is one possibility. Another alternative is the use of database tools that allow the analyzing of differences with more ease. However, no matter whether you analyze the changes with the help of the SQL scripts or a third party database tool, you have to consider the following kind of changes:

Structural Changes:	<ul style="list-style-type: none"> • Columns - Create, Delete, Rename, Alter Width • Tables - Create, Delete, Rename
---------------------	--

	<ul style="list-style-type: none"> Indexes - Create, Delete Constraints - Create, Delete Triggers - Create, Delete Schema - Create, Delete, Rename, Split
Data Changes:	<ul style="list-style-type: none"> Initial values Value changes (of code systems and permissions for instance)
Integrity Checks:	<ul style="list-style-type: none"> New or changed constraint Other assumptions (such as a table is not used anymore and can be completely dropped)

Configuring Migration Step

CAUTION: Log4J Configuration

Due to the multi-threaded nature of the migration tool it requires a specific Log4J Appender. As such it looks for a Log4J appender named "**MIGRAPP**". Please make sure to add this appender to your Log4J configuration as follows:



```
<!-- ===== -->
<!-- Composite appender for upgrade -->
<!-- ===== -->
<appender name="MIGRAPP" class="com.icw.ehf.commons.logging.
CompositeFileAppender">
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d{ISO8601} %-5p [%t] %C#%M(%L) -
%m%n"/>
    </layout>
</appender>
```

The eHF Build Plug-in and the eHF Migration Tools control the upgrade process. They are parameterized by a set of properties and configuration files. The next step of the upgrade preparation is the setup of the migration steps dependent on the analyzed changes. You configure the migration steps in XML files that are the main input to the eHF Build Plug-in. The XML files have the following structure:

```
<migration-configuration>
[...]
<mode>@upgrade.mode@@</mode>
<sql-output-filename>@upgrade.path@@/sql-output.txt</sql-output-filename>
<log-filename>@upgrade.path@@/logs/log-sql-scripts.txt</log-filename>

<db-connection>
    <driver>@connection.driver@@</driver>
    <url>@connection.url@@</url>
    <database>@database.type@@</database>
</db-connection>

<migration-steps>
    <!-- Migration Step 1 -->
    <!-- Migration Step 2 -->
    <!-- Migration Step 3 -->
```

```
[...]
</migration-steps>
</migration-configuration>
```

The first part of the file contains common configuration and the database connection configuration. *mode* defines, whether the SQL statements are executed during the upgrade or just logged to the file configured in *sql-output-filename*. As in the example, property tokens (see [Tokens](#) on page 57) define most of the configuration information. The property tokens will be replaced during the build by their corresponding values that are defined in environment dependent property files.

The next part of the XML file then defines the actual migration steps. The eHF Migration Tools provide a number of ready to use migration steps that can be configured in these files. The following three migration steps are usually sufficient for most upgrade tasks:

SQL Execution Step

Generally most upgrade scenarios such as adding a column or index can be carried out via simple SQL statements. These SQL statements are stored in *.sql files. The eHF Migration Tool parses and executes the SQL via a JDBC connection against the target database, using one of the eHF users in the database. This step has three configuration parameters:

- *filename* - The file containing all the upgrade SQL scripts. Many files can be configured for one SQL Execution Step.
- *username* - The database user for the database connection.
- *password* - The password for the database connection.

Column Data Integrity Check Step

This step has the following configuration parameters:

- *data-integrity-check-sql-filename* - This file should only contain SQL SELECT statements.
- *integrity-output-filename* - This file is used by this step to store any output generated by the *data-integrity-check-sql-filename*.
- *username* - The database user for the database connection.
- *password* - The password for the database connection.

The step executes all the SQL in *data-integrity-check-sql-filename*, and stores the output in *integrity-output-filename*. This file is then checked for any output and if there is any, the process will stop. The expectation is that the SQL statements will return an empty result. This step is usually used for checking the state of the database before performing modifications.

Replace Data Step

This migration step updates data within columns and has the following configuration parameters:

- *locations* - A location is a *table* and *column* configuration pair where the data change should occur.
- *replacements* - A replacement declares the update of a column from old to new values.

Via the properties *locations* multiple locations can be defined. The list *replacements* defines transformations from old to new values. This step performs an UPDATE statement for each location-replacement combination.

The following listing shows a configuration example of the three migration steps:

```
<migration-steps>
    <!-- 1. SQL Execution Step -->
    <com.icw.ehf.migration.steps.SqlExecutionStep>
        <filename>@upgrade.path@/ehf-document/1-BAS-896.sql</filename>
        <username>EHF_DOCUMENT</username>
```

```

<password>@@connection.document.pass@@</password>
</com.icw.ehf.migration.steps.SqlExecutionStep>

<!-- 2. Column Data Integrity Check Step -->

<com.icw.ehf.migration.steps.ColumnDataIntegrityCheckStep>
  <data-integrity-check-sql-filename>@upgrade.path@@/
    check-data-integrity.sql</data-integrity-check-sql-filename>
    <integrity-output-filename>@upgrade.path@@/logs/
      check-data-integrity.log</integrity-output-filename>
    <username>EHF_USERMGNT</username>
    <password>@@connection.usermgnt.pass@@</password>
</com.icw.ehf.migration.steps.ColumnDataIntegrityCheckStep>

<!-- 3. Replace Data Step -->

<com.icw.ehf.migration.steps.ReplaceDataStep>
  <locations>
    <com.icw.ehf.migration.steps.config.Location>
      <table>EHF_RECORD_MEDICAL.T_ENCOUNTER</table>
      <column>C_TYPEKEY</column>
    </com.icw.ehf.migration.steps.config.Location>
  </locations>

  <replacements>
    <com.icw.ehf.migration.steps.config.Replacement>
      <oldValue>fitness-group</oldValue>
      <newValue>110</newValue>
    </com.icw.ehf.migration.steps.config.Replacement>
    <com.icw.ehf.migration.steps.config.Replacement>
      <oldValue>fitness-group_end_wo</oldValue>
      <newValue>110.110</newValue>
    </com.icw.ehf.migration.steps.config.Replacement>
  </replacements>

  <username>EHF_RECORD_MEDICAL</username>
  <password>@@connection.record-medical.pass@@</password>
</com.icw.ehf.migration.steps.ReplaceDataStep>

</migration-steps>

```

Writing custom Migration Steps

Although the mentioned migration steps are sufficient for the most cases, you have the possibility to extend the eHF Migration Tools with your own custom migration steps. [Figure 36](#) shows the `MigrationStep` class hierarchy. The method `process()` is the starting point for the migration step. The parameter `MigrationConfiguration` contains the common configuration parameters of the migration step. You implement the interface `SqlMigrationStep` for migration steps that require a database user and password, otherwise implement `MigrationStep`. Please note that this custom class has to be contained in the classpath of the migration execution environment.

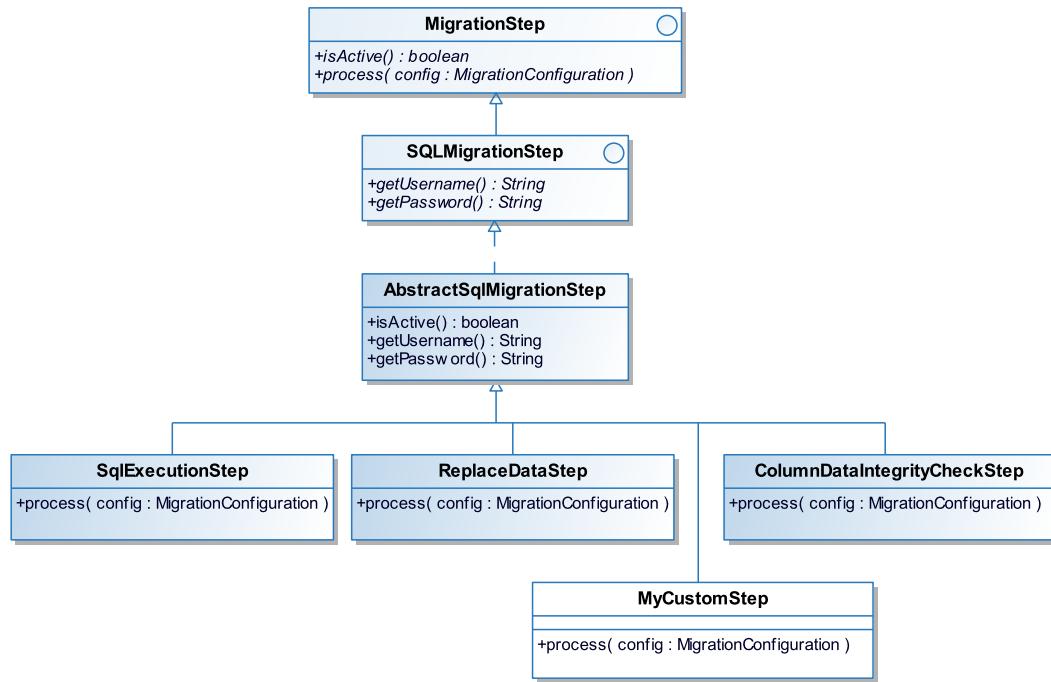


Figure 36: MigrationStep class hierarchy for implementing custom migration steps

Creating Upgrade scripts in a Module

Since eHF 2.9, Upgrade scripts are supported in a module and aggregated during creation of a release artifact for later execution. In this case, the SQL scripts and Migration Steps are contained in Module and referenced from the assembly. The directory structure should be the same in the module as the one used in the assembly i.e. configuration/database/upgrade/ehf-[version].

For example, if you want to add a new upgrade script to the Codesystem module for eHF-2.8-2.9, then you should at the script in the module under configuration/database/upgrade/ehf-2.8-2.9/ehf-codesystem and add the migration step to configuration/database/upgrade/ehf-2.8-2.9/upgrade-ehf-codesystem.xml.

Next you need to make sure the upgrade-ehf-codesystem.xml is included by the upgrade.tasks in the assembly. This is described in the following section.

Executing tasks in parallel

The upgrade process is usually divided into long-running tasks, which require different types of resources. While encryption-related tasks are CPU-intensive, the bottleneck of SQL scripts is the database. The order of many of the tasks does not depend on the others. This makes it more efficient to parallelize the execution in order to speed up the process.

Executing arbitrary tasks in parallel is done using the `<parallel>` tag. It is a container task, which can group tasks, which execute simultaneously. The parallel container task is only finished when the individual containing tasks are finished. If any of the tasks fails, the parallel task also fails immediately. For instance, to execute 3 tasks in parallel, you need the following migration configuration:

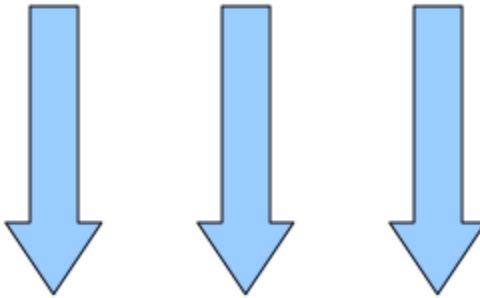
```

<parallel>
    <com.icw.ehf.migration.steps.MigrationConfigStep>
    ...
    </com.icw.ehf.migration.steps.MigrationConfigStep>
  
```

```

<com.icw.ehf.migration.steps.MigrationConfigStep>
    ...
</com.icw.ehf.migration.steps.MigrationConfigStep>
<com.icw.ehf.migration.steps.MigrationConfigStep>
    ...
</com.icw.ehf.migration.steps.MigrationConfigStep>
</parallel>

```



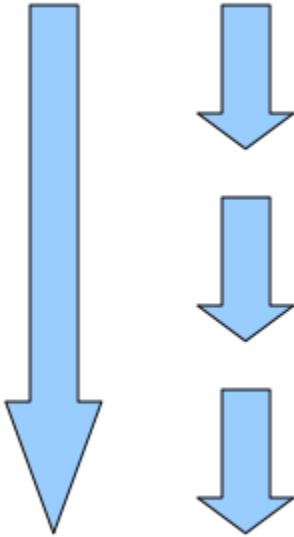
Sometimes all you want is for one long running task to execute in parallel to all other tasks, which execute sequentially. This can be achieved using the `sequential` tag. It can be nested inside a parallel task to indicate tasks which are to be executed sequentially. For example, to run a long-running task in parallel to 3 other tasks (which execute one after the other), the following configuration can be used:

```

<parallel>
    <com.icw.ehf.migration.steps.MigrationConfigStep>
        ...
    </com.icw.ehf.migration.steps.MigrationConfigStep>
    <sequential>
        <com.icw.ehf.migration.steps.MigrationConfigStep>
        ...
        </com.icw.ehf.migration.steps.MigrationConfigStep>
        <com.icw.ehf.migration.steps.MigrationConfigStep>
        ...
        </com.icw.ehf.migration.steps.MigrationConfigStep>
        <com.icw.ehf.migration.steps.MigrationConfigStep>
        ...
        </com.icw.ehf.migration.steps.MigrationConfigStep>
    </sequential>

```

</parallel>



The parallel and sequential tags can be nested to an arbitrary depth, although it doesn't make sense to nest parallel in parallel, or sequential in sequential tasks.

The syntax and concept of the upgrade parallel tasks is very similar to Ant's parallel and sequential tasks.

- <http://ant.apache.org/manual/CoreTasks/parallel.html> ↗
- <http://ant.apache.org/manual/CoreTasks/sequential.html> ↗

Debugging tasks executing in parallel has specific challenges. In order to make the output of these tasks understandable and easy to read, the following additional configuration tags can be used:

log-filename

can be specified per task, overriding the setting in the migration configuration

step-name

can be specified to identify running times of tasks in output, log files and performance reports

sql-output-filename

can be specified for instances of SqlMigrationStep and override the settings in the migration configuration

Executing tasks in parallel will not guarantee correct execution if it works successfully sequentially. It is the responsibility of the performance expert to understand dependencies between tasks and determine which ones cannot be executed simultaneously.

Understanding performance reports

The performance report is aimed at helping pinpoint hot spots in the upgrade configuration in order to optimize its performance. The excel spreadsheet represents a tree of tasks and their subtasks. The subtasks of a task are displayed nested one column deeper and directly above the row of the parent task.

The performance report contains two identical sheets with different representations of time. One sheet contains human-readable times, while the other represents time in milliseconds - it is more appropriate to be further processed, e.g. for creating charts.

The configuration of parallel tasks is designed to be backwards-compatible with the existing migration configuration format. While a best effort is made to create a meaningful output for performance reports, specifying task names will improve readability of reports. If such a name is missing then the module name or associated SQL script file would be used (if applicable).

Performance reports are normally saved in the logs directory, where report has the name of the upgrade task configured in [Creating Upgrade scripts in a Module](#) on page .

13.2.2 Upgrade Execution

After the upgrade is prepared and packaged in a release, the upgrade will be time-delayed performed in the staging environment. The upgrade execution requires the configuration of the environment dependent properties and the execution of the migration steps.

Configuring the Application

The property files of the assembly module contain all kind of properties for setting up an eHF based application. The file `configuration.properties` is the master configuration file. This file includes other property files using the property `configuration.include`. The most important properties for the upgrade are declared in the file `configuration.product.instance.properties`. The following listing shows a fragment of the file:

```
# upgrade properties

# upgrade mode and path
upgrade.mode=SQL
upgrade.path=configuration/database/upgrade/ehf-2.9-2.10

# comma separated list of upgrade script source paths. This property is optional.
# Note: "upgrade.path" is always implicitly considered a source path
upgrade.merge.source.paths=configuration/database/upgrade/ehf-2.9-2.10,
configuration/database/upgrade/medCab-2.9-2.10

# maven upgrade:all task execution order (not used in production)
upgrade.tasks=\
    local:ehf-assembly-upgrade.xml, \
    local:ehf-assembly-upgrade2.xml
```

In this example, the `upgrade mode` is `SQL` and the main upgrade path is `configuration/database/upgrade/ehf-2.9-2.10`. `upgrade.merge.source.paths` defines two source paths for upgrade scripts. The first is the same as the main upgrade path and could actually have been left out, but does no harm being repeated here, while the second is a new source path. Finally you declare all the XML migration step files in the property `upgrade.tasks`.

These properties should be adapted at the environment in which the application is installed. The next step is to propagate the properties to the tokens of the migration scripts, which can be performed via the following target in the Ant install scripts `ant -f install/install.xml upgrade:configure`. The modified upgrade scripts are copied to an isolation directory.

Executing the Upgrade Steps

Now, you are ready to perform the upgrade. To execute the upgrade procedure, you could enter the following Ant command:

```
ant -f install/install.xml upgrade:execute -Dtask=local:<migration-step-file>.xml
```

However, this will only execute the migration scripts defined in this one XML file. As pointed out an upgrade requires usually more than one migration step. The eHF Ant install scripts also provides the target `ant -f install/install.xml upgrade:all` that will automatically run the upgrade based on all configured migration step files.

Part IV - Application Platform

The eHF application platform consists of business service modules and library modules that are built on top of industry-standard application frameworks and can be deployed on any J2EE 1.4-compliant application server.

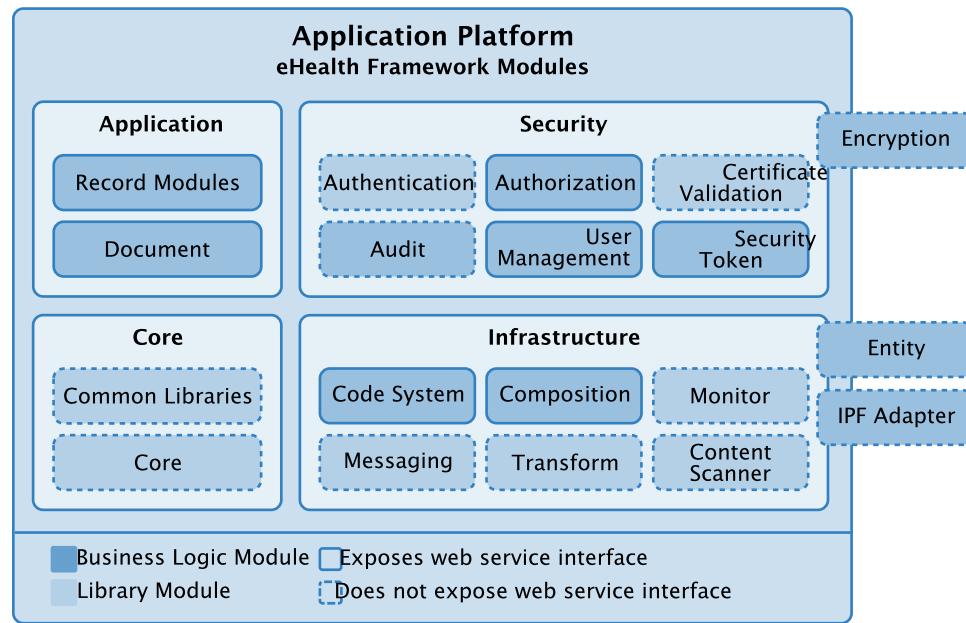


Figure 37: Application Platform Modules

The eHF modules are organized into four categories. [Figure 37](#) shows the categories Core, Infrastructure, Security and Application.

14 Core Modules

The Core modules establish the platform for other modules and provide basic functionalities. The Commons library module provides infrastructural features, which are used as a basis by individual business service modules. The Core library module hosts a general domain model, which is used throughout the framework.

14.1 Core

In order to prevent duplication of models and associated logic, the Core Library module defines domain objects shared among several modules. The domain model is enriched with domain-specific logic to further enhance the usage of these central model elements. Stored in a centralized module, such a collection of core classes simplifies and encourages reuse. Typical members of the Core Library module belong to the following data categories:

- Participants (Person, Address, ...),
- Terms (Date, Time, Time Interval, ...)
- Codes (Code System, Code, ...)
- Identifiers and Qualifiers (Reference, Instance Identifier, ...)

This module does not provide a persistence mechanism of its own. Each class is considered a value object. So the actual persistence of the data is performed by the modules using the respective domain object of the Core Library module.

14.1.1 Coded Attributes

While the Code Services provide a lot of functionality for browsing and authoring of codes, they do not store the code in a semantic context. Normally a medical entity is composed by several properties. Some of these properties describe quantities while others contain semantic information. Typically those properties that encode semantic are represented by codes and are called *Coded Attributes*. Several tasks are necessary to determine the semantic and attach it to a domain object like a diagnosis. Starting with the retrieval of the code concept modeled in the meta data for a particular attribute, the range of possible codes can be derived. Once a code is determined it must be converted into a storable form. Then the associate domain object is persisted using CRUD services. For all these tasks the module `ehf-core` provides helper classes, persistable representations and convenience methods.

Note: Warning



The Code Services provide low level functionality for accessing coded information. Placing these codes into context with some medical content requires a specialized representation and the CRUD Services for attaching a code to a medical domain object. Therefore a transformation is needed that converts the code information provided by the Code Services to a representation required by a domain object.

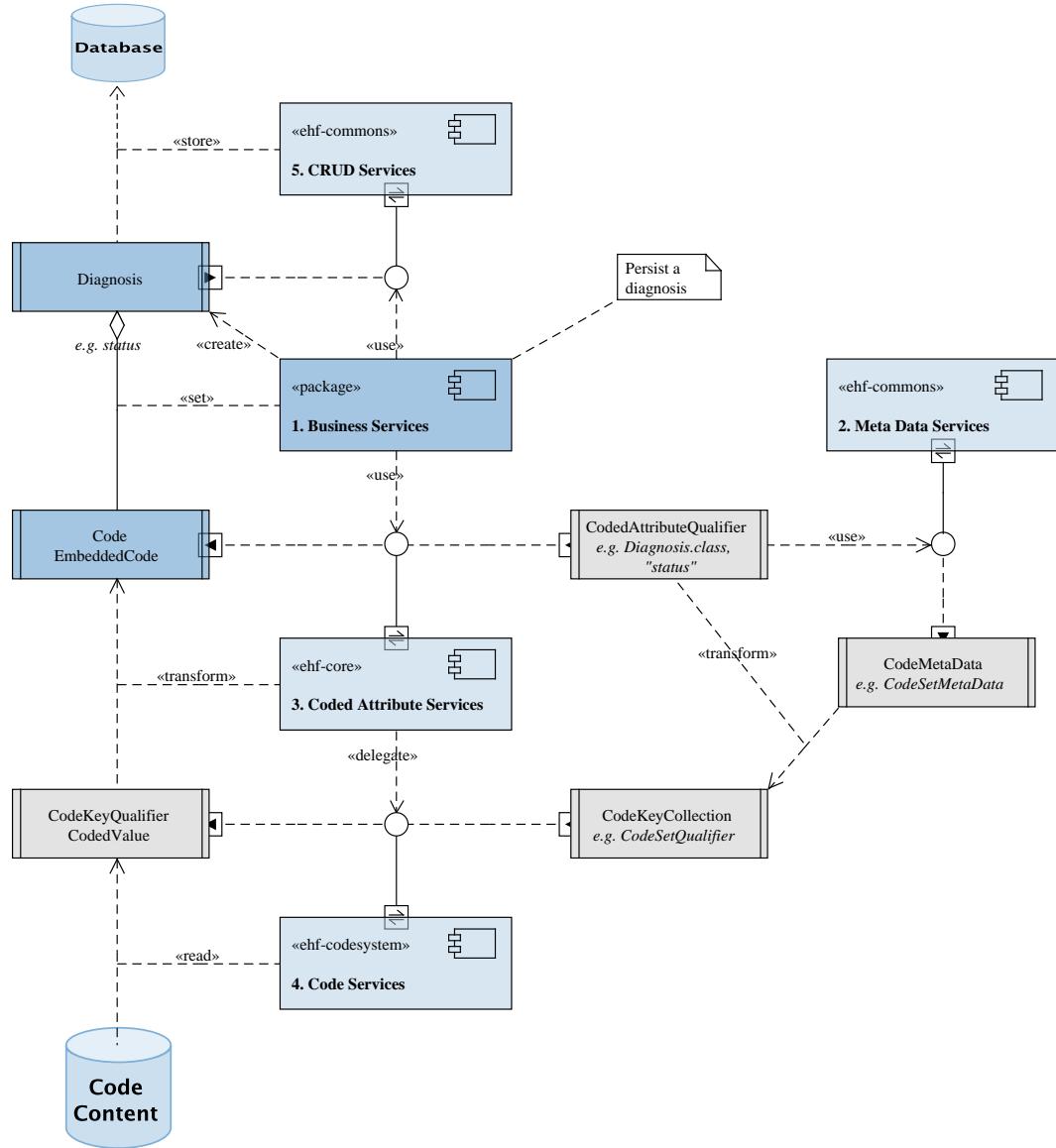
The following sections explain the handling of coded attributes in detail.

14.1.1.1 From Code Services to CRUD Services

This section illustrates the services and objects involved in storing coded information. Normally a procedure like the one described here is conducted in order to encode semantic information.

1. Begin the business process for persisting a diagnosis. Create a diagnosis. Set a code. Persist the diagnosis.
2. Determine a coded attribute of the diagnosis and retrieve its code concept from the meta data.
3. Retrieve the storable form of a code.
4. Query the code content for the desired code concept.
5. Persist the diagnosis using the CRUD Services.

Please note that the Code Services and Meta Data Services are not used directly.

**Figure 38:** CodeAttributeService

14.1.1.2 Persistable Representation of Codes

In order to persist coded information along with a domain object (e.g. `Diagnosis`), these information must be provided in a storable type. The class `com.icw.ehf.core.domain.Code` is such a type and has setter methods for the `key`, `systemId` and `version` data. Once such a `Code` is prepared, it can be set as an attribute of the domain object. When the domain object is created using CRUD services, the coded information is stored in the same table as the encapsulating domain object.

```
// Choose some random code which is element of the code category
"C-DIAGNOSIS-CODE"
Code code = new Code();
code.setKey("ADMDX");
code.setSystemId("2.16.840.1.113883.3.37.1.9.3.1");
code.setVersion("1.0.0");

// Create a new Diagnosis object
```

```

Diagnosis diagnosis = new Diagnosis();

// Set code
diagnosis.setCode(code);

// Persist Diagnosis
Diagnosis persistedDiagnosis = diagnosisService.create(diagnosis);

```

14.1.1.2.1 Restricting Attributes:

For a particular coded attribute not every code makes sense. Therefore it is possible to declare a code concept that defines all possible codes. For example a code category like C-MARITAL-STATUS would contain all codes that are reasonable for the marital attribute of a person class. Such a restriction can be made in the UML domain model of a module. With the help of the generator the code concept is written into the meta data of the domain object respectively. The following tags control this feature:

Tag	Type	Description
concept	CodeSet , CodeSystem , CodeCategory	Determines the kind of code concept.
conceptDescriptor	String	<p>Determines the concrete concept. This conceptDescriptor is a concatenation of the concept's attribute separated by a ' ' character (like the text representation of a CodeQualifier).</p> <ul style="list-style-type: none"> • CodeSet -- name and optional version (e.g. "CodeSetName 1.0.0"). • CodeSystem -- systemId (oid) and optional version (e.g. "SystemId 1.0.0"). • CodeCategory -- name (e.g. "CategoryName").
conceptValidation	Code , SystemId	<p>Determines the kind of validation.</p> <ul style="list-style-type: none"> • Code -- key, systemId and version are validated • SystemId -- systemId and version are validated

Table15. Generator Tags for Code Concepts

See the generator and meta data references for a comprehensive explanation of domain objects, meta data and validation aspects.

14.1.1.2.2 CodedAttributeQualifier :

The range of a coded attribute is given by a code concept that is stored in the meta data of the owning domain object class. With the help of a `CodedAttributeQualifier` such a code concept can be specified without knowing the concrete concept. The qualifier takes either the `CodeMetaData` or the classname of the domain object and the attribute name and is created like other `CodeQualifier`.

```
// Determine the attribute named 'code' of a Diagnosis
CodedAttributeQualifier codedAttribute =
CodedAttributeQualifier.create(Diagnosis.class, "code");

// Get the code concept
CodeKeyCollection codeKeyCollection = codedAttribute.toCodeKeyCollection();
```

The `CodedAttributeQualifier` integrates nicely into the existing `CodeQualifier` hierarchy.

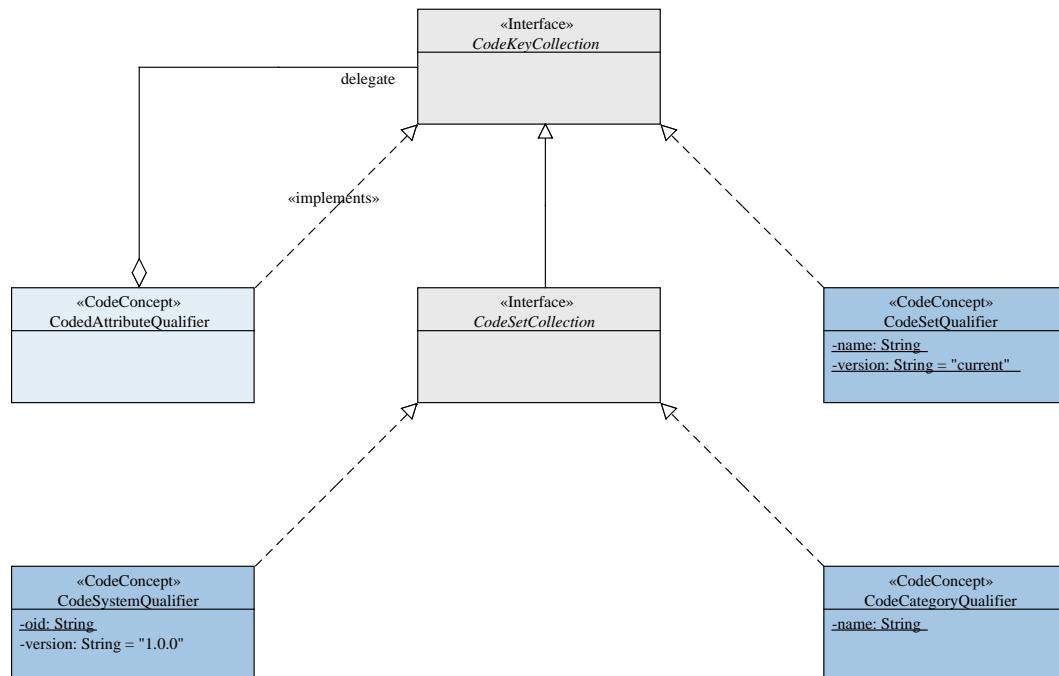


Figure 39: CodeAttributeQualifier Hierarchy

As the `CodedAttributeQualifier` is also of the type `CodeKeyCollection`, it can be used as a parameter for the service respectively.

```
// Get all codes of the concept
CodeContainer<CodeKeyQualifier> container = codeFinderService.find(codedAttribute,
    null);
```

14.1.1.2.3 EmbeddedCode :

Normally an object of the type `com.icw.ehf.core.domain.Code` is created and passed to a domain object. The conversion from a service result like a `CodeKeyQualifier` has to be made by hand. The `EmbeddedCode` object from `ehf-core` helps in converting the results of the Code Services into a storable form.

```
// Get the first code of a result container arbitrarily
CodeKeyQualifier codeKeyQualifier = container.getCodes().get(0);

// Create the persistable code object
EmbeddedCode code = new EmbeddedCode(codeKeyQualifier);

// Create the diagnosis
Diagnosis diagnosis = new Diagnosis();
diagnosis.setCode(code);
```

In the above code snippet the `EmbeddedCodeDto` can be used instead. Both the `EmbeddedCode` and its transfer object can be created from a `CodedValue` the same way while preserving the `value`, `valueType` and `locale` information. That allows the usage of an `EmbeddedCode` as both a display value for a UI and for persistence.

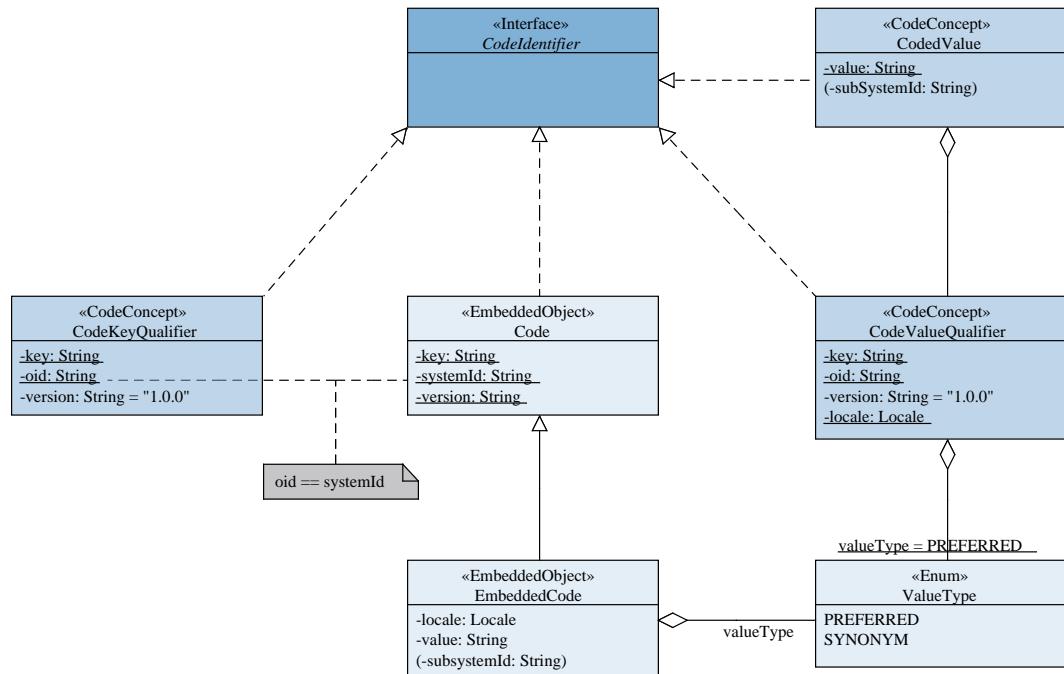


Figure 40: `EmbeddedCode`

14.1.1.3 Wrapping the Code Services

As the copying of coded information from a `CodeIdentifier` to a `EmbeddedCode` is quite simple it is still annoying. Therefore further conversion support can be obtained by using the convenience methods of the following service adapter:

- `CodedAttributeFinderService`
- `CodedAttributeResolverService`
- `CodedAttributeValidatorService`

14.1.1.3.1 CodedAttributeFinderService :

The finder services provide functionality to get the codes for an attribute. Again the result is wrapped in a `CodeContainer` but hosts codes represented by an `EmbeddedCode` or its transfer object equivalent `EmbeddedCodeDto` instead of `CodeKeyQualifier`. In order to perform this task the `CodedAttributeFinderService` requires the `CodeFinderService`.

Service	Parameter	Result	Description
find	CodeKeyCollection<Code> , CodeCriteria<Find>	CodeContainer<EmbeddedCode>	Finds codes within a collection of codes.
findDto	CodeKeyCollection<Code> , CodeCriteria<Find>	CodeContainer<EmbeddedCode>	Finds codes within a collection of codes.

Table16. CodedAttributeFinderService

Using the wrapper is like using the service.

```
// Create the wrapper
CodedAttributeFinderService finderServiceWrapper = new
    CodedAttributeFinderService(codeFinderService);

// Retrieve all codes in the persistable form
CodeContainer<EmbeddedCode> container = finderServiceWrapper.find(codedAttribute);
```

14.1.1.3.2 CodedAttributeResolverService :

This service wrapper removes the burden of converting between CodedValue and EmbeddedCode representations. It requires a reference to a CodeResolverService implementation.

Service	Parameter	Result	Description
resolve	Code , Locale	EmbeddedCode	Returns a EmbeddedCode object containing the value belonging to the given code.
resolveDto	CodeDto , Locale	EmbeddedCodeDto	Returns a EmbeddedCodeDto object containing the value belonging to the given code.
resolve	CodeKeyQualifier , Locale	EmbeddedCode	Returns a EmbeddedCode object containing the value belonging to the given code.
resolveDto	CodeKeyQualifier , Locale	EmbeddedCodeDto	Returns a EmbeddedCodeDto object containing the value belonging to the given code.
resolve	CodeKeyCollection<Code> , CodeCriteria<Resolve>	CodeContainer<EmbeddedCode>	Returns an array of all codes belonging to the given

Service	Parameter	Result	Description
			CodeKeyCollection and localization.
resolveDto	CodeKeyCollection CodeCriteria<Resolve>	CodeContainer<EmbeddedCodeDto>	Returns an array of all codes belonging to the given CodeKeyCollection and localization.
search	CodeKeyCollection CodeCriteria<Search>	CodeContainer<EmbeddedCode>	Query the system for codes and translations which adhere to the given Search criteria.
searchDto	CodeKeyCollection CodeCriteria<Search>	CodeContainer<EmbeddedCodeDto>	Query the system for codes and translations which adhere to the given Search criteria.

Table17. CodedAttributeResolverService**14.1.1.3.3 CodedAttributeValidatorService :**

This class wraps the CodeValidatorService and supports the validation of a Code and its association with the code concept constraint of some coded attribute.

Service	Parameter	Result	Description
validateCode	Code , CodedAttributeQualifier	boolean	Validates if the given Code is an element of the CodedAttributeQualifier .
validateSystemId	Code , CodedAttributeQualifier	boolean	Validates if the systemId of the given Code is an element of the CodedAttributeQualifier .
validateSpecification	Code	boolean	Validates if the properties of the given Code are within specification. That means if a key and a systemId is provided.

Table18. CodedAttributeValidatorService

14.2 Commons Library

The Commons Module is the single most fundamental library module in the eHealth Framework (ehF). It provides the core infrastructure and building blocks to implement the architecture proposed in sections [Conceptual Architecture](#) on page 25 and [Architecture and Life Cycle Realization](#) on page 34. It also offers additional utility and enabling features.

For the most part, other modules, including your own, are supposed to extend or implement the classes and interfaces provided by the commons module. In fact, most, if not all modules shipped with the eHealth Framework do this. For you as a module developer, this task is greatly simplified by the eHF Generator (see [Generator](#) on page 88). Its output builds upon the features provided by the Commons Module and lets you concentrate on your module's business functionality. The information in this chapter will help you understand the technical details going on in the background.

The eHF Commons Module provides functionalities in a large number of distinct areas. They can be roughly divided into four categories.

Architectural Building Blocks

The architectural building blocks implement fundamental parts of the [Standard Service Module Architecture](#) on page 35. A module typically extends the classes provided by these building blocks. This includes DAOs, the object manager, and object assemblers, to name but a few.

Infrastructure

In addition to the fundamental building blocks, a large number of classes support the architecture and play an important role in running an eHF application. While other modules typically do not extend these parts of the Commons Module, they will nonetheless have to use them. This category includes, for example, error and exception handling, Spring framework extensions, and Axis web service customizations.

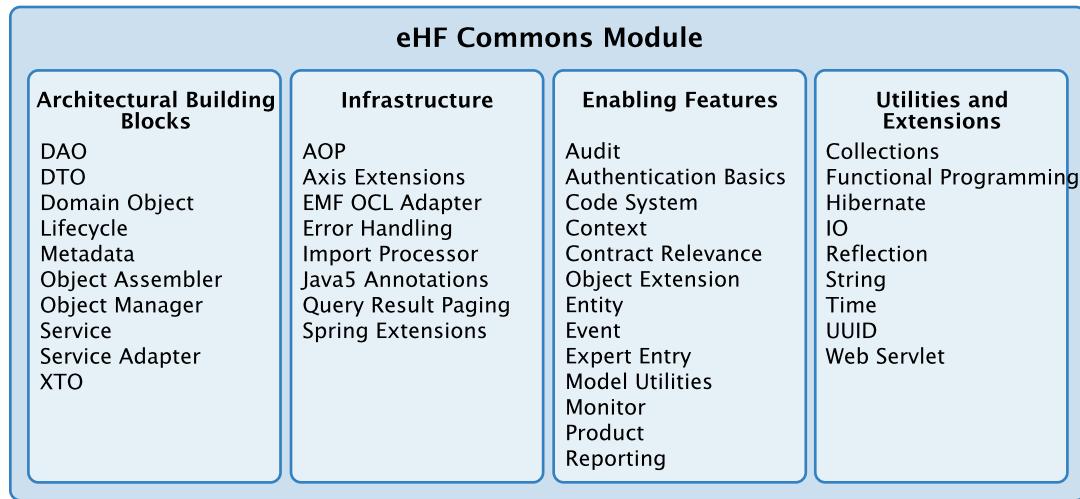
Enabling Features

This part of the eHF Commons Module provides optional features that can be used by other modules. Many of them merely define common semantics or base implementations of a certain functional area. Some modules may implement or extend these, while other modules rely on their presence. The list of enabling features is rather long and includes, among many others, authentication fundamentals, code system management, an audit facility, and the expert entry concept.

Utilities and Extensions

This is a very technical part of the eHF Commons Module. It defines a set of classes that augment shortcomings of other frameworks used within the eHF, and provides utilities to facilitate easy access to commonly used features of these frameworks. Examples include improvements for Java collection and reflection handling, and utilities for the Hibernate persistence engine.

[Figure 41](#) shows an overview of the contents of the eHF Commons Module. The following chapters explore these contents in more detail.

**Figure 41:** Contents of the eHF Commons Module

14.2.1 Architectural Building Blocks

14.2.1.1 Functional Area

...

14.2.1.1.1 Design:

...

14.2.1.1.2 Usage:

...

14.2.2 Infrastructure

14.2.3 Enabling Features

This part of the eHF Commons Module provides optional features that can be used by other modules. Many of them merely define common semantics or base implementations of a certain functional area. Some modules may implement or extend these, while other modules rely on their presence. The list of enabling features is rather long and includes, among many others, authentication fundamentals, code system management, an audit facility, and the expert entry concept.

14.2.3.1 Audit

Data security is among eHF's top priorities. One aspect of data security, which is also very often demanded by governmental or organizational constraints, is that certain data manipulations must be audited. Should data have been tampered with, this audit information can help find out who was responsible for illegal manipulations.

The eHF Commons Audit Module provides an audit facility that captures any CRUD operations on domain objects and publishes appropriate events through a messaging service. It is defined in the `ehf-commons-audit` module.

14.2.3.1.1 Design:

The Audit Module provides listeners for events issued by the Object Manager (`ObjectEvent`). The listeners translate these events into `DomainObjectEvents` (see [Event on page 119](#)) and send a `DefaultMessage` to the eHF Messaging Module. Each message contains one `DomainObjectEvent` as payload. This general mechanism is shown in [Figure 42](#).

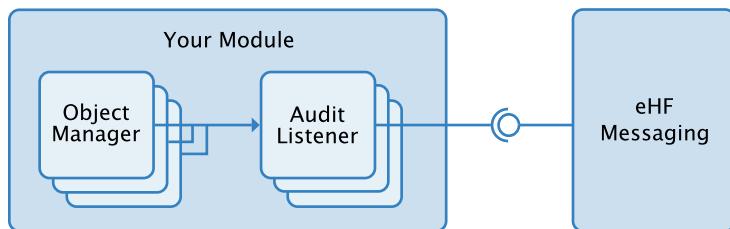


Figure 42: Audit Application Design Overview

The audit listeners are defined in the eHF Commons Audit Module as `Audit{Create|Read|Update|Delete}ObjectEventListener`. They can be registered for their respective events at the eHF Commons' Object Manager. All of them inherit from a common base class `AbstractAuditObjectEventListener`, which provides a filter mechanism based on domain class name. The connection to the messaging service is also defined in this base class.

Registering listeners with the Object Manager(s) and connecting them to the messaging service is configured in your module's custom Spring context. This is a most flexible approach that lets you define various listeners for specific object managers, as well as re-use the same group of listeners for all your module's managers.

14.2.3.1.2 Usage:

Configuration

To use audit listeners in your module, you will have to declare a dependency on the `ehf-commons-audit` module. In your module's custom context (`mymodule-custom-context.xml`), you must declare audit listener beans and inject them into the Object Manager(s) of the domain object(s) you want to audit. It is usually wise to use the parent-bean of all distinct object managers for this registration. Should this cover too many domain objects, you can use the `AbstractAuditObjectEventListener`'s class filter mechanism to get rid of unwanted events. Here is a typical setup code:

```

<!-- Define parent bean for listeners. -->
<!-- Connect to the messaging service. -->
<!-- Exclude some domain class(es) from auditing. -->
<bean id="myModuleAuditEventListener" abstract="true"
      class="com.icw.ehf.commons.audit.listener.AbstractAuditObjectEventListener">
    <property name="messageSender">
      <ref bean="eventMessageSender" />
    </property>
    <property name="excludeClasses" >
      <set>
        <value>com.icw.my.module.domain.MyDomainClass</value>
      </set>
    </property>
  </bean>

<!-- Define update listener list -->
<bean id="myModuleUpdateEventListenerList"
      class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList" >
    <list>
  </list>
  </property>
</bean>
  
```

```

<!-- This is the audit listener. Uses the parent bean defined above. -->
<bean
    class="com.icw.ehf.commons.audit.listener.AuditUpdateObjectEventListener"
    parent="myModuleAuditEventListener" />
<!-- Add any other listeners required in your setup, e.g. meta data -->
<bean class="com.icw...DefaultUpdateObjectEventListener">
    <...>
</bean>
<bean class="..." />
</list>
</property>
</bean>

<!-- Define create listener list. -->
...
<!-- Define delete listener list. -->
...
<!-- Define read listener list. -->
...

<!-- Inject all listeners into the parent object manager bean. -->
<!-- This covers all my module's domain objects, -->
<!-- except for those defined in the class filter. -->
<ehf:inject target-ref="myModuleAbstractObjectManager">
    <ehf:propRef path="createObjectEventListeners"
        ref="myModuleCreateEventListenerList"/>
    <ehf:propRef path="readObjectEventListeners"
        ref="myModuleReadEventListenerList"/>
    <ehf:propRef path="updateObjectEventListeners"
        ref="myModuleUpdateEventListenerList"/>
    <ehf:propRef path="deleteObjectEventListeners"
        ref="myModuleDeleteEventListenerList"/>
</ehf:inject>

```

Note that in the example above, your global system context will have to provide a bean named `eventMessageSender` that implements the eHF Messaging's `MessageSender` interface.

Disabling Auditing

Once audit listeners are registered with your Object Managers, no action will pass un-audited, except for the class filters defined in your configuration. At some points in your application, however, you may want to bypass auditing for certain actions. To this end, the eHF Audit Module provides the `DisableAuditTemplate` class that executes a piece of code without auditing the actions invoked therein. The following code fragment shows how to use this template.

```

public void myMethod(...) {
    new DisableAuditTemplate() {
        public void doWithoutAudit() {
            // perform some actions that are not audited
        }
    }.execute();
}

```

Note that bypassing auditing is a potential security leak and should be considered carefully.

14.2.3.2 Event

The eHF Commons Module provides base classes for an event facility.

14.2.3.2.1 Design:

[Figure 43](#) shows the class hierarchy of the basic event classes. The `Event` interface at the top is merely a marker interface to distinguish eHF-related events from any others. `AbstractEvent` serves as a base class for any concrete event type and provides a

timestamp to indicate when the event was issued. Finally, the `DomainObjectEvent` class captures the event of a CRUD action on a particular domain object.

Figure 43: Event Class Hierarchy

14.2.3.2.2 Usage:

`DomainObjectEvents` are primarily used by the Audit Module (see [Audit](#) on page 117). Other modules, particularly those in the security domain, like authorization and authentication, provide specializations of `AbstractEvent` to capture their event information.

In the unlikely event that you may want to implement your own event mechanism in your module, be sure to derive your events from `AbstractEvent`.

14.2.4 Utilities and Extensions

The Commons Module contains some fundamental utility classes that are used throughout the eHF, although few of them surface to the module developer directly. Most of these are technicalities that simply augment and simplify existing technologies. This chapter gives a short overview of these utilities. Please refer to the JavaDoc documentation for more information.

Collections

The `com.icw.ehf.commons.collections` package defines some utility classes that operate on standard Java collections and arrays, as well as some specialized collection implementations.

Functional Programming

The `com.icw.ehf.commons.functional` package contains abstractions that replicate constructs from the functional programming paradigm. They are used in the collections utility classes.

Hibernate

The `com.icw.ehf.commons.hibernate` package defines utility classes for use with the Hibernate persistence manager, for example a unique ID generator and a composite interceptor.

IO

The `com.icw.ehf.commons.io` package defines specializations of `java.io.FilterReader`.

Reflection

The `com.icw.ehf.commons.lang.reflect` package offers some helpers to work with the Java reflection mechanism.

String

The `com.icw.ehf.commons.text` package contains utility classes that help working with Java Strings.

Time

The `com.icw.ehf.commons.time` package contains helper classes for working with date and time. It makes use of the Joda-Time (<http://joda-time.sourceforge.net/>) library.

UUID

The `com.icw.ehf.commons.uuid` package defines a class that generates unique identifying Strings based on the JDK's built-in UUID mechanism.

Web Servlet

The `com.icw.ehf.commons.web.filter` and `com.icw.ehf.commons.web.servlet` packages contain Java servlet filters and adapters that may be useful in some circumstances.

14.2.5 Contract Relevance**14.2.6 Object Manager**

The Object Manager (OM) is one of the central concepts of the eHealth Framework (eHF). It addresses several concerns regarding the domain object model. The OM is part of the persistence layer (see [Architectural Layers](#) on page 31) but can be considered as a deterministic layer between the service and the persistence layer. It provides enhanced handling of objects which are used between these two layers. The OM thus performs some additional steps and delegates the object service calls to the Data Access Objects (DAO). The OM does not copy Hibernate, the O/R mapping tool which is used by the eHF. Rather the generated service implementations of the eHF expect a certain behavior from the persistence layer, which is ensured by the OM. Moreover this deterministic layer provides the possibility of stronger control and a more flexible administration of the persisting behavior. Wherever Hibernate provides suitable control of administration, it is used, for anything above and beyond Hibernate's capabilities the OM takes over.

By means of the OM complex, object graphs can be managed. It provides methods to supplement the integrity of an object graph before being persisted and deals with the overall consistency of the requested objects. It provides, for example, the possibility to reload any association of an object. With the OM it is also possible to enhance or modify the existing hierarchies without manipulating Hibernate. This will be done with so called *Links* (see section [Consistency of the Object Graph](#) on page 122) which are not realized within the database.

An additional positive side effect of using the OM is that it provides a further abstraction away from the persistence layer and in particular Hibernate's database specific behavior.

14.2.6.1 Overview

As depicted in the class diagram [Figure 44](#) the OM top-level consists of the interface `ObjectManager`, which is provided in the commons module [Commons Library](#) on page 115. The interface `DaoBaseObjectManager` specifies the extended signature for an OM based on a `CrudDao`. The abstract class `AbstractObjectManager` implements this interface and includes generic code which is the main part of the required behavior of an OM. It provides, for example, methods for traversing the object graphs or to endorse, load, persist, update and delete domain objects. Furthermore it provides methods for validating, managing and proofing the consistency of an object graph.

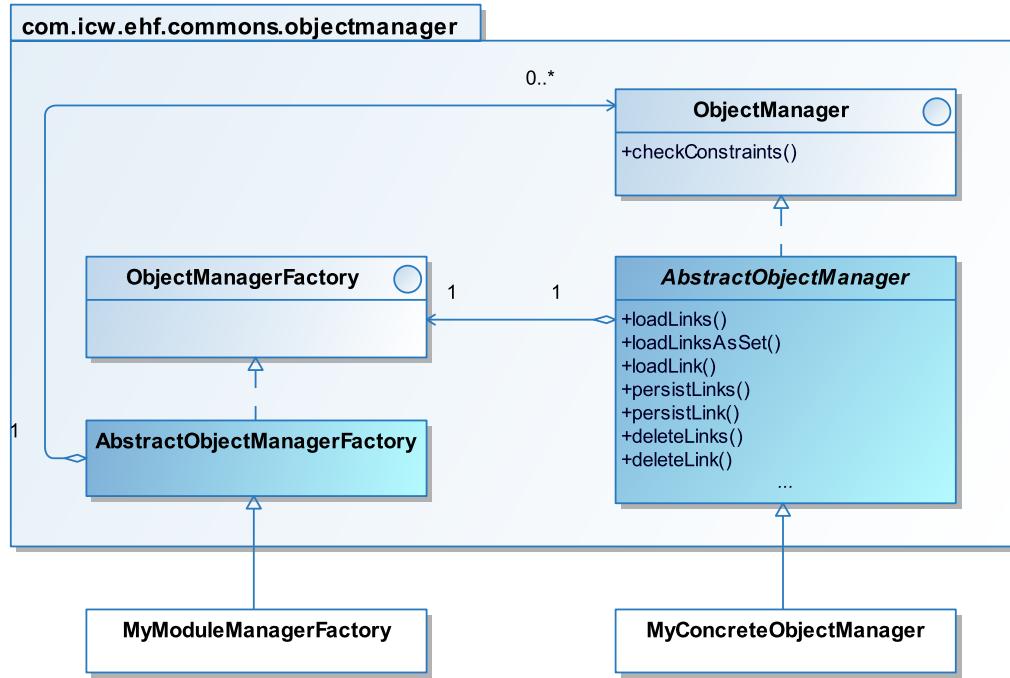


Figure 44: Object Manager Domain Model

Although there is always notion of one OM, there is in fact one concrete `ObjectManager` class for each `DomainObject`, shown as `MyConcreteObjectManager` in the class diagram. These specific implementations of the `ObjectManager` are generated for each `DomainObject` by the eHF Generator (see [Generator](#) on page 88) according to the domain model of a module. They implement the abstract class `AbstractObjectManager` and are focused on a particular domain object class which means that they only add traversal aspects based on the domain object meta information, the so called metadata (see [Metadata](#) on page 135).

The eHealth Framework makes use of the Abstract Factory Pattern to get the concrete instance of the specific `ObjectManager` for a domain object. The `AbstractObjectManager` holds an instance of the factory interface `ObjectManagerFactory` for this. The class `AbstractObjectManagerFactory` implements this interface. This one manages a list of the concrete `ObjectManager` instances according to the different types of domain objects. The specific `ObjectManager` instance will be delivered with the `Class` object of the domain object which has to be passed as a parameter. In spite of the class name it is not an abstract class. However, the eHF Generator creates a specific implementation of the factory class for each module which extends this class to achieve the decoupling from the commons and the specific module.

It is possible to implement your own `ObjectManager`. This is necessary if a module does not use code generation. But the recommended way is to use the generated OM.

As discussed, first of all, the OM is a deterministic layer between the services and the persistence layer. In addition, the OM takes responsibility for several concerns which are common for handling `DomainObjects`, described in detail in the following sections.

14.2.6.2 Consistency of the Object Graph

The traversal aspects of an object and its object graph is based on the domain object meta information (see [Metadata](#) on page 135). But there are some business relationships between domains which are not suitable for modeling in an UML domain model. it models

are too polluted with business constraints they become too complicated to handle. For example, a consultation maybe results in several measures, such as medications, as well as the results of a physiotherapy session. Modeling all these business relations results in a very complex model. Although it has to be possible to demonstrate such relations. For this purpose the `ObjectManager` provides the management of relations with the aid of so called *Links*. These *Links* have to be loaded explicitly and are not implicitly handled by the Object/Relational Mapping tool Hibernate. The OM can load and interpret them. In the context of eHF this data structure is called a complex object graph. This is then handled by the OM.

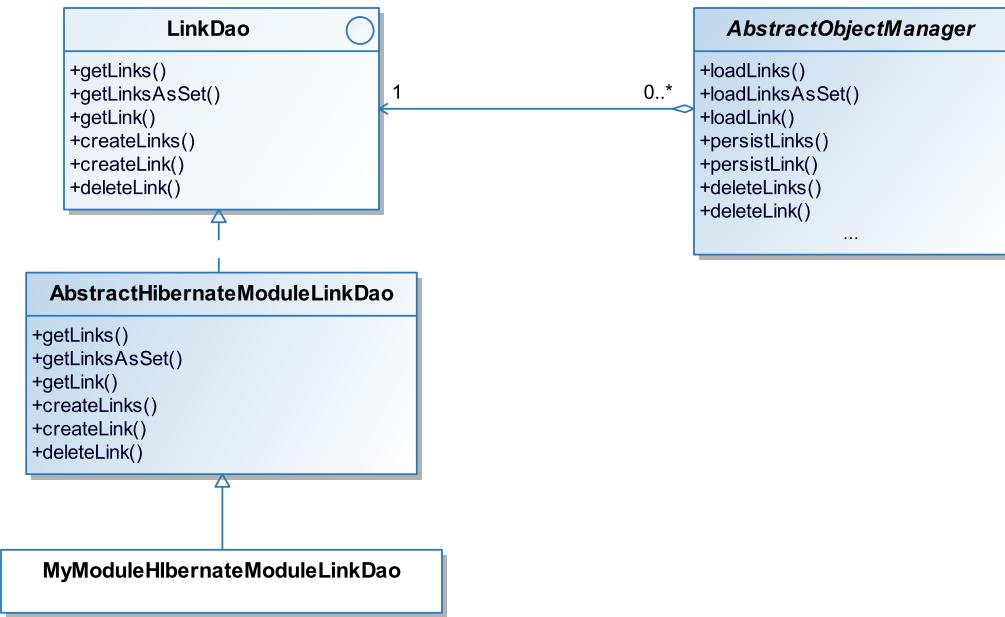


Figure 45: Object Manager Link Model

Figure 45 shows the corresponding class diagram of the *Link Model*. The `AbstractObjectManager` holds a `LinkDao`, the interface for Dao classes which defines methods to provide the functionality for linking DomainObjects and thus to enable the linking of complex object graphs. The `AbstractObjectManager` provides methods to load these *Links*, for example, the method `loadLinks` which yields a list of objects linked with the current domain object. The OM gets these *Links* from the `LinkDao` which also provides some variants of a method signature `getLinks`. An example of one of these variants is shown below.

The following method signature of the `LinkDao` retrieves all object instances of the given class that are linked to the given object:

```
<S extends DomainObject, T extends DomainObject> List<T>
    getLinks(S source, Class<T> targetClass);
```

The class `AbstractHibernateModuleLinkDao` is an initially empty implementation of the interface. This is dependent on the general implementation concept of the eHF to keep the interface to the generator stable. The class is not declared as an abstract method because the required methods should be provided as empty stubs so their implementation is not mandatory. Thus, if you want to use linking, this class has to be implemented. However you do not have to do anything if the linking mechanism is not used.

The eHF Generator creates a concrete implementation of these methods in one specific class for each module, `MyModuleHibernateModuleLinkDao` in this

example. The eHF Generator generates this class within the path of the generated classes `src/main/gen`. This one is provided within the package `com.mycompany.mymodule.dao.hibernate` for Hibernate concerns. In this class the module specific links are implemented by overwriting the corresponding methods of the superclass `AbstractHibernateModuleLinkDao`. It has to be moved from `src/main/gen` to the main source tree `src/main/java` to move it under developer control, to make sure no changes are overwritten by the generator.

The persisting aspect of the self-implemented `Links` is left open. You have to implement them in the concrete `LinkDao` class as well.

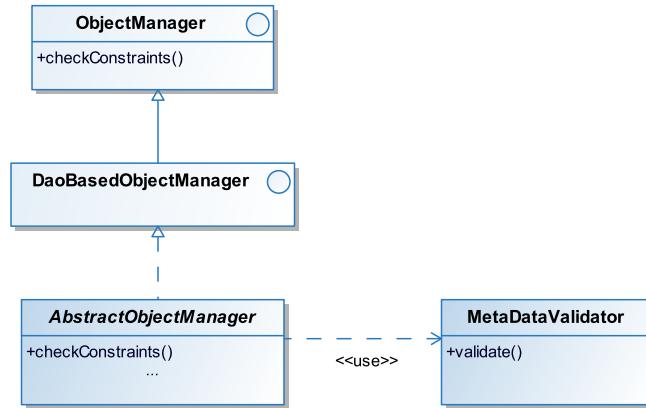
The processing of object service calls is delegated to the specific `ObjectManager`. The tree-traversal within the complex object graph is done by the OM. However, each specific `ObjectManager` just operates on its specific root domain object whose direct dependencies the OM knows. It delegates the operations to be carried out on the child objects to the respective `ObjectManager` of those child objects. Its current state is passed as a `ManagerContext`. This container class holds all the required information to handle the object graph of the current domain object.

Since the OM handles complex object graphs, it is also responsible for ensuring the consistency of these graphs. While loading the domain objects the OM evaluates also the dependency tag of the stereotype `ehf-association`. This tag marks an association as dependency. Whenever an object has to be loaded to a desired completeness, controlled by the context, it is able to invoke its method `loadObject`. The OM then will include links as well as the standard relationships from the database automatically and will fire load events only on the root object (see [Context Sensitive Generation of Events](#) on page 125).

But the OM is also responsible for updating domain objects, if necessary, and thus handles the cascading demarcation. Within the composition boundaries, for example, objects must have the same `scope`; in fact the child objects must have the `scope` of the root object. The propagation of this `scope` is handled by the OM and the `scopes` of the child objects will be adjusted by their concrete OM. Furthermore during each operation on a domain object its specific `ObjectManager` makes sure, that the links to the associated objects are also updated.

14.2.6.3 Validation of the Object Graph

The `ObjectManager` also takes care of the validation management for an object graph. The validation itself is handled as described in [Validation of metadata](#) on page 139 by the `ModuleMetaDataValidator`. However it is triggered by the associated OM of a domain object when a `create` or an `update` operation is performed but before the Data Access Object (DAO) is called.

**Figure 46:** Object Manager Validation

The interface `ObjectManager` provides the method `checkConstraints()` for checking the OCL-based (Object Constraint Language) constraints of a domain object. Also for checking constraint tags of the `ehf-domainobject`, the `ehf-association` or `ehf-attribute` stereotypes such as `maxLength`. Furthermore for carrying out some security relevant checks and for checking Codes (see [Codes, Code Sets and Code Systems](#) on page 14). The `AbstractObjectManager` implements this method. First it examines if the object has to be validated. The information is contained in the given the `ManagerContext` of this operation. This can prevent the execution of the validation operation twice on an object. If the validation has to be done the OM executes the validation method on the `ModuleMetaDataValidator`.

14.2.6.4 Context Sensitive Generation of Events

Another concern which is covered by the OM is context sensitive event generation. [Figure 47](#) shows, for example, the class diagram of events and listeners for a *READ* operation. The interface `ObjectEventListener` defines the signatures for handling `ObjectEvents`; `preEvent` and `postEvent`. The method `preEvent` is executed before the actual action is performed. The `postEvent` method is invoked after the actual action triggering the event has been processed. Using this mechanism enables the implementation of additional strategies.

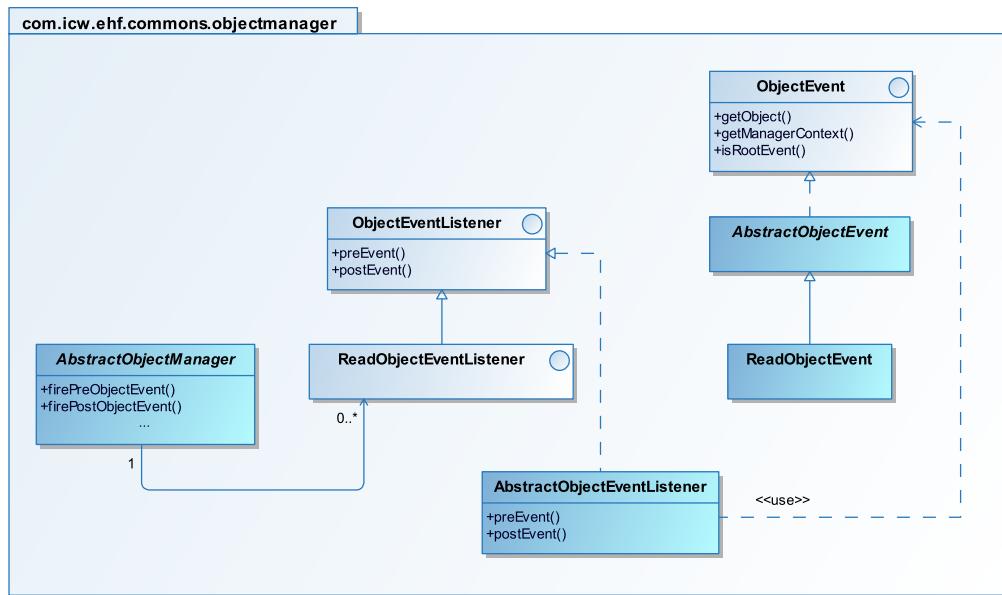


Figure 47: Object Manager Events and Listener

The interface `ReadObjectEventListener` is the marker for *READ* operation listeners. There are also interfaces for *CREATE*, *UPDATE* and *DELETE* operations. A special implementation of this interface can only handle a special type of event. The interface `ObjectEvent` is common to all object events in the OM sense. The abstract class `AbstractObjectEvent` implements this interface and its methods for getting the domain object, the manager context and for proving that the current domain object is the root object of the object graph. For each event there is a special implementation of its abstract class. For the *READ* operation, for example, this is the class `ReadObjectEvent`.

The `AbstractObjectManager` holds one collection for each type of registered listeners. While manufacturing objects it fires events if the current domain object is the root object of the object graph. Before the actual action the `preEvent` and afterwards the `postEvent` will be fired. The methods `firePreObjectEvent` and `firePostObjectEvent` will be executed for this. These events can then be handled by registered listeners.

For example, there is a special implementation in the audit module (see [Audit](#) on page 126) with its class `AuditReadObjectEventListener` which implements the interface `ReadObjectEventListener` and extends the abstract class `AbstractObjectEventListener`. Thus, the generated event might be, that a particular object has just been read from the database - so a *READ* event should be fired. This event triggers the creation of an audit entry that documents that this read has been performed by whom and when (see [Audit](#) on page 126). If an object from the database should be deleted, it has to be read by Hibernate from the database too, but in that case since it is only a technical read, no audit entry shall be written and therefore, no *READ* event should be created. Hibernate can not differentiate between these two read operations. The OM takes care of such context information regarding event generation.

Another specific is the event creation for particular types of domain objects; for example the registration for *READ* events on `Medication`. Hibernate does not support this possibility. Hibernate listeners are always informed about any *READ* event and are not able to be type-sensitive.

The registration of listeners is managed through. Spring uses its methods `addCreateObjectEventListener`, `addReadObjectEventListener`, `addUpdateObjectEventListener` and `addDeleteObjectEventListener` for the

registration of specific listeners. The listener which should be registered has to be declared as a bean within the configuration file *mymodule-custom-context.xml*. The following code example shows the registration of an audit event listener for the *READ* operation to the module *<modulename>*:

```
<!-- ====== -->
<!-- ObjectManagerEventListeners configuration -->
<!-- ====== -->
<bean id="myModuleAuditEventListener"
      abstract="true"
      class="com.icw.ehf.commons.audit.listener.
           AbstractAuditObjectEventListener">

    <property name="messageSender">
      <ref bean="eventMessageSender" />
    </property>
</bean>

<bean id="myModuleReadEventListenerList"
      class="org.springframework.beans.factory.config.ListFactoryBean">

    <property name="sourceList">
      <list>
        <!-- Audit -->
        <bean
          class="com.icw.ehf.commons.audit.listener.
               AuditReadObjectEventListener"
          parent="myModuleAuditEventListener">
        </bean>
      </list>
    </property>
</bean>

<ehf:inject target-ref="myModuleAbstractObjectManager">
  <ehf:propRef path="readObjectEventListeners"
    ref="myModuleReadEventListenerList" />
</ehf:inject>
```

14.2.7 Exception and Error Handling

This chapter is concerned with the exception handling of the eHealth Framework (eHF). It is based on an exception infrastructure (see [Infrastructure](#) on page 129) within the commons module. The exception handling process and the exception infrastructure will be explained in the further sections.

14.2.7.1 Overview

The eHF Exception Handling concept provides a highly flexible and extensible way to process and manipulate exceptions at the Web Service level.

The eHF defines interfaces for exception handling and processing. It provides implementations of those interfaces, for logging or wrapping of third-party or JDK exceptions in eHF exceptions for example. It also provides support for error mapping (see [Error Codes](#) on page 128), which is necessary, that is, to prevent information leakage and to enable the localization (I10n) of error messages.

With the aid of the eHF exception mapping, internal exceptions and thus implementation details are hidden. For example, no stack traces of the exceptions are passed within the error messages. Instead tickets are generated. A ticket is a unique identifier for a certain exception occurrence. By means of the ticket number, it is possible to relate an error message sent to the client to the corresponding exception in a LOG file. The LOG file contains the complete stack trace. This, for example, can then be used in handling support requests.

The eHF uses predominantly unchecked exceptions. In other words, you don't have to catch any of the thrown exceptions.

14.2.7.2 Error Codes

Internal exceptions and third-party exceptions have to be mapped to error codes. An error code is a unique identifier for a certain exception type or a certain error situation. The mapping is performed by a dedicated error handler that is Spring configured (see [Configuration](#) on page 133).

Exception Type - Exceptions and other error situations are always instances of a certain Java type, which is called exception type in the context of eHF's exception handling mechanism. Typically, but not necessarily, exception types are subclasses of `java.lang.Exception`.

Exception Classifier - Some exception types can optionally be classified, thereby defining logical subtypes of exception types. An exception classifier is a simple String. Components in the exception handling processing chain may infer information from it, for example which kind of error message to produce.

Error Codes – An error code represents the occurrence of a certain error condition. It is inferred from the exception type and optionally the exception classifier. The concepts of the code system module (see [Code System](#) on page 133) is used for defining error codes. This means that error codes have to be encoded using an oid/key tuple (a version is not required, since the code systems have internal character). The eHF will support a general code system for error codes (codesystem-repository) and potentially additional code systems for each module (module or codesystem-repository; support modularization). Products based on eHF can choose their own strategy on how to provide additional code systems for error codes. The recommendation is to use a code system per unit.

Error Code Substructure – For encoding certain information the structuring of the error code is used. The localization is covered by the OID. In particular the OID could include the module as a last digit. For example, the OID looks like 2.16.840.1.113883.3.37.5.9.1. The key can also include a substructure. The eHF currently uses the following keys `<Category>.<Number>`. The first item categorizes the exception (system error, usage error, ...), while the second is a simple sequence number.

Error Codes on Web Service Level – For reporting error codes on the level of web services a QName is required. The QName contains a namespace that is configurable (framework paradigm; consistent with namespace strategy on webservices). In the eHF scope an error code will look as follows: `http://ehf.icw.icw/errorcode/<oid>/<key>`.

14.2.7.3 Error Handling Process

The following image depicts the overall exception handling process which is only provided on Web Service level:

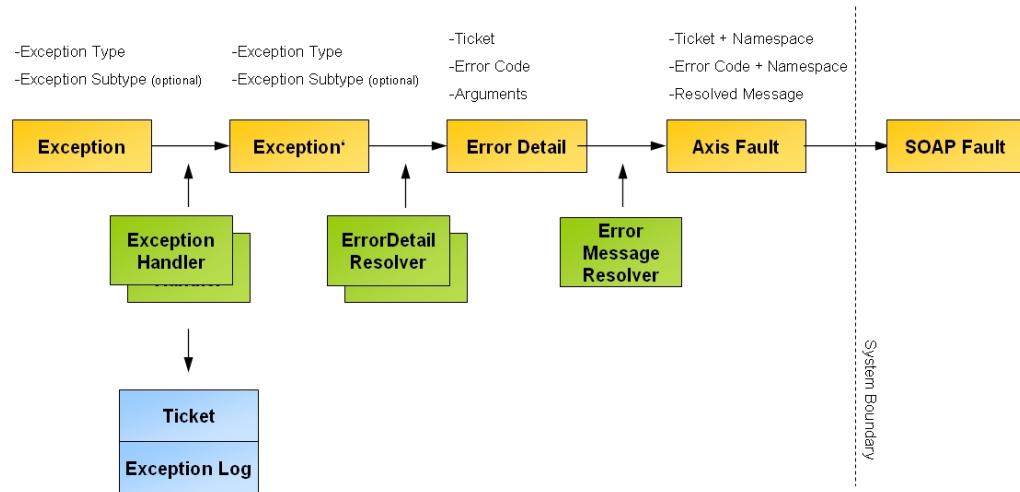


Figure 48: Web Service Exception Handling Process

The Web Service consumer sends a request. While processing this request an error occurs and the corresponding class throws an exception of a specific exception type (or optionally of an exception subtype). The `LogExceptionHandler`, a specific implementation of the `ExceptionHandler` creates a ticket and passes the exception to the `ErrorDetailResolver`. It is responsible for the mapping of exceptions to error codes. The resolver creates an `ErrorDetail` and passes it to the `ErrorMessageResolver`. This `ErrorDetail` is the context of a certain exception occurrence and contains the ticket, the error code and the arguments. The `ErrorMessageResolver` looks up the corresponding error message. Then Axis creates an `AxisFault`. This `AxisFault` contains the ticket with namespace, the error code with namespace and the resolved error message. An error message is the human-readable text associated to a code. The axis fault will be converted to a soap fault by Axis if it leaves the system boundary. The consumer receives this soap fault as the result of its original request.

14.2.7.4 Infrastructure

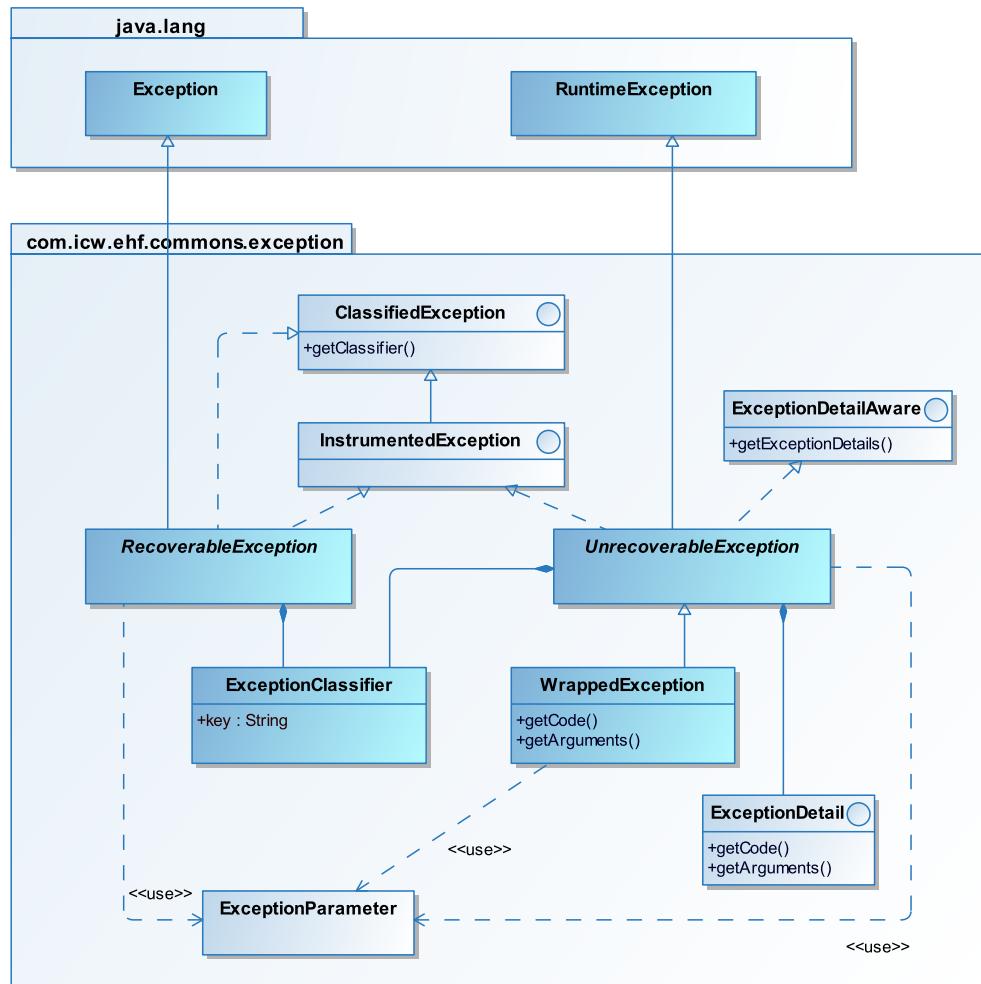
The following sections describe the infrastructure of the participating components of the exception handling process.

Exceptions Infrastructure

The commons module contains the exception handling infrastructure. It provides classes that allow for the handling of exceptions of an eHF based system in a consistent manner. It also provides custom exception classes to encapsulate third party and JDK exceptions.

The package `com.icw.ehf.commons.exception` provides the eHF exception class hierarchy. The exception hierarchy reflects the architectural layers and components of an eHF based module or system. The hierarchy should encourage reuse of exception types or enable sub classing to define more specific exceptions.

Please note that eHF exceptions are supposed to be free of error codes. The most important element to an exception is its type and - if specified - its classifier. Mapping error codes is enabled through use of this information.

**Figure 49:** eHF Exceptions Infrastructure

On the highest level of the eHF exception class hierarchy stands the interface `ClassifiedException` for exceptions that have been classified without defining a specific exception type. An `ExceptionClassifier` is used here. Exceptions which additionally provide an array of Object as arguments implement the sub-interface `InstrumentedException`. The subjacent level differ in the two abstract classes `RecoverableException` and `UnrecoverableException`. This differentiation is equivalent to the both types of exceptions in Java, the checked `Exception` and the unchecked `RuntimeException`. The names are supposed to add semantics to the semantic-free Exceptions.

Some errors may have multiple causes that they may wish to expose. They can do so by implementing the `ExceptionDetailAware` interface and provide separate `ExceptionDetail` instances for each distinct cause. Their `code` property typically acts as an exception classifier.

Furthermore there exists the class `ExceptionParameter`; a convenience class for specifying parameters to an exception. This enables that exceptions only require one constructor instead of serving many kinds of combinations or parameters. Implementing subclasses enables the construction of a parameter class hierarchy aligned to an appropriate exception hierarchy.

The `WrappedException` extends the `UnrecoverableException`. It will be used for exceptions, which a client cannot be expected to handle.

Error Detail and Error Code

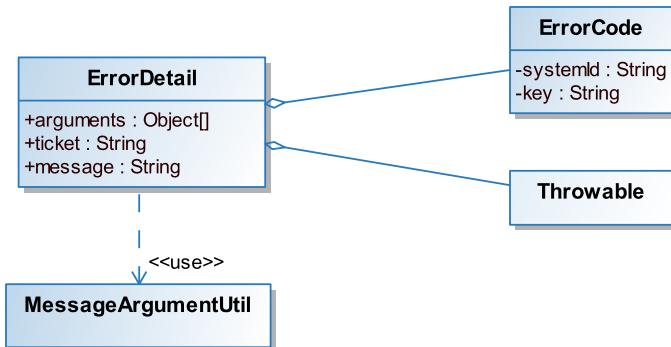


Figure 50: Exception Handler Class Hierarchy

The error detail and potential subclasses of this class carry all relevant information concerning an error condition. It contains the ticket number, the arguments and the error message. It also holds an instance of the concerned `ErrorCode` and the original cause of the error by its `Throwable`. For handling arguments in messages it uses the utility class `MessageArgumentUtil`.

Error details can be nested if the original `Throwable` instance contains multiple exception details (by way of `ExceptionDetailAware`). Only one level of nesting is supported.

Exception Handler

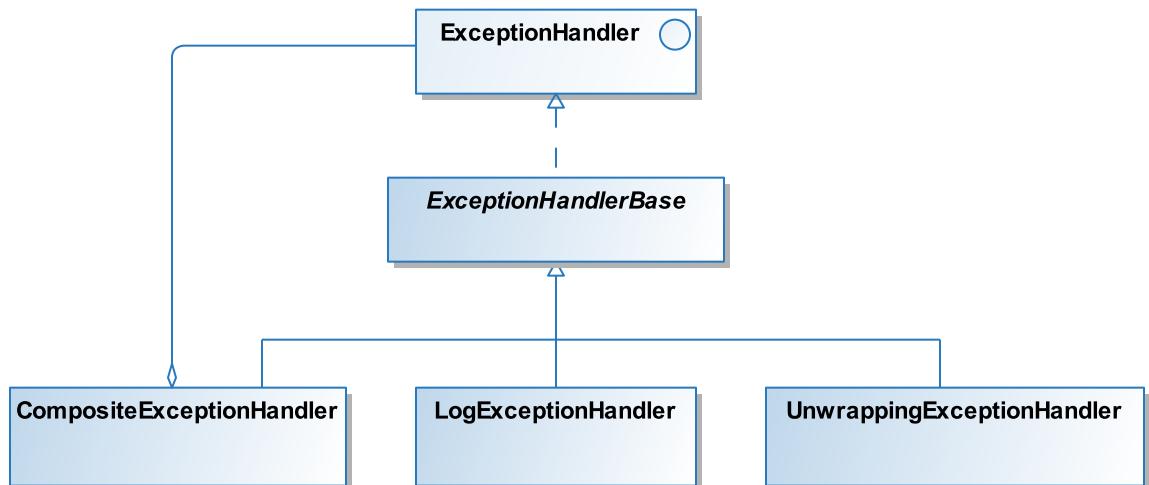


Figure 51: Exception Handler Class Hierarchy

The exception handler is the entry point of the exception handling process. For processing the Composite Pattern is used. The exception handling process contains a `CompositeExceptionHandler` which is a composite implementation of the `ExceptionHandler` interface. The implementation extends the abstract class

`ExceptionHandlerBase` and uses the Chain of Responsibility pattern to handle exceptions using an internal list of specific `ExceptionHandlers`, for example, the `LogExceptionHandler`. The composite handler passes the exception to all of its internal, specific handlers. Each of these processes the request and returns the result.

Error Detail Resolver

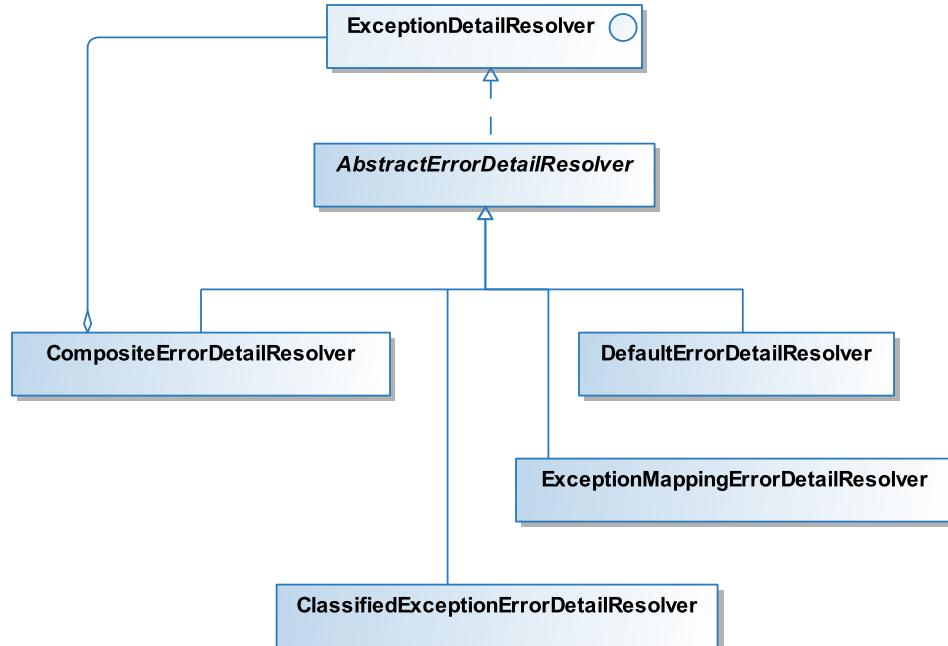


Figure 52: Error Detail Resolver Class Hierarchy

The `CompositeErrorDetailResolver` is hooked in the processing chain of the exception handling. It is the composite implementation of the `ErrorDetailResolver` interface. The composite resolver delegates the `ErrorDetail` invocation to an ordered list of its specific configured `ErrorDetailResolvers`. The first resolved `ErrorDetail` is returned.

The `CompositeErrorDetailResolver` handles instances of `ExceptionDetailAware` exception types and resolves their owned `ExceptionDetails` into separate (nested) `ErrorDetails`.

Instead of the `CompositeErrorDetailResolver` you can also use the `PlatformCompositeErrorDetailResolver` that has a hard-coded fallback delegation to the `DefaultErrorDetailResolver`. This can be useful to avoid Spring setup problems.

Error Message Resolver

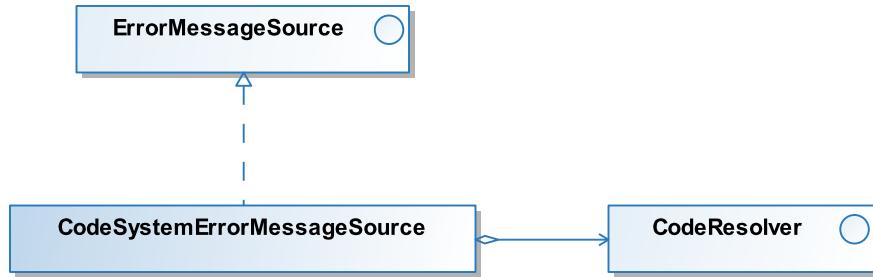


Figure 53: Error Message Resolver Class Hierarchy

The `CodeSystemErrorMessageSource` implements the `ErrorMessageSource` interface for resolving translated error messages by error code. It holds an instance of an implementation of `CodeResolver` for this.

Client Side - SOAP Fault

The code example below shows the resulting SAOP fault as it appears at the client's side.

```

<soapenv:Envelope>
  <soapenv:Body>
    <soapenv:Fault>

      <faultcode>
        ns1:2.16.840.1.113883.3.37.2.9.5.2/VAL-001003
      </faultcode>

      <faultstring>
        Please choose a unique user identifier.
        The identifier (USERNAME) is already assigned.
      </faultstring>

      <detail>
        <ns2:ticket>
          8e781075-b849-4a39-ac41-57ff0871c7c3
        </ns2:ticket>
      </detail>

    </soapenv:Fault>
  </soapenv:Body>
</soapenv:Envelope>
  
```

The `<faultcode>` contains the namespace `ns1`, the error code and the exception classifier. Within the `<faultstring>` the error message is provided and the ticket number with its namespace `ns2` is found in the `<detail>`.

14.2.7.5 Configuration

The components of the exception handling process are Spring-managed beans. Therefore they are configured within the corresponding XML configuration files.

14.2.7.5.1 Module Specific Configuration:

Error codes and error detail resolvers used only by a single module are contributed by this module and therefore will be configured within the module's `src/main/resources/META-INF/ehf-<module>-runtime-context.xml`. The `ErrorDetailResolver` has to be exported via the module context.

Error Codes and Error Detail Resolvers

The following example shows an error configuration within the user management module (see [User Management](#) on page 210). eHF provides an error XML namespace that allows for a very simple definition of both error codes and error detail resolvers.

```
<error:resolver id="classifiedExceptionErrorDetailResolver"
    systemId="2.16.840.1.113883.3.37.2.9.5.2" type="classified">
    <error:mapping errorCode="VAL-001001" key="com.icw.ehf.usermgnt.exception
        .UserSecretSyntaxInvalidException#PASSWORD"/>
    <error:mapping errorCode="VAL-001002" key="com.icw.ehf.usermgnt.exception
        .UserSecretSyntaxInvalidException#/"/>
</error:resolver>

<error:resolver id="exceptionMappingErrorDetailResolver"
    systemId="2.16.840.1.113883.3.37.2.9.5.2">
    <error:mapping errorCode="VAL-001003" key="com.icw.ehf.usermgnt.exception
        .UserIdentityUniquenessViolationException"/>
    <error:mapping errorCode="VAL-001004" key="com.icw.ehf.usermgnt.exception
        .UserNameSyntaxInvalidException"/>
    <error:mapping errorCode="VAL-001005"
        key="com.icw.ehf.usermgnt.exception.InvalidGenderException"/>
    <error:mapping errorCode="VAL-001006"
        key="com.icw.ehf.usermgnt.exception.PersonNotFoundException"/>
    <error:mapping errorCode="VAL-001007"
        key="com.icw.ehf.usermgnt.exception.UserNotFoundException"/>
    <error:mapping errorCode="VAL-001008"
        key="com.icw.ehf.usermgnt.exception.InvalidRoleException"/>
</error:resolver>
```

The `error:resolver` tag defines an error detail resolver, either of type `ExceptionMappingErrorDetailResolver` or of type `ClassifiedExceptionErrorDetailResolver`, the latter by specifying the `type="classified"` property. The mapping from exception types (with optional exception classifier after the hash # character) to error codes is done with nested `error:mapping` tags. This defines the error codes on-the-fly. The code system id given in the `error:resolver` tag applies to all error codes. You can also specify a `systemId` property in the `error:mapping` tag.

It is very typical to configure more than one `ErrorDetailResolver`, as seen above. You will then have to configure a `CompositeErrorDetailResolver` for the module which will ultimately be hooked into the processing chain of exception handling.

```
<bean id="usermgntErrorDetailResolver"
    class="com.icw.ehf.commons.errorhandling.resolver.
CompositeErrorDetailResolver">

    <property name="errorCodeResolvers">
        <list>
            <ref bean="exceptionMappingErrorDetailResolver"/>
            <ref bean="classifiedExceptionErrorDetailResolver"/>
        </list>
    </property>
</bean>
```

14.2.7.5.2 Central Configuration:

Central Configuration of the exception handling components is done at the `src/config/ehf-errorhandling-context.xml` within the assembly.

Error Detail Resolver

The module resolver will be included in the central error handling configuration file with the fragment `@@@error.resolver.fragment@@@` to fit it into the exception handling processing chain. As the following example shows it will be defined within the web service composite resolver and thus a nested processing is achieved.

```

<bean id="webserviceErrorDetailResolver"
      class="com.icw.ehf.commons.errorhandling.resolver.
      CompositeErrorDetailResolver">

    <property name="errorCodeResolvers">
      <list>
        @@@error.resolver.fragment@@@
        <ref bean="exceptionMappingErrorDetailResolver" />
        <ref bean="defaultErrorDetailResolver" />
      </list>
    </property>

    <property name="throwableArgumentExtractor"
              ref="webserviceThrowableArgumentExtractor" />
</bean>
```

The additional error detail resolvers which are referenced in this context file are defined and configured in the same way as the error detail resolver for the modules but now within the central configuration file.

Exception Handler

This section shows an example for the assembly configuration of the exception handler. In this case the Composite Pattern is used as described above.

```

<bean id="webserviceExceptionHandler"
      class="com.icw.ehf.commons.errorhandling.handler.CompositeExceptionHandler">

    <property name="exceptionHandlers">
      <list>
        <bean
          class="com.icw.ehf.commons.errorhandling.handler.
          UnwrappingExceptionHandler" />
        <bean
          class="com.icw.ehf.commons.errorhandling.handler.
          LogExceptionHandler" />
      </list>
    </property>
</bean>
```

The CompositeExceptionHandler will be configured with a list of specific exception handler, the UnwrappingExceptionHandler and the LogExceptionHandler in this case, only by defining a bean with the concerned class as value for them.

Compatibility

There are applications based on older versions of the eHF which do not implement this exception handling mechanism. For reasons of backward compatibility therefore it is possible to disable this mechanism. Exceptions then will be completely converted into soap faults which, for example, contains the complete stack trace.

The example below shows an example of the configuration. In case of exceptionCompatible = false the mechanism is enabled. This is the default configuration.

```

<bean id="webserviceErrorHandler"
      class="com.icw.ehf.commons.axis.servlet.AxisErrorHandler">

    <property name="errorProcessor" ref="webserviceErrorProcessor" />
    <property name="errorDetailNamespace"
              value="http://ehf.icw.com/errordetail" />
    <property name="errorCodeNamespace"
              value="http://ehf.icw.com/errorcode" />

    <!-- this property enables backward compatibility -->
    <property name="exceptionCompatible" value="false" />
</bean>
```

14.2.8 Metadata

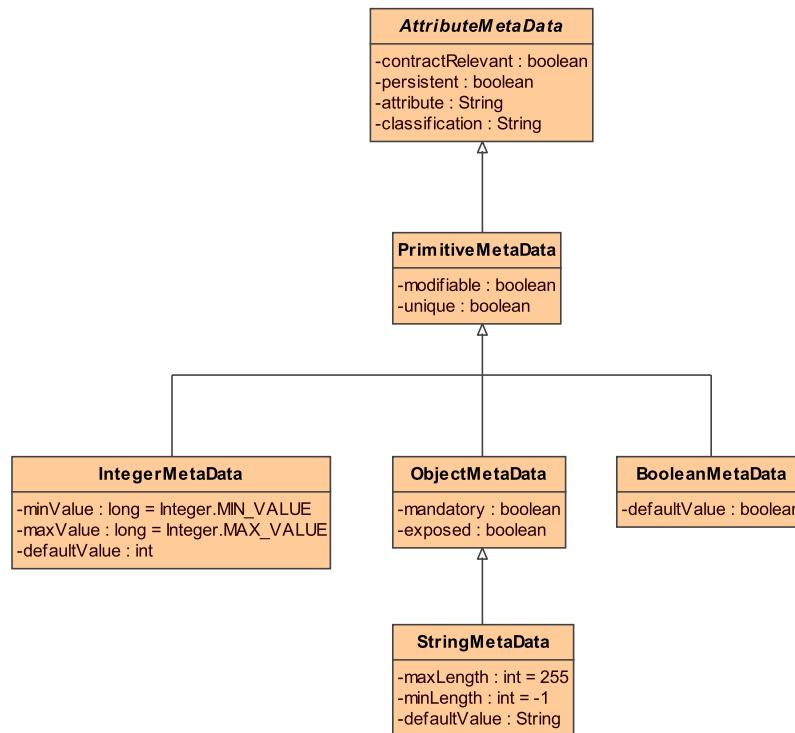
The purpose of metadata in the eHF context is to provide additional information on the domain objects. This metadata information may be used to validate the user data for consistency and to control the access on the data. First the metadata information must be created using the eHF Generator. The eHF Generator provides tags which correspond with the single metadata information. You might write the metadata definition by hand but this is not recommended. Either way the metadata of a domain object is defined in an xml structure and is named accordingly. For example the metadata for a Diagnosis is called `diagnosis.xml`. Typically the XML file is stored in the same package as the domain object. Once there is metadata present for a domain object it may be evaluated. An evaluation process will access the metadata through a provider interface which abstracts from the xml format and the location of storage. The provider serves the metadata structured into a class hierarchy. The metadata might be enhanced by a custom strategy or extended to fit your own requirements. Here ends the short overview on the metadata. The following sections provide more details on the different aspects of metadata usage.

14.2.8.1 Hierarchically organization of metadata

The actual data a user may store is structured into domain objects. Additional information on structure elements such as restriction on the range of an integer or a textual value, flags for classification of the content, business rules, etc. are maintained as information called metadata. The metadata information for every instance of a domain object of the type `BaseDomainObject` is assembled together into an associated metadata object graph, which is rooted in the class `ClassMetaData`. Information on properties `AttributeMetaData` and business rules `OclConstraints` can be retrieved from the `ClassMetaData` object. Overall the `ClassMetaData` contains the following information:

Metadata	Description
<code>contractRelevant</code>	Marks a domain object as contract relevant. Such an object may neither be updated nor deleted by a user, since it represents essential information on a contract. Which of the properties are relevant is indicated through the contract relevant flag on the respective attribute. See below for <code>AttributeMetaData</code> ;
<code>modifiable</code>	This flag indicates whether the CRUD service update is allowed. This flag is set to <code>true</code> by default;
<code>creatable</code>	This flag indicates whether the CRUD service create is allowed. This flag is set to <code>true</code> by default;
<code>deletable</code>	This flag indicates whether the CRUD service delete is allowed. This flag is set to <code>true</code> by default;
<code>permissionClassifier</code>	The permission classifiers are used to define permission profiles for this domain object. For more see the chapter on access control;
<code>oclConstraints</code>	Describes business restrictions on class-level verbalized through OCL expressions;
<code>attributeMetaData</code>	Contains metadata on the attributes of the domain object.

The `AttributeMetaData` hierarchy and its information are arranged as follows:



Metadata	Description
attribute	The name of the attribute on which the metadata applies;
contractRelevant	A user needs special permissions to change an attribute marked as contract relevant. An attribute may only be contract relevant if the domain object itself is contract relevant;
persistent	In case the associated domain object can be persisted, this attribute is stored in the database;

The abstract class `PrimitiveMetaData` and their derivations encapsulates metadata on basic Java types:

Metadata	Description
modifiable	If an attribute is modifiable its content may be altered by a client;
unique	This information is normally used only on database level and mirrors the constraint onto the table in which the object is stored;
mandatory	Marks this attribute as mandatory. The domain object cannot be persisted if no value for this attribute is set;
minValue	The minimum value of this integer attribute. The default is the lowest possible integer value -2147483648;
maxValue	The maximum value of this integer attribute. The default is the highest possible integer value 2147483647;
defaultValue	Specifies the default value for the attribute;

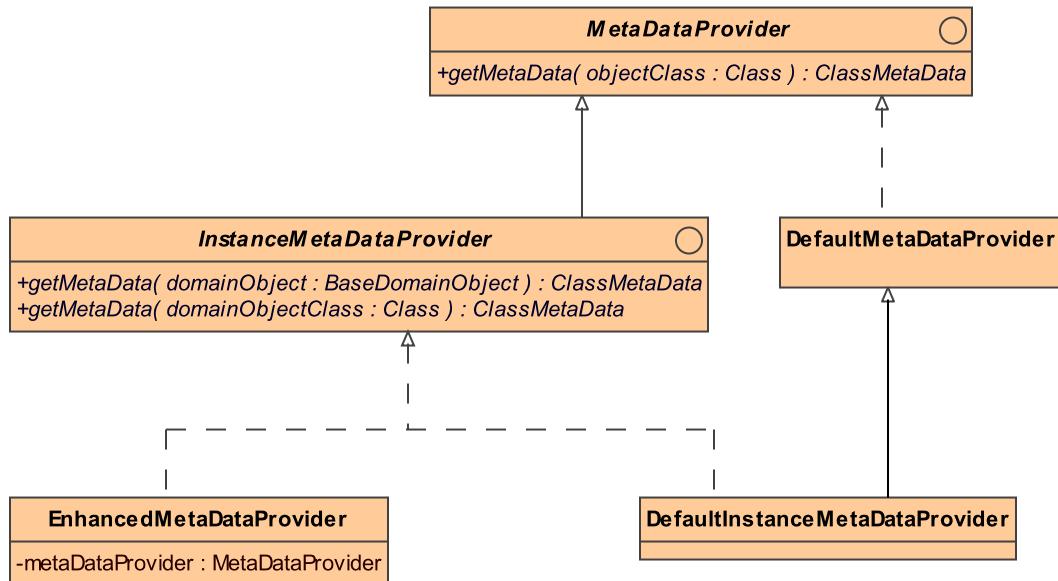
Metadata	Description
minLength	Specifies the minimum length for a string. The value -1 is the default and means that no minimum length can be provided (empty string or null allowed);
maxLength	Specifies the maximum length for a string attribute. The default is derived from the common database default value which is 255. The max value should not exceed 1024.

Currently all the classes which hold metadata are located in the package `com.icw.ehf.commons.metadata.transfer` in the `ehf-commons` library module.

14.2.8.2 Retrieving metadata

In order to evaluate the metadata of a domain object, it must be retrieved. For that the information must be located and then the metadata must be extracted. Currently the XStream library is used for serialization of the metadata into the xml structure and vice versa. Typically the XML file is located in the package of the domain object class. eHF offers means to abstract from the localization and the extraction of the metadata definition. The `ClassMetaData` can be obtained by using a `MetaDataProvider` implementation or by calling the `loadMetaData` method provided by the `ModuleService` implementation of the business module to which the domain object belongs. In fact the `loadMetaData` service delegates its task to the `MetaDataProvider` of the respective module.

A `MetaDataProvider` provides the `ClassMetaData` that is registered for a given class of a `BaseDomainObject`. An `InstanceMetaDataProvider` extends a `MetaDataProvider` with the functionality to extract the `ClassMetaData` for a given instance of a `BaseDomainObject`. This allows the dynamic modification of the metadata based on a particular instance like the `EnhancedMetaDataProvider` does. See the next section for more on the enrichment of metadata.



The `DefaultDataProvider` is a file-based implementation of the `MetaDataProvider` that accesses the xml representation of the metadata for the class of a domain object and deserializes the metadata into a `ClassMetaData` object. Similarly the `DefaultInstanceDataProvider`, which implements the `InstanceMetaDataProvider` interface, also creates a `ClassMetaData` object but

additionally supports the extraction of metadata when given a domain object instance. The eHF Generator configures one `MetaDataProvider` for a module.

14.2.8.3 Validation of metadata

While some of the metadata constraints are applied directly on the database level (e.g. mandatory properties are treated as NOT NULL columns) others cannot be implemented with standard SQL statements easily or not at all. While a database table might restrict the maximum length of a VARCHAR column there is basically no mechanism to enforce a minimum length other than zero. Therefore eHF follows the policy that the authoritative validation of persistent data happens at runtime before storing them in the DB. The metadata validation of a `BaseDomainObject` is triggered by an associated `ObjectManager` when a create or an update operation is performed but before the `DataAccessObject` is called. See the previous chapter for more on the `ObjectManager`. The `ModuleMetaDataValidator` performs the validation process on domain objects by iterating through the attributes. A `ValidationContext` helps to keep track of the iteration. For each attribute the `MetaDataProvider` interface is used for retrieving the corresponding metadata. From the configured validation strategies a matching `MetaDataValidator` is retrieved and is asked to check the constraint against the actual values. When a mismatch is detected, a message is stored in the `MetaDataValidatorContext`. After the domain object is evaluated against its `AttributeMetaData`, the OCL constraints are validated. The `ModuleMetaDataValidator` adds a constraint violation message when a domain object violates the OCL expression defined for a given domain object. By default the eHF generation process configures the `ModuleMetaDataValidator` of a module to use a `MetaDataValidator` implementation for the validation strategies of `String` and `Integer` attributes of a `BaseDomainObject`.

In order to support validation procedure of 3rd party components (e.g. the JSF mechanism only validates the changes to property) eHF provides helper classes for accessing the metadata validation. A `ValidationParameter` wraps information that is required to implement validation from a GUI. The `ValidationService` interface defines a method called `validate` which takes the domain objects within a `ValidationParameter` for validation. This interface is implemented by the service adapter of the module to which the domain objects belong. The returned `ValidationResult` instance provides details based on each domain object passed by the `ValidationParameter`. According to the kind of the detail, the results are structured attribute-centric in an `AttributeValidationResult` or object-centric in an `ObjectValidationResult`.

Please note that some of the metadata information represents constraints, which can not be validated because they are rather flags to support or trigger certain operations like access control on domain objects, contract relevance tests or CRUD service availability. See the respective sections for more information on their meanings.

14.2.8.4 Enhancement of existing metadata

A `MetaDataEnhancer` can modify existing `ClassMetaData`. Its behavior can be dependent on the properties or state of a given object or class. In order to apply such a `MetaDataEnhancer`, it must be added to the list of enhancement strategies of an `EnhancedMetaDataProvider`. The `EnhancedMetaDataProvider` is a provider implementation that decorates the default provider by applying one or more `MetaDataEnhancer` that modify the metadata that has been read by a default provider. An implementation is the `CustomMetaDataEnhancer` which extends the default metadata by additional customized metadata information. For example this could be additional OCL constraints.

The custom metadata is expected to be defined in the same xml format used by the `DefaultMetaDataProvider`. The only difference is that the names of the metadata file use a predefined additional prefix which must be specified at construction time of

the `CustomMetaDataEnhancer`. For example the custom metadata definition file for a Diagnosis should be named `myapp-diagnosis.xml` and must be stored in the same package as the metadata file generated by eHF. The 'myapp' prefix must be passed to the constructor of the `CustomMetaDataEnhancer` so that it can locate the custom metadata definition in the classpath of the 'myapp' application.

Please note that the customization of the metadata must never weaken the constraints predefined by eHF. In other words domain objects which validates successfully against the customized constraints must do so as well against the framework constraints. The following table lists some examples for such custom constraint restrictions:

Level	eHF constraint	custom constraint
Class	Association cardinality: 1:M	Association Cardinality 1:5
Attribute	Property is optional (nullable)	Property is mandatory (never null)
Attribute	String max length: 255	String max length: 100

Currently the eHF Generator defines as the main `MetaDataProvider` for each module an `EnhancedMetaDataProvider`. The bean definition of the provider contains an empty list of `MetaDataEnhancers`, which are then expected to be set from the module's custom context (the Spring configuration). There you might add an additional `MetaDataEnhancer` like the `CustomMetaDataEnhancer` that has the responsibility to merge in the custom constraints.

Custom constraints – because they are defined outside the model – are not applied at all at the database level. The user should be aware of the fact that the DB cannot be trusted to enforce any of the custom integrity constraints that have been added.

An example for metadata enhancement is the LifeSensor application which uses the mechanism described above for adding OCL constraints on domain objects of the `ehf-record` module.

14.2.8.5 Custom metadata

You might introduce completely new metadata classes or validation strategies to eHF. The framework already provides additional metadata and validation strategies in the following domains:

- Metadata information and validation of codes,
- validation of annotated codes and
- verification of ISO dates.

See the next paragraphs for more.

In order to handle Code attributes in domain objects a `CodeMetaData` instance stores additional information:

Metadata	Description
<code>codeSystemCategory</code>	The code system category restricts the attribute to a group of code systems;
<code>codeSystemId</code>	The code system id also known as OID specifies a particular code system on a code attribute;
<code>codeSubsystemId</code>	Determines that the code must be from a specific subsystem;
<code>codeSystemName</code>	Restricts the code to a code set. Previously a code set was addressed by a code system name identifier. For compatibility reasons this data is not yet renamed;

Metadata	Description
codeSystemVersion	Defines the code system version. Per default the code system version is 1.0.0;
codeSystemCategoryValidation	Indicates whether the codeSystemCategory has to be validated against the code repository;
codeSystemValidation	Indicates whether the codeSystem has to be validated against the code repository;
codeValidation	Indicates whether the full Code has to be validated against the code repository;
defaultValue	Specifies the default value for the attribute.

This metadata class is derived from the `ObjectMetaData` definition. Therefore the validation of constraints like the mandatory flag is already performed by eHF. In order to validate the code-specific metadata a `CodeMetaDataValidator` must be set as an additional validation strategy to the `MetaDataValidator`. See the chapter on codes to grasp when a Code is sufficiently specified and will validate successfully.

An `AnnotatedCode` extends a `Code` by a free text field named `value`. eHF provides an `AnnotatedCodeMetaDataValidator` which extends the `Code` validation by taking the `value` attribute into consideration. The `AnnotatedCode` is also considered valid when no code-specific attributes of the `AnnotatedCode` are set but the `value` field is set.

The core module of eHF provides a class to handle date information. The date is stored in a string attribute according to the ISO date specification. A string validation on a `Date` instance wouldn't make much sense. Therefore eHF provides a specialized `DateMetaDataValidator` which validates if the string is formatted according to the specification.

The configuration of additional validation strategies should be done in the modules custom context using the post injection capabilities of eHF. Inject a set of additional validation strategies to the `additionalValidators` property of the `MetaDataValidator` which belongs to the module.

14.2.9 Validation via Annotations

Input validation is the key to “Defensive Programming”. Checking input of a method and to reject it in a controlled fashion will minimize the risk to run into unpredictable errors caused by malformed input data.

A “robust” application verifies input data at designated points in the control flow. As a “Rule of Thumb” an application should validate all input provided by external components. What an external component defines depends on the application, it could be:

- a jar file
- a different package
- a different class

In the realm of eHF, input validation using annotations is enforced on the Service Layer (see [Service Layer](#) on page 37), because this layer implements the business logic of a module and is therefore the natural place to validate input data. eHF services are annotated in the following way:

```
@Validate
public interface MedicationService extends AbstractMedicationService<Medication> {
    public DiscontinuedInformation addDiscontinuedInformation(
        Medication parent, DiscontinuedInformation child);

    public void removeDiscontinuedInformation(Medication parent);
}
```

```

    public SupplyInformation addSupplyInfo(Medication parent,
        SupplyInformation child);
    public void removeSupplyInfo(Medication parent);
}

```

The `@Validate` annotation will trigger validation for all parameters of all methods declared in this interface. The implementation retrieves the `java.lang.Class` type of the relevant parameter as a key into a validator lookup table (see “eHF Howto - Validate a Service”).

By default, during input validation only `String` parameters are validated. These can either be the `String` attributes of domain objects, individual `Strings`, or the elements of `String` arrays or parameterized Collections of `Strings` (i.e. `List<String>`). You can expand the input validation to include other parameter types by writing your own Validator and adding this to the input validation configuration (see “eHF Howto - Validate a Service” for details).

14.2.9.1 Metadata Validation

Validation of metadata is enforced either by the database or by the `ModuleMetaDataValidator`, triggered by the `ObjectManager` (see [Validation of metadata](#) on page 139). Both alternatives enforce validation relatively late in the layered architecture. On the other hand, it is desirable to enforce the validation step early, to reject invalid input. This will enhance the overall performance and reduce the memory footprint of eHF.

To meet the mentioned requirements from above, eHF is shipped with the following validators:

- `com.icw.ehf.commons.domain.BaseDomainObject`
- `java.util.Collection<BaseDomainObject>`

Validation of a `com.icw.ehf.commons.domain.BaseDomainObject` is delegated to the appropriate `ModuleMetaDataValidator`.

14.2.9.2 Implementing a custom Validator

Implementing a custom validator is straight forward:

1. Implement `com.icw.ehf.commons.metadata.validator.Validator`
2. Register the implemented validator
3. Annotate the class or parameter

After these steps, the validator is ready to use, that's all. For detailed information how to write and configure a validator please refer to “eHF Howto - Validate a Service” (Sec. “Writing Your Own Validator”).

14.2.10 Result Paging

14.2.11 Service Strategies

The eHealth Framework offers the possibility to add custom service strategies via AOP. These custom strategies must extend the interface `com.icw.ehf.commons.aop.ServiceStrategy`. For example, in the module eHF Record the following two custom service strategies are used:

- `com.icw.ehf.record.medical.aop.DataEntererStrategy` : This strategy initializes the data enterer for data uploaded by a user and a device.
- `com.icw.ehf.commons.aop.MetaDataServiceStrategy` : ServiceStrategy implementation for security checks on the available metadata. For example if an

attribute of a domain object is marked as *immutable*, then its content cannot be modified.

The custom service strategies are handled by `com.icw.ehf.commons.aop.ServiceStrategyInterceptor`. This interceptor is invoked before a service call. The strategies are invoked one after the other.

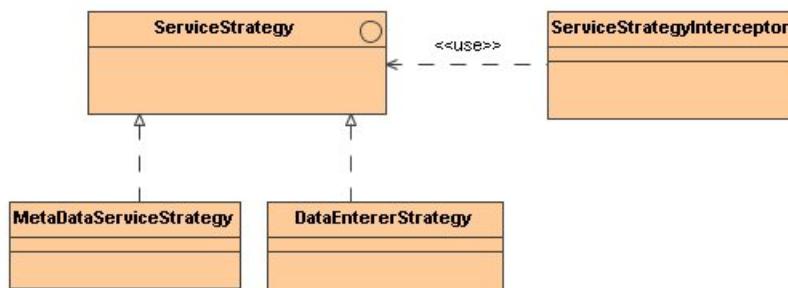


Figure 54: Service Strategy

The XML snippet below shows the Spring configuration. The beans `dataEntererStrategy` and `metaDataStrategy` are injected into `recordMedicalServiceStrategyInterceptor` which again is injected to the secure CRUD services, e.g. `recordMedicalDiagnosisSecureService`.

```

<bean id="dataEntererStrategy"
      class="com.icw.ehf.record.medical.aop.DataEntererStrategy">
  ...
</bean>

<bean id="metaDataStrategy"
      class="com.icw.ehf.commons.aop.MetaDataServiceStrategy">
  ...
</bean>

<bean id="recordServiceStrategies"
      class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
    <list>
      <ref bean="metaDataStrategy" />
      <ref bean="dataEntererStrategy" />
    </list>
  </property>
</bean>

<ehf:inject target-ref="recordMedicalServiceStrategyInterceptor"
            path="serviceStrategies" ref="recordServiceStrategies" />

<bean id="recordMedicalDiagnosisServiceImpl"
      parent="recordMedicalAbstractCrudService"
      class="com.icw.ehf.record.medical.service.DiagnosisServiceImpl"/>
<bean id="recordMedicalDiagnosisSecureService"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="recordMedicalDiagnosisServiceImpl" />
  <property name="proxyInterfaces"
    value="com.icw.ehf.record.medical.service.DiagnosisService" />
  <property name="interceptorNames">
    <list>
      <value>recordMedicalDiagnosisServiceSecurityAspect</value>
      <value>recordMedicalServiceStrategyInterceptor</value>
    </list>
  </property>
</bean>
  
```

14.3 Commons Expert Entries Library

For medical data the functional role of the data enterer is regarded as being of relevance: If such an object was created by e.g. a physician it must not be updated by anyone else than the original creator ("expert rule").

A source indicator is associated with a data enterer and indicates how well he can judge the correctness of data he just entered. For example, a doctor might be regarded as a "Medical Expert" which means that we expect high quality data from him. However, a non-professional may misinterpret some medical information and therefore produce input of lower quality.

Each product integrating ehf-record can define how a qualification role (QR) for each user is defined. Based on the qualification role of the user who entered or changed data this data entry is interpreted as an expert entry (e.g. QR = professional) or not (e.g. QR <> professional).

14.4 Commons Camel

IPF supports various deployment options ranging from lightweight embedded or stand-alone deployments. None of these options are suitable in context of running IPF inside a web application in where route entry points can be directly exposed. This is due to the fact that IPF is based on Apache Camel which does not provide any component for such a scenario.

Although Camel is not designed towards this requirement, there is nothing against embedding Camel inside a web application. However, since Camel is an integration platform the isolated usage of Camel does not make much sense. Instead, a deployment solution is needed that embeds IPF inside of eHF in a way that allows the direct exposure of routes within an application. Further, the solution should be in alignment with the Camel approach for providing connectivity to other systems and protocols. At the current time of writing, the only solution offered by Camel itself into this direction consists of launching an embedded Jetty server inside of a web application. This is accomplished by the Camel Jetty-component. But launching Jetty inside of a servlet container is not recommended. Neither does it make sense to run two servlet containers in parallel or nested nor is Jetty a supported infrastructure component for eHF with the latest security patches.

The eHF provides two custom Camel components in order to deploy IPF in eHF in the above presented fashion. Both components provide HTTP based endpoints for consuming HTTP requests. The next section will first give an overview of the major concepts in Camel with focus on the Camel component. After that, the custom Camel components provided in eHF are introduced.

14.4.1 Camel Concepts

The four major concepts in Camel are components, endpoints, processors and the domain-specific language (DSL). Components provide connectivity to other systems. From components, endpoints are created for sending and receiving messages. Endpoints are uniquely identified by their URIs. Processors are used to route and transform messages between endpoints. To wire endpoints and processors together, Camel provides a DSL for Java.

Camel components not only provide connectivity to other systems but also provide extension points in Camel to add connectivity to other systems. In context of a web application it is obvious that the connectivity consists of exposing HTTP endpoints from

within the servlet container. This is due to the fact that the HTTP protocol is the application protocol of the web.

The basic idea of the custom components is that a servlet dispatches requests into routes. Further, you can distinguish between service-oriented endpoints to expose procedural services and resource-oriented endpoints to expose data-centric services. The next two sections introduce these two custom Camel components.

14.4.2 Service Component

Process-oriented services are characterized by a large number of routing and processing steps that can also involve queries to other external systems in order to gather information. This processing logic is represented by a route where the routing and mediation engine Apache Camel is well suited. In order to expose these routes from inside a web application eHF provides a custom Camel service component. The eHF service component binds the consumer endpoint part of a route to a generic servlet that dispatches HTTP requests based on the relative path of the URL. Since the service component is in alignment with the component principle of Camel, its usage follows the URI principle:

```
from("service:<relative-path>").process(...)
```

This route definition has to be defined in a `SpringRouteBuilder` class that is configured in your module as a basic Spring bean. Then, this bean can be registered to the global Camel context by using the contribution mechanism. While deploying a eHF-based application with this module in a web container, Camel will automatically register all contributed routes to the Camel context. Beyond that, the component prefix `service` of the URI initiate Camel to publish the defined route through a HTTP endpoint at the URL `https://localhost/<product>/service/<relative-path>`. Any HTTP client can send messages to this endpoint by issuing a HTTP request and pass additional context information within the `ServletRequest`. The `ServletRequest` and the `ServletResponse` are wrapped in a `ServletExchange` that is an implementation class of a the Camel `Exchange` interface. The `Exchange` interface provides an abstraction for an exchange of messages, that is, a request message and its corresponding reply or exception message. In Camel terminology, the request, reply and exception messages are called in, out and fault messages. The `ServletExchange` is made available to the subsequent processor of the route and holds the context information of the request and the response. Consider a simple example:

```
from("service:hello/world").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getOut().setBody("Bye");
    }
});
```

With this route definition inside of an eHF-based application Camel publishes a HTTP endpoint at `https://localhost/<product>/service/hello/world`. A HTTP client issuing a HTTP request against this URL will receive as result the content "Bye" as byte stream in the `ServletResponse`. This is achieved by the subsequent processor that operates on the `exchange` parameter in order to write to the underlying servlet output stream.

14.4.3 Resource Component

While the eHF Camel service component is the first choice when dealing with process-oriented services (e.g. workflows) some applications offers their functionality on system resources. In brief, a resource in a system is everything that can be identified. Applications that focus on resources are designed towards a resource-oriented architecture. Resource-oriented architectures are mainly found in domains with a data-centric focus where

resources are persisted in a database and manipulated by CRUD operations, as in the case of eHF based applications. In context of the web Roy Fielding coined the term REST that is an acronym for Representational State Transfer. REST defines a set of architectural principal by which web services can be designed in a resource-oriented architecture style.

The eHF Camel resource component focuses on this architecture style. It binds a HTTP endpoint to a restlet infrastructure based on the URI template according to the Java specification for Restful web services (JSR-311). The definition of a resource consumer endpoint is similar to the one in the service component, except that this time the URI starts with the prefix resource. More, the relative path is interpreted as an URI template:

```
from("resource:<uri-template>?restletmethod=<http-method>").process(...)
```

The prefix resource of the URI triggers Camel to publish the above route through a HTTP endpoint at the URL `https://localhost/<product>/resource/<uri-template>`. In the same manner as with the service component, any HTTP client can send messages to this endpoint by issuing a HTTP request. But in contrast to the endpoint exposed by the service component the resource endpoint only consumes requests if the specified `<restletmethod>` corresponds with the HTTP method made by the producer. The supported HTTP methods are the standard HTTP methods GET, POST, PUT, DELETE and HEAD. These methods provide a uniform interface for manipulating system resources and enables the HTTP protocol to be used not only as a transport protocol but also as an application protocol. An additional feature offered by the URI template mechanism is the ability to pass in parameters into the URL, that are then available as context information within the route. Consider a simple example for the resource component:

```
from("resource:hello/world/{world}?restletMethod=get").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        exchange.getOut().setBody("Bye " + exchange.getIn().getHeader("world"));
    }
});
```

With this route definition Camel exposes a HTTP endpoint at `https://localhost/eHF/resource/hello/world/{world}` where the parameter world (the term in curly brackets) can be passed in. A HTTP client issuing a HTTP request to this URL with the parameter "Earth" for example will receive as result the content "Bye Earth" as byte stream in the `ServletResponse`. This is achieved by the subsequent processor that operates on the exchange parameter and extracts from it the header fields of the in message.

14.5 Commons Security Library

This chapter gives an overview about the elements of the commons security library module.

14.5.1 Cross Site Request Forgery Guard

Motivation

Most web-based applications are exposed to attacks of the Cross-Site Request Forgery (CSRF) category. In literature, CSRF is often referred to as XSRF or Session Riding as well. Because of the hard pronunciation of CSRF, "SeaSurf" is as well an alternative expression widely seen. For a short introduction of CSRF attack scenarios and possible countermeasures, we refer to http://www.isecpartners.com/documents/XSRF_Paper.pdf.

In short, every web-based application with a predictable invocation structure is vulnerable against CSRF attacks. A predictable invocation structure exists if a request to perform a certain action always follows the same pattern. For Java EE applications, this means that HTTP requests executed with the same intention (e.g. change email address) always have the same structure. In the email address example, the only variant is the new email address

itself. This enables attackers to rebuild such requests and present them to victims. One common technique is mail spoofing, where the attacker sends an email with a prepared link to execute an intended request to the victim. If the victim clicks on the link, the action is performed on the victim's behalf. If applications are vulnerable against Cross-Site Scripting, such a predefined request could as well be injected inside a running application.

A commonly accepted countermeasure is to use cryptographic tokens to prove the action formulator knows a session- and action-specific secret. In Java EE applications, we already have the concept of session IDs, a secret shared by client and server. But nevertheless, a CSRF attack is executed by the victim. During execution, the attacker is passive. The victim only performs actions he is allowed to. Audit traces for that reason do not help in later forensic analysis because the attacker does not leave any traces on the target system. So, having an additional shared secret between client and server prevents us from being CSRF vulnerable. Only if the action formulator as well provides the expected secret, the action is performed. Since the attacker does not know the action-specific secret, he can not prepare the formulation of the forged request.

Overview

eHF provides a sophisticated solution against CSRF attacks applicable for Java EE applications. It implements the above mentioned countermeasure which as well is recommended by iSEC Partners. The implementation comes as a servlet filter implementation with various configuration and extension possibilities.

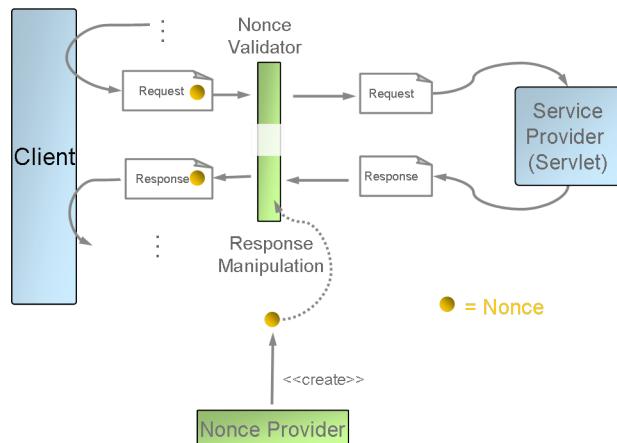


Figure 55: eHF CSRF Guard Overview

The CSRF Guard filter could be introduced in any Java EE application. It only has to be registered in the application's servlet filter chain. So, security concerns are completely isolated from business concerns. The CSRF Guard implementation so follows the 'Security Last' principle.

The previously mentioned secret is called nonce within this chapter, where nonce stands for "Number used once". It's a randomly generated and cryptographically strong number that is only used once during an application's runtime. With the CSRF Guard filter in place, each incoming request gets intercepted and a so-called NonceValidator checks whether the expected nonce is part of the request. If yes, the request is passed as intended. If not, all registered so-called NonceValidationListeners are notified to react on the possible attack. On the way back from the server to the client, the response as well gets intercepted. Here, the new nonce has to be produced and incorporated in the response in order to be present in the subsequent request. The NonceProvider is responsible for creating strong nonces. NonceIntroducer implementations thereby introduce produced nonces into HTTP responses. Following this pattern, the client always gets nonces back from the system which need to be provided in the subsequent request in order to pass validation. Under

the assumption, the underlying channel is secured (e.g. SSL), nonces are only known by client and server. An attacker can not prepare a preformulated request because the nonce needed for action execution is not known to him.

Components

NonceValidator:

A NonceValidator implements the interface `com.icw.ehf.commons.security.web.csrf.NonceValidator` and has the responsibility of analyzing incoming requests about the existence of a nonce. If no nonce is found, an event is triggered on which NonceValidationListeners can react upon. eHF Commons Security provides the `RequestParamNonceValidator` which validates that a nonce is provided as a request parameter for incoming request.

NonceValidationListener:

If an expected nonce is not detected by a NonceValidator implementation, all registered NonceValidationListener are notified. Implementations of the `com.icw.ehf.commons.security.web.csrf.NonceValidationListener` describe how applications react on possible CSRF attacks. Currently, eHF provides the following implementations:

- `LogNonceValidationListenerImpl` : This listener produces a log entry about a missing nonce. This listener should NOT be used in production since it only logs potential attacks instead of reacting with an appropriate countermeasure.
- `AuditNonceValidationListenerImpl` : This listener produces an audit event in case a validation failed. As a result, an audit entry about a potential attack is persisted in the audit database. Again, using only this listener is NOT an appropriate countermeasure
- `DefaultNonceValidationListenerImpl` : This listener throws a `NonceValidationException` in case a validation failed. It is up to the application to react accordingly. The most secure way would be to react with an instant termination of the current session.

Using a combination of listeners is as well possible.

NonceProvider:

The `com.icw.ehf.commons.security.web.csrf.NonceProvider` supplies the CSRF Guard with nonces. Currently, eHF Commons Security provides you with these implementations:

- `DefaultNonceProvider` : This nonce provider returns randomly generated UUIDs as nonces. Such nonces are not very secure since no cryptographic operations are performed to produce a strong nonce. So, it is recommended to not use it in production.
- `RandomNonceProvider` : This nonce provider produces cryptographically strong pseudo-random values as nonces and is suitable to use in production.

NonceIntroducer: For being present in subsequent requests, the nonces have to be introduced into responses. This is done by `com.icw.ehf.commons.security.web.csrf.NonceIntroducer` implementations. The nonce introducers are the only web-technology-aware components in the CSRF Guard implementation. Each nonce introducer is in charge of extending responses produced by a certain web technology. Currently, eHF Commons Security provides you with these implementations:

- The `XHTMLResponseExpander` introduces the nonce for XHTML responses. Thereby, the nonce is introduced as hidden field inside `<form>` tags and is attached to links of `<a>` tags.
 - The `JSF12ResponseExpander` modifies responses produced by the JavaServer Faces 1.2 framework. Since all actions in JSF are triggered through JavaScript onclick events, this filter expands the onclick JavaScript code to incorporate the nonce there. Additional implementations for particular web technologies could be easily added.
- `AbstractResponseExpander` represents an abstract base class which already does the

parsing of responses. Concrete implementations only need to register on certain response document nodes they are interested to expand.

CSRGuardFilter: This component glues all the above mentioned components together. This realization of a servlet filter needs to be hooked into every Servlet-based application's filter chain to protect against CSRF attacks. The following configuration possibilities exist:

- Via property `nonceValidators` `NonceValidator`'s could be registered. If more than one is registered, validation is performed chained. Once a nonce is detected, the validation is aborted.
- Property `nonceValidationListeners` defines the listeners which are notified in case an expected nonce is not detected.
- The property `nonceProvider` defines the producer of nonces for the CSRF Guard filter. If not specified, the `DefaultNonceProvider` is taken as default.
- Nonce introducers are registered via property `nonceIntroducers`.
- The described workflow does only work, if the application always returns modified responses containing nonces. Unfortunately, the first initial request does not carry a nonce. In such a case, nonce validators will alarm because of a missing nonce. So, application entry points need special attention. With the property `entryPoints` you have to specify the entry points of your application. For those specified entry points, no nonce validation is performed. Try to keep the number of entry points as low as possible to provide minimal attack surface.

Configuration

The configuration of the CSRF Guard is most suitable when using the Spring framework. In this case, you define a `DelegatingFilterProxy` filter and a filter mapping for it in your `web.xml` descriptor:

```
<filter>
    <filter-name>csrfGuardFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
<filter-mapping>
    <filter-name>csrfGuardFilter</filter-name>
    <url-pattern>/webgui/*</url-pattern>
</filter-mapping>
```

The delegating filter proxy maps to a Spring bean with the same name as the filter and delegates calls to this bean. So, you need to specify the CSRF Guard filter bean in Spring:

```
<bean name="csrfGuardFilter"
      class="com.icw.ehf.commons.security.web.csrf.filter.CSRFGuardFilter">
    <property name="entryPoints">
        <set>
            <value>/webgui/welcome.jsp</value>
        </set>
    </property>

    <property name="nonceIntroducers">
        <set>
            <bean class="com.icw.ehf.commons.security.web.csrf.introducer.
XHTMLResponseExpander" />
        </set>
    </property>

    <property name="nonceValidationListeners">
        <set>
            <bean class="com.icw.ehf.commons.security.web.csrf.listener.
DefaultNonceValidationListenerImpl" />
        </set>
    </property>

    <property name="nonceValidators">
        <set>
```

```
<bean class="com.icw.ehf.commons.security.web.csrf.validation.  
RequestParamNonceValidator"/>  
    </set>  
    </property>  
</bean>
```

15 Security Modules

Sophisticated services to address the sensitivity of medical information (privacy, integrity and availability) are provided by the modules of the Security category.

The **Authentication** library module verifies the identity of users or external systems. Only authenticated requests are allowed to interact with eHF services.

The **Authorization** module manages access rights (permissions) and controls access to protected resources. Its services implement hierarchical role-based access control (hierarchical RBAC) mechanisms, including instance-based access control features and black list / white list functionalities.

The **Audit** module implements means to store audit information of security relevant actions. The auditing of events is critical for satisfying specific regional data security requirements. The User Management module can be used by a health care application to manage the life cycle of its userspecific information. The administration of user roles is supported as well.

Further security-related modules explained in this chapter are

- **certificate validation**,
- **content scanner**,
- and **secure token services**.

15.1 Audit

Any kind of application that deals with sensitive date is required to produce a security relevant audit log or event trail. The main motivation is to enable an auditor to analyze the data in order to detect breaches on data privacy or to proof a sequence of actions have been performed by certain individuals.

But this kind of forensic usage represents only a very common denominator for application dealing with sensitive information. Often an alerting system is built on top of an audit trail to automatically detect and indicate inconsistencies or problematic situations that require intermediate actions.

The eHF Audit in its current implementation focuses on the first use case described above. The module is intended to produce an application security audit trail that contains all data for forensic investigation.

Please note that the audit module is passive, meaning that it does not actively pull this information from the other eHF modules or modules based on eHF. In the contrary the application developers are required to configure which application event results in an audit event that is then logged as an audit entry in the audit module. The audit module is therefore the last element in a event processing chain that is configured dedicated to the needs of an application.

15.1.1 Domain Model

The domain model of the audit module is very simple. It only provides the structure of the audit entries. Please note that the audit module is developed with encryption support. This means that certain attributes are encrypted on application level before being persisted into the audit data source. This is done to protect audit against data privacy attacks by e.g. performing simply data correlations.

15.1.2 Usage

Using the eHF Audit module is fairly simple. The module exposes a single bean to the application context named `auditAuditEntryLoggerDtoAdapter`, which adheres to the interface

```
com.icw.ehf.audit.service.adapter.AuditEntryLoggerDtoAdapter
```

The interface provides a single method, which is

```
void log(List<AuditEntryDto> entry);
```

Using this interface audit entries can be logged into the audit module. Please note that the audit module does not provide any asynchronous processing of the audit entries. All processing and queuing is supposed to happen before the audit entry is logged to the audit module.

15.1.3 Configuration

The module can be simply included in any ehf based assembly by supplying the following dependencies:

```
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-audit-config</artifactId>
    <version>SNAPSHOT</version>
    <properties>
        <ehf-module>audit</ehf-module>
        <ehf-web-api>true</ehf-web-api>
        <ehf-persistent>true</ehf-persistent>
    </properties>
    <type>jar</type>
</dependency>
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-audit-runtime</artifactId>
    <version>SNAPSHOT</version>
    <properties>
        <war.bundle>true</war.bundle>
    </properties>
    <type>jar</type>
</dependency>
```

In order to function properly the following beans must be supplied on assembly level:

Bean securityService

The bean should adhere to the interface `com.icw.ehf.authorization.service.SecurityService`. The audit module uses this bean to check permissions on the method level. Please note that for writing to the audit no further object level permissions are required.

Bean auditDataSource

This bean is the data source to which the audit module is persisting the logged audit entries. It must implement the interface `javax.sql.DataSource`.

Bean cryptoService

As mentioned above the audit module is using encryption to protect the data. For encryption an implementation of the `com.icw.ehf.commons.encryption.api.CryptoService` interface is required to be supplied by the platform.

When configuring encryption on application-level the following configuration should be supplied:

```
<module module-id="com.icw.ehf.audit">
```

```

<config-element classification="identifier">
    <parameters key-pool="pool_identifier" />
</config-element>
<config-element classification="confidential">
    <parameters key-pool="pool_confidential" />
</config-element>
<key-pool id="pool_identifier"/>
<key-pool id="pool_confidential"/>
</module>

```

15.1.4 Dependencies

Audit only depends on core eHF functionality and the eHF encryption support.

15.2 Authentication

For secure healthcare systems it is essential to identify users interacting with the system. Authentication is the process of verifying the identity of a subject. This applies both to external systems or human users. When authenticating an external system (for example, a blood pressure gauge), the goal is to verify that the system is genuine. When authenticating a human user (for example, a patient or a doctor), the goal is to verify that you are not dealing with an imposter.

[Figure 56](#) shows the conceptual operation of authentication. The *subject* provides *credentials* to the *verification control*. Credentials provide information for identifying as well as verifying a subject. Verification techniques range from a simple login, which validates users based on something that only the user knows — like a password, to more powerful security mechanisms that use something that the user has — like tokens, public key certificates, and biometrics. The result of the verification process is an *authenticated subject* with populated *principals*. Principals are, for example, a qualifying name ("John Doe") or a user ID ("123-45-6789"), which distinguishes it from other subjects.

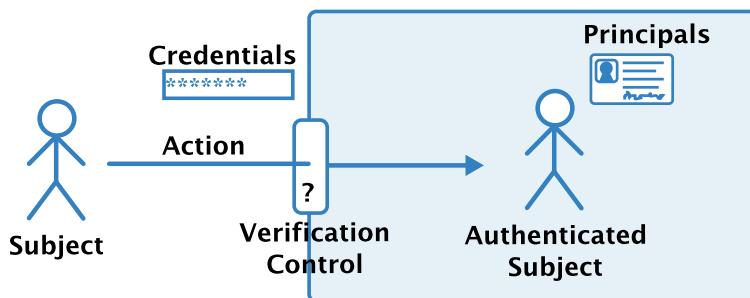


Figure 56: Authentication Concept

The eHF Authentication module provides several authentication methods. Users can either be authenticated by username and password, by making use of a secure PIN, or through a valid X.509 client certificate, using standard public key infrastructure (PKI) mechanisms. Authentication with health professional cards (HPC) and electronic health cards (EHC) are special cases of client certificate authentication. Furthermore, the eHF Authentication module can easily be extended with additional authentication methods.

15.2.1 Architectural Design

The eHF Authentication module ensures that all external requests are properly authenticated. The authentication methods are enforced via login filters that are based on servlet filters. In this way, the architectural design of the eHF Authentication module

breaks down a limitation of the Servlet 2.4 specification by allowing for more than one authentication method per web application. The servlet filters can be used in any combination and are available for all Servlet 2.4 specification authentication types (BASIC, CLIENT-CERT and FORM) as well as a custom PIN authentication method.

As illustrated in [Figure 57](#), the servlet filters delegate the validation of the supplied user credentials to the JAAS framework (Java Authentication and Authorization Service, <http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html>) that allows easily configurable security policies of the application server. According to the JAAS architecture, eHF Authentication is separated into one part using the JAAS API and one part providing a default implementation of the JAAS login module. The JAAS implementation delegates the authentication requests to the authentication adapter interface of the eHF Authentication module. It allows integration with various user stores (for example, LDAP or other legacy user management systems) to compare user data with the supplied credentials. Please note that the eHF Usermanagement module (see [User Management](#) on page 210) provides the AuthenticationService as the default implementation for accessing the eHF Usermanagement database store.

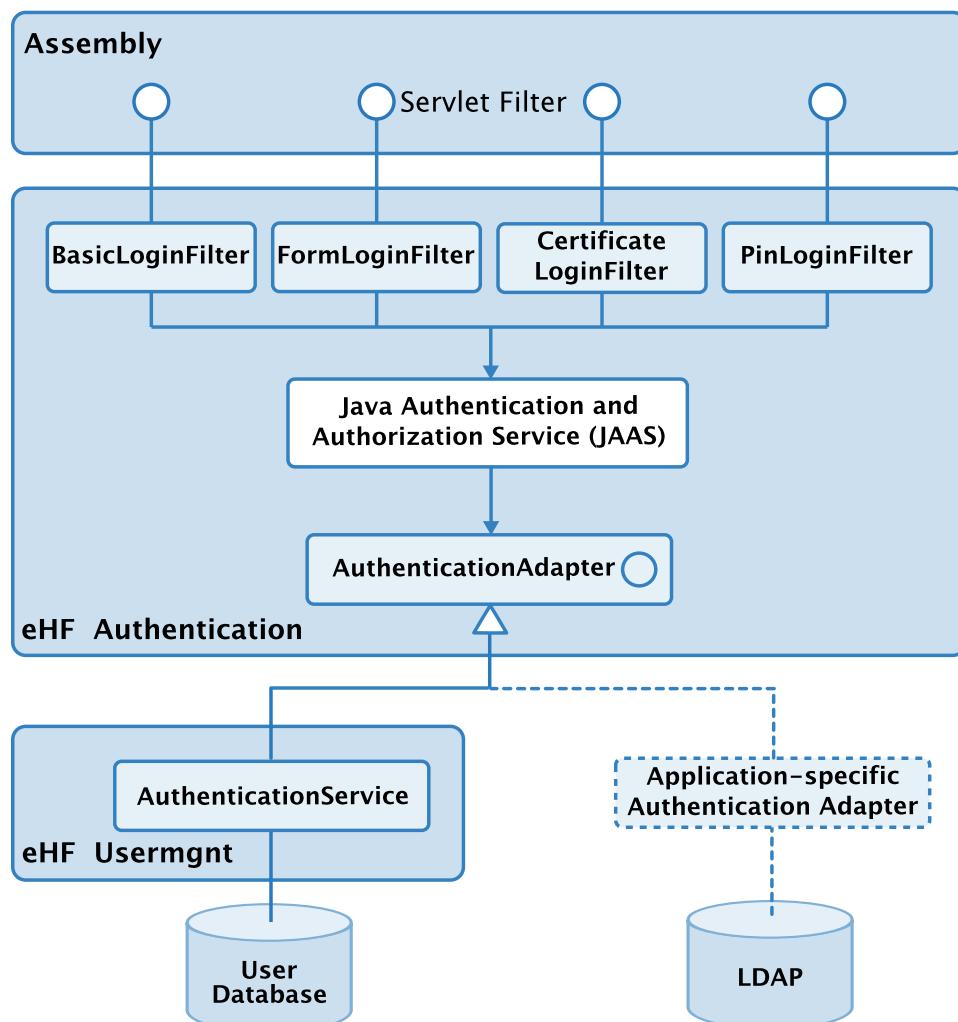


Figure 57: Authentication Architecture

As a more detailed example, the following happens while handling an authentication login request (see [Figure 58](#)): The LoginFilter is the abstract class of all login filters of the eHF Authentication module and receives the request. This login filter is responsible

for the authentication of subjects providing valid credentials. The login filter creates, first, a `DefaultCallbackHandler` instance (1) that will be used later by the login module. Please note that the `DefaultCallbackHandler` supports more than one authentication method by default (see JavaDoc). However, if you need application-specific authentication methods, you will implement both, a custom login filter and a custom callback handler. Next, the login filter creates an instance of the JAAS class `LoginContext` - parameterized with the callback handler - and calls the `login` method of the login context (2). The login context of JAAS describes the basic methods used to authenticate subjects and provides a way to develop an application independent of the underlying authentication technology. A configuration specifies the `LoginModule` to be used with a particular application (3). Therefore, different login modules can be plugged in under an application without requiring any modifications to the application itself. The eHF Authentication module provides the `DefaultLoginModule` implementation. With the help of the call back handler it extracts the provided credentials from the authentication request (4) and delegates the actual authentication decision to a class that implements the `AuthenticationAdapter` interface (5). The called `login` method returns the identities (or better said principals), which will be added to the authenticated subject.

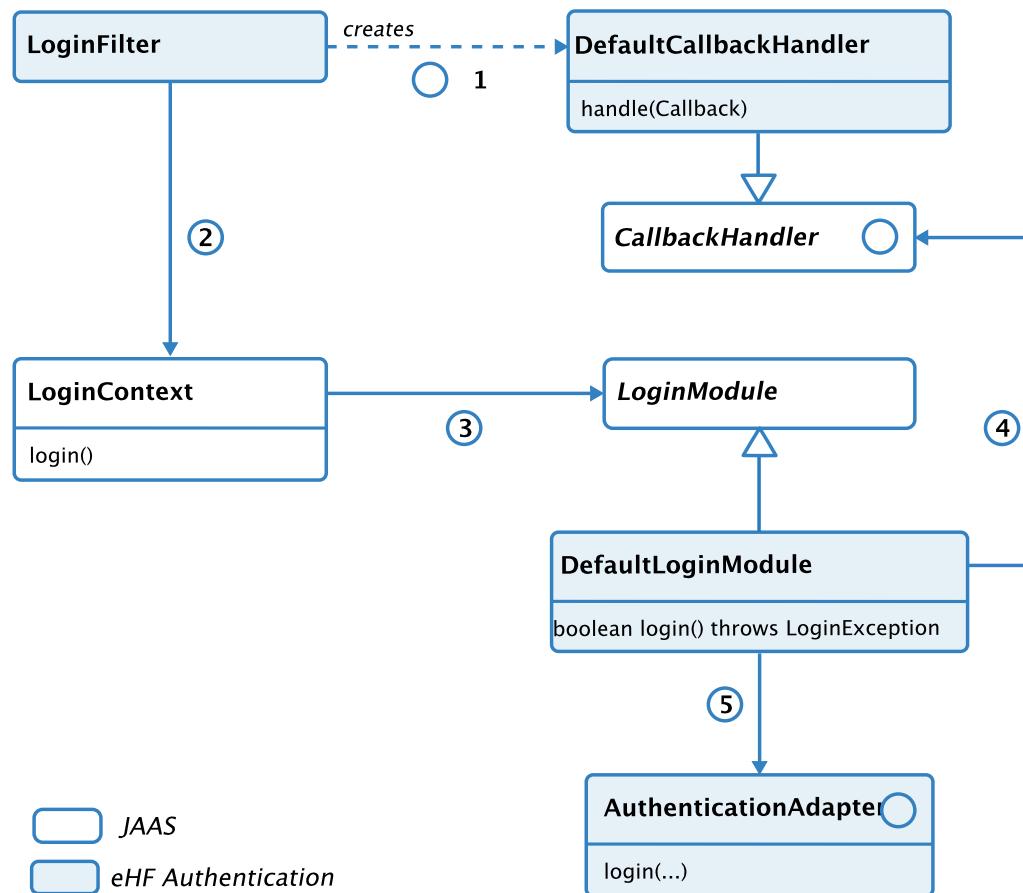


Figure 58: eHF Authentication Architectural Design

The class `LoginFilter` is the base class of the supported authentication methods listed in [Table 19](#). The first three filters cover the mandatory authentication methods according to the Servlet 2.4 specification. The PIN login filter is an enhancement for special use cases.

Filter Type	Description / Usage
BASIC Login	Performs username and password authentication. Makes use of browser's HTTP BASIC authentication capabilities.
Form Login	Performs username and password authentication. Credentials are provided through standard HTML web forms.
Client Certificate Login	Requires an X.509 certificate. Incoming certificates are validated against the server's truststore. Authentication with health professional cards (HPC) and electronic health cards (EHC) are special cases of client certificate authentication.
PIN Login	Authenticates users providing a secure PIN. The user's PIN is requested through a standard HTML web form.

Table19. Supported Login Filters of eHF Authentication

15.2.2 Usage

You do not use the eHF Authentication module from other eHF-based modules, but rather prohibit unauthenticated access on a system level from client side requests to the application server. The use of the eHF Authentication module that is configured in the application server forces the client side to provide proper credentials.

The following listing shows an example of how you can provide the username and password credentials. The example makes use of the Jakarta Commons HttpClient library (see <http://hc.apache.org/httpclient-3.x/index.html>). HttpClient handles authenticating with servers almost transparently, the only thing a developer must do is actually provide the login credentials.

```
// (1)
HttpClient client = new HttpClient();
// (2)
client.getState().setCredentials(
    new AuthScope("[host]", 443, "realm"),
    new UsernamePasswordCredentials("username", "password")
);
// (3)
GetMethod get = new GetMethod("https://[host]/[service]");
// (4)
get.setDoAuthentication( true );
try {
    // (5)
    int status = client.executeMethod( get );
} finally {
    // release any connection resources used by the method
    get.releaseConnection();
}
```

After creating an instance of HttpClient with default parameters (1), you pass the credentials to HttpClient (2). They will only be used for authenticating to servers with the realm "realm" on the "[host]". To authenticate against an arbitrary realm or host change the appropriate argument to `null`. In this example, credentials for basic authentication (username and password) are provided. Afterwards, you create a GET method that reads a resource or service over HTTPS (3). The next line of code tells the GET method to automatically handle authentication (4). The method will use any appropriate credentials to handle basic authentication requests. Setting this value to `false` will cause any request for authentication to return with a status of 401. It will then be up to the client to handle the authentication. Finally, the GET method is executed (5).

15.2.3 Configuration

The eHF Authentication module requires various configuration files that are contained in the assembly and, for security reasons, are located on the application server (see [Figure 59](#)).

The eHF assembly provides the servlet filters that are configured in `filters.fragment` and `filters.mapping.fragment`, located in the directory `src/main/config/merge/assembly/web-xml`. These fragment files are merged into the `web.xml` while executing the build process. The servlet filter delegates the authentication to the login filters of the eHF Authentication module. The login filters are Spring components. You find their corresponding Spring configuration in the file `ehf-authentication-context.xml` of an assembly.

As described in the [Architectural Design](#) on page 153 the eHF Authentication module is set on top of JAAS that requires an application server configuration. Tomcat, for example, provides this configuration in the directory `[Tomcat_Home]/config`. If you use the certificate login filter, you will need a resolver configuration file. The resolver configuration specifies, which information is needed from a certificate. The following sections describe the configuration artifacts in detail.

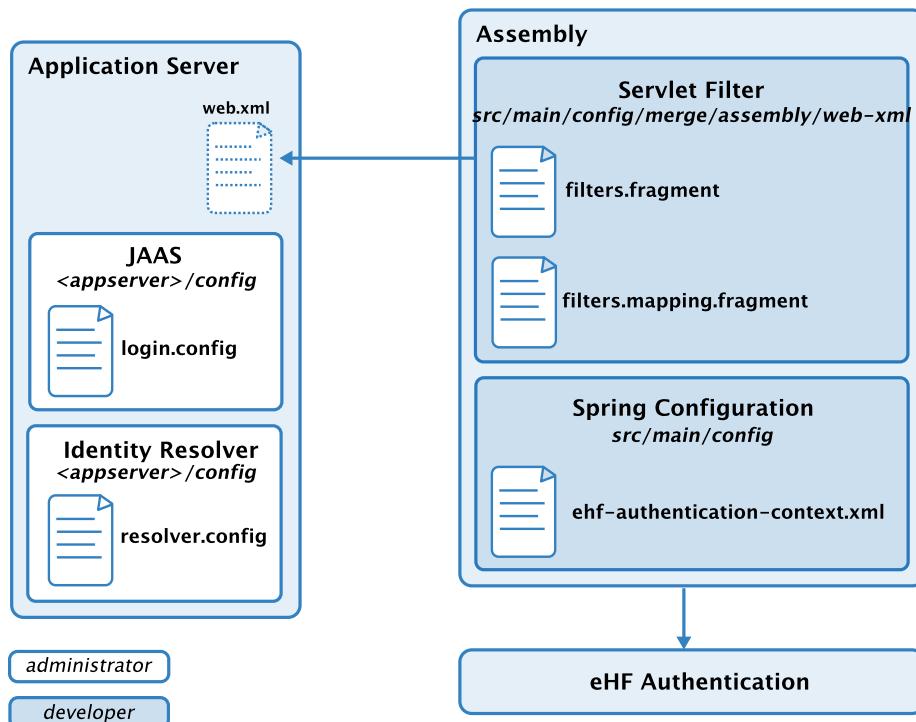


Figure 59: eHF Authentication Configuration Artifacts

Login Filter Configuration in the Assembly

The four existing login filters could be declared in the standard Servlet specification manner by adding them to the `web.xml` file inside a deployable artifact. Because we use Spring a lot in eHF, we recommend to use `org.springframework.web.filter.DelegatingFilterProxy` filters for your convenience. This allows you to inject properties of login filters with default Spring mechanisms and releases you from declaring init parameters for your filters inside the `web.xml` descriptor and keeping the `web.xml` clean and simple.

```
[...]
<filter>
```

```

<filter-name>formLoginFilter</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>

<filter-mapping>
<filter-name>formLoginFilter</filter-name>
<url-pattern>/webgui/*</url-pattern>
</filter-mapping>
[...]

```

This web.xml configuration block declares such a servlet filter which delegates to a Spring bean with ID formLoginFilter, which implements the javax.servlet.Filter interface. In the example above, this filter secures all web resources of the pattern /webgui/* by enforcing authentication. The login filter configuration is done via Spring:

```

[...]
<bean name="formLoginFilter"
      class="com.icw.ehf.authentication.web.form.FormLoginFilter">
    <property name="realm" value="ehf" />
    <property name="loginUrl" value="/login.jsp" />
    <property name="defaultUrl" value="/webgui/welcome.jsp" />
</bean>
[...]

```

A login filter always needs to get the security realm injected. Security realms are a concept from JAAS to distinguish parts of applications with different security constraints. For example, one application could be configured to consult different user stores for different URL patterns. All login filters have additional properties for their special purposes. The form login filter, for example, needs to know the URL of the login form (loginUrl) plus the URL to redirect to if the authentication was successful (defaultUrl). The following configuration listing shows an example configuration for the four provided login filters. For the complete list of configuration properties, please consult the corresponding Javadoc of the login filters.

```

<bean name="basicLoginFilter" class="com.icw.ehf.authentication.web.basic.
BasicLoginFilter">
    <property name="realm" value="ehf" />
    <property name="realmName" value="eHF" />
    <property name="retry" value="false" />
</bean>

<bean name="formLoginFilter" class="com.icw.ehf.authentication.web.form.
FormLoginFilter">
    <property name="realm" value="ehf" />
    <property name="loginUrl" value="/login.jsp" />
    <property name="defaultUrl" value="/webgui/welcome.jsp" />
    <property name="supportsTargetUri" value="true" />
</bean>

<bean name="certLoginFilter" class="com.icw.ehf.authentication.web.cert.
CertificateLoginFilter">
    <property name="realm" value="ehf" />
    <property name="optional" value="true" />
    <property name="beforeAuthenticationInterceptor">
        <bean class="com.icw.ehf.authentication.interceptors.
CertificateValidationInterceptor" >
            <property name="validator" ref="validator" />
        </bean>
    </property>
</bean>

<bean name="pinLoginFilter" class="com.icw.ehf.authentication.web.pin.
PinLoginFilter">
    <property name="realm" value="ehf" />
    <property name="loginUrl" value="/loginPin.jsp" />
    <property name="defaultUrl" value="/webgui/pinWelcome" />

```

```
<property name="pinContext" value="EMERGENCY-KEY" />
</bean>
```

JAAS Configuration

As stated before, the JAAS configuration has to be provided in order to retrieve JAAS realms. Detailed information about the JAAS login configuration file can be found at the web site <http://java.sun.com/j2se/1.5.0/docs/guide/security/jgss/tutorials/LoginConfigFile.html>. For eHF Authentication, a possible JAAS configuration could look like the following:

```
[...]
ehf {
    // Login module
    com.icw.ehf.authentication.jaas.DefaultLoginModule required

    // Login module options
    adapter=com.icw.ehf.usermgnt.service.AuthenticationService
    userServiceName=usermgntUserService
    logoutHandler.1=com.icw.ehf.MyLogoutHandler;
};

[...]
```

Following the JAAS convention, ehf is the realm name. The `com.icw.ehf.authentication.jaas.DefaultLoginModule` is the standard login module implementation for eHF-based applications. The default login module delegates authentication requests to the configured authentication adapter. In the above case, the `com.icw.ehf.usermgnt.service.AuthenticationService` adapter implementation from the eHF User Management is configured to be the authentication adapter. As well, there is one logout handler registered, which is executed every time a session is destroyed. At server startup time, this configuration file has to be contributed by setting the JVM argument `-Djava.security.auth.login.config`. The argument value is the location of the JAAS configuration file.

Identity Resolver Configuration

When a user authenticates with username and password, the user identifier is mapped to a user in the backend user store. In certificate-based authentication scenarios we deal with client certificates and do not have a user identifier by default. For certificate-based authentication the client certificates represent user credentials. As with usernames and passwords we have to ensure that the passed credentials are valid for every login attempt. Instead of storing the complete client certificate information in the database, eHF solely stores the authentication-relevant subset of information which uniquely identify a user. For each client certificate login attempt, the authentication-relevant information is extracted from the certificate and compared with the value in the database. As with passwords, the authentication succeeds if both values match.

The extraction of authentication-relevant information out of X.509 certificates is done by the eHF identity resolving which is part of the eHF authentication module. The different resolving strategies are maintained in the global configuration `resolver.config` file in the Tomcat `conf` folder. Each eHF-based application which needs to support client certificate authentication must provide this configuration file.

The resolver configuration file adheres to the following XML structure:

```
<resolver-config>
  <resolvers>
    <resolver>...</resolver>
    <resolver>...</resolver>
    ...
  </resolvers>
```

```
</resolver-config>
```

Each `<resolver>` element represents a different identity resolver strategy which will be used to resolve the authentication-relevant information out of certificates.

The abstract class `com.icw.ehf.authentication.cert.IdentityResolver` is the base class for all identity resolvers. Currently, the eHF provides the following concrete implementations of identity resolvers:

- `com.icw.ehf.authentication.cert.SerialNumberResolver` - this resolver extracts the certificate's serial number.
- `com.icw.ehf.authentication.cert.SerialNumberIssuerResolver` - An identity resolver which extracts both the certificate's serial number and the distinguished name of the issuing certificate authority (CA). X.509 certificates always carry information about the certificate subject and the issuer of the certificate. This resolver constructs the authentication-relevant information using the pattern [DN of issuer] : [serial number].
- `com.icw.ehf.authentication.cert.IssuerSerialNumberNotAfterResolver` - This resolver extracts the issuer distinguished name, certificate serial number and the not-after date as identity. The identity complies with the pattern [DN of issuer]:[serial number]:[not after date in milliseconds].
- `com.icw.ehf.authentication.cert.CommonNameResolver` - this resolver extracts the CN field from the certificate's distinguished name as authentication-relevant information.
- `com.icw.ehf.authentication.cert.SubjectFieldResolver` - A generalization of the `CommonNameResolver`. Instead of always retrieving the CN field, this resolver can be configured on which fields of the certificate's distinguished name should be extracted. The option `ownerIdentifierFieldList` of this resolver expects a comma-separated list of field keys (e.g. CN,OU,C). The authentication-relevant information is constructed using the separator `_` to differentiate between the different distinguished name fields.

To use an identity resolver strategy in your eHF-based application, you simply add it to the `resolver.config` configuration file and specify the fully qualified name of the resolver implementation:

```
<resolver resolverClass="com.icw.ehf.authentication.cert.SubjectFieldResolver">
  <options>
    <option key="..." value="..." />
  </options>
</resolver>
```

The `<options>` section allows for the provision of additional key-value settings to configure the resolver implementation appropriately. If none of the identity resolvers above are suitable, you are able to provide your own custom identity resolver by extending the abstract class `com.icw.ehf.authentication.cert.IdentityResolver`.

Typically, an eHF-based application will need to support client certificates issued by different CAs, so that more than one identity resolver is required. For more than one identity resolver configuration, the eHF has to determine the correct resolution strategy which should be applied. This selection criterion is expressed in so called certificate matchers. Such certificate matchers simply describe when the corresponding identity resolver should be selected. Therefore, each identity resolver configuration element has to provide a matcher definition as well:

```
<resolver resolverClass="com.icw.ehf.authentication.cert.SubjectFieldResolver">
  ...
  <matcher matcherClass="com.icw.ehf.authentication.cert.FieldRegexMatcher">
    <options>
      <option key="issuerName" value="CN=MyCA .* C=US" />
    </options>
  </matcher>
</resolver>
```

```
</matcher>
</resolver>
```

In the code example above, the `FieldRegexMatcher` certificate matcher was chosen. As with the identity resolvers, you can further configure the matchers by providing additional options. The `FieldRegexMatcher` is able to analyze different certificate portions as to whether their values comply with a configured regular expression. In the example, the matcher would report positive matchers for all client certificates which are issued by a CA called `MyCA` which is located in the United States. The `FieldRegexMatcher` can be configured using the following options:

- `issuerName` - the value of this option defines the regular expression which is evaluated on the distinguished name of the issuer.
- `subjectName` - the value of this option defines the regular expression which is evaluated on the distinguished name of the subject.
- `serialNumberName` - the value of this option defines the regular expression which is evaluated on the certificate serial number.
- `notBefore` - the value of this option defines the regular expression which is evaluated on the `notBefore` element of the certificate.
- `notAfter` - the value of this option defines the regular expression which is evaluated on the `notAfter` element of the certificate.

If you need other matching rules, you are always free to implement your own certificate matcher by implementing the `com.icw.ehf.authentication.cert.CertificateMatcher` interface.

The order of the identity resolver configuration is important. For each client certificate authentication, the identity resolution algorithm iterates over all configured identity resolvers and invokes each corresponding matcher. The first identity resolver whose certificate matcher returns `true` is taken to calculate the identity of the user.

15.2.4 Extension

The eHF Authentication module provides two extension points that are covered in the following sections.

Extending Login Filters with Business Logic

An application using eHF Authentication can extend the login filters with business logic specific for this application. For that purpose, eHF Authentication provides three interceptors that can be injected into every login filter:

- `BeforeAuthenticationInterceptor`: Is executed before the authentication is performed
- `AfterSuccessfulAuthenticationInterceptor`: Is executed after a successful authentication attempt
- `AfterFailedAuthenticationInterceptor`: Is executed after a failed login attempt

For more information about the interceptor interfaces, consult the Javadoc. Authentication interceptors are injected via Spring.

```
[...]
<bean name="formLoginFilter"
      class="com.icw.ehf.authentication.web.form.FormLoginFilter">
    <property name="beforeAuthenticationInterceptor">
      <bean class="com.icw.ehf.authentication.interceptors.
AlreadyLoggedInCheckInterceptor" />
    </property>
</bean>
[...]
```

In the example above, an interceptor is injected, which is always executed before an authentication is performed. In this example, the interceptor checks whether the user is already logged in. Please note, that interceptors can only be defined when using Spring for login filter definitions as shown above. Setting the filter init properties in the `web.xml` directly does not have any effect on the interceptor configuration of a login

filter. If your application needs to support additional authentication mechanisms which are not covered by the existing filter bundle, you could implement a custom login filter. For your convenience, extend `com.icw.ehf.authentication.web.LoginFilter` to reuse the generic implementation equally for all login filters provided by eHF.

Authentication Adapter Implementations

With the `com.icw.ehf.authentication.jaas.DefaultLoginModule`, eHF Authentication provides an implementation of the `javax.security.auth.spi.LoginModule` to realize the JAAS Service Provider Interface. The default login module itself delegates authentication requests to an implementation of the `com.icw.ehf.authentication.jaas.AuthenticationAdapter` interface. Such an authentication adapter is responsible for performing the authentication against a dedicated user store. The eHF User Management provides such an authentication adapter implementation which consults its internal user store. Through custom adapter implementations, it is possible to adapt legacy user stores.

Logout Notification

eHF Authentication supports logout notification by providing the interface `com.icw.ehf.authentication.jaas.LogoutHandler`. Multiple logout handler implementations can be registered to be notified about logout activities (either through user logout or session timeout). This can be used to trigger cleanup activities (for example session-based cache cleaning) before the session is destroyed.

15.3 Authorization

Health care applications typically deal with highly sensitive data. Access to patients' health records - containing personal data, medical observations, treatments and other confidential information - must at all times be controlled and restricted. It is of vital importance that an application developed with the eHealth Framework (eHF) is able to protect its data from unauthorized access.

[Figure 60](#) gives an overview of the general authorization concept. Applications first need to authenticate the source of the request (please refer to [Authentication](#) on page 153). Once authenticated, the authorization process that takes place involves five parties. To begin with, the result of a successful authentication is a *subject*. A subject may be any entity, be it a person or service. Once the subject is authenticated, it is populated with associated principals. For example, a person may have a qualifying name ("John Doe") and a user ID ("123-45-6789"), which distinguish it from other subjects. The other parties involved are the called *resource*, the *action* that will be performed on the resource, and the *access control* in the form of a policy. The access control will decide whether the access is allowed or not. The decision is based on the access control relevant attributes: subject, action, and resource. However, sometimes additional attributes from the *environment* are needed in order to make the authorization decision.

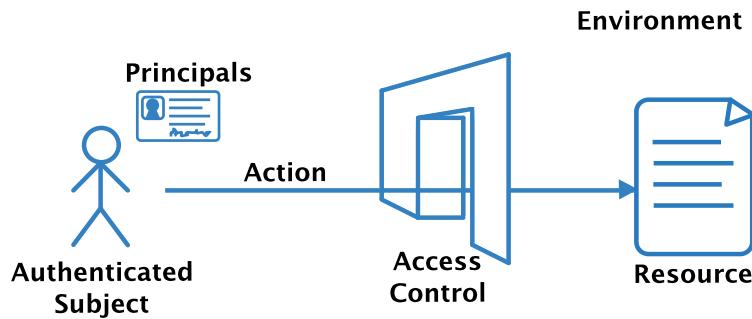


Figure 60: Authorization Request

The eHF Authorization module provides a hierarchical role-based access control that can grant or deny access to arbitrary resources. It is capable of selecting individual instances or groups of instances as protected resources. In particular, this instance-based access control allows for the management of permissions down to the level of individual domain object instances. Furthermore, the eHF Authorization module provides an extensible access control framework for authorization that can be extended for realizing application-specific or module-specific access control requirements.

15.3.1 Architectural Design

The access control architecture of the eHealth Framework defines three participating components: policy enforcement point (PEP), policy decision point (PDP), and policy administration point (PAP). The interaction between the three of them is shown in [Figure 61](#). The PEP, commonly referred to as a "gatekeeper", intercepts the business request (1). The PEP then forwards information about itself and the subject that has made the access request to the PDP (2). The PDP evaluates the authorization request relative to the security context and returns the decision to the PEP (5). It is possible that the authorization request does not contain all the access control relevant attributes necessary to make the access decision. In this case the PDP may request further access control relevant attributes from the environment, for example, business services of the application (3). The PDP then consults the configured policy from the PAP to determine whether access should be allowed or denied (4). A policy is typically a list of permissions (access control list, ACL). Additionally the PAP also provides services for administrating the policies of the policy persistence store. The intercepted business request will be performed if the authorization decision grants the access (6). Otherwise, the business request will be interrupted.

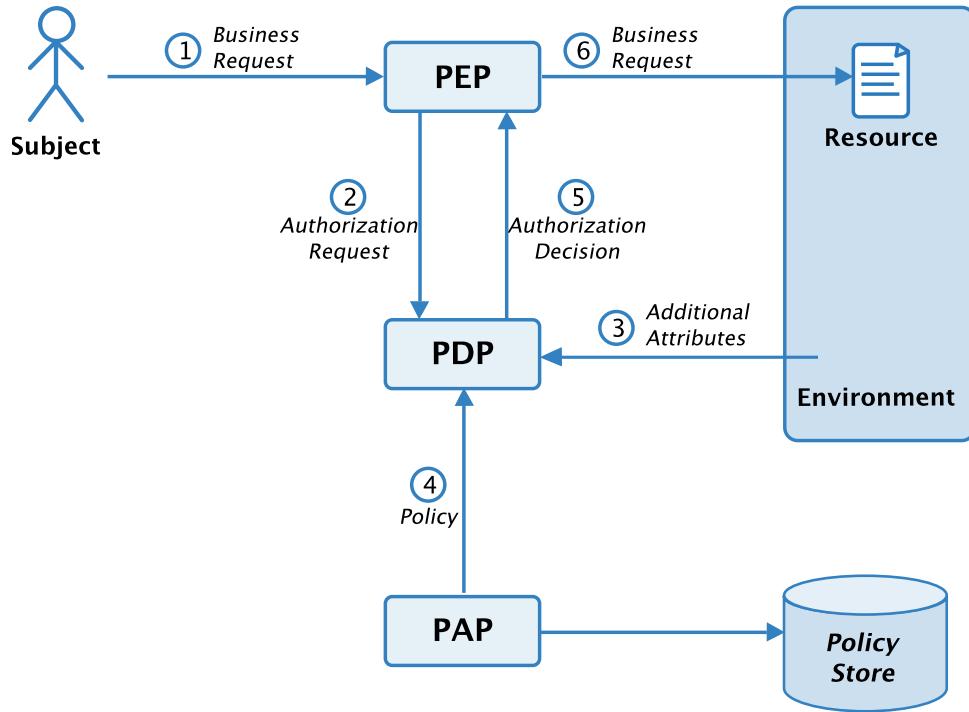


Figure 61: Interaction between PEP, PDP, and PAP while handling a business request.

Authorization is a typical cross-cutting concern, since it potentially affects every single call to a service or the domain layer. Thus, it is important to separate authorization logic from business logic. [Figure 62](#) shows where the components PEP, PDP, and PAP are realized in the eHF. The eHF implements the PDP and PAP fully in the eHF Authorization module. It provides the `SecurityService` interface for receiving authorization requests and the `PermissionManager` interface for administrating the policies. Only the policy enforcement resides in the individual business modules, since it needs to intercept all business service requests. However, you do not need to implement the policy enforcement in eHF-based modules manually, as all enforcement related artifacts are generated for you by the eHF Generator (see [Generator](#) on page 88).

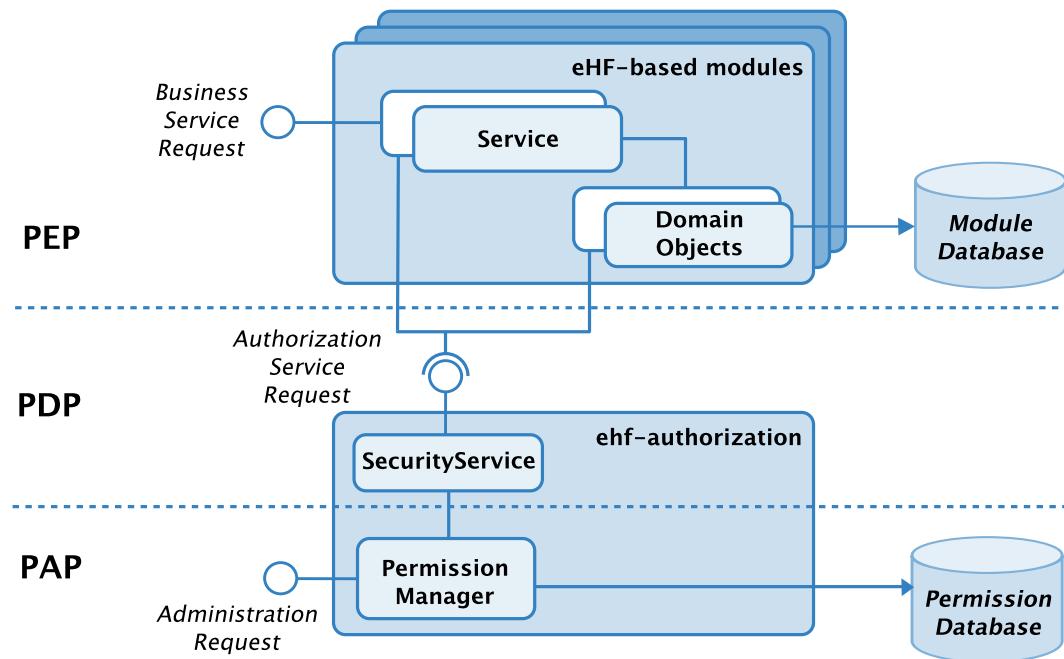


Figure 62: eHF Authorization Design

Policy Enforcement Point

The eHF Authorization uses the Security Annotation Framework (SAF, <http://safr.sourceforge.net>) for the interception of service operations. SAF is an instance-level access control framework based on Java 5 annotations and AOP technologies (Spring AOP, AspectJ). SAF is a mechanism for authorization policy enforcement, i.e. the SAF `SecurityInterceptor` intercepts the call of annotated code and enforces authorization.

Policy Decision Point

The policy decision point is based on the JAAS framework (Java Authentication and Authorization Service, <http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html>) as a generic, extensible framework for restricting access to arbitrary resources. From a user perspective, no deeper understanding of JAAS is required, since it is mostly hidden behind eHF-specific facades and interfaces.

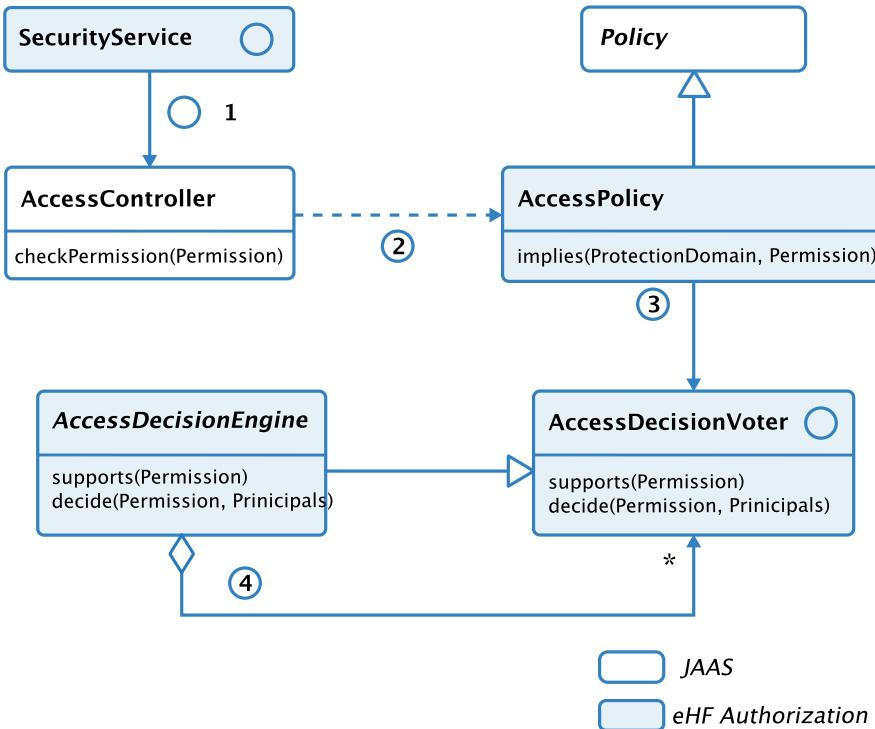


Figure 63: Overview of the processing of the access decision process.

The central interface of the policy decision point is the `SecurityService`. Typically, it is called by the policy enforcement interceptors of the eHF-based modules. The processing of an authorization request consists of the following steps (also shown in [Figure 63](#)):

1. The default implementation of the `SecurityService` acts as a facade for JAAS' `AccessController`. It creates a `java.security.Permission` object based on the access relevant attributes and calls the static method `AccessController.checkPermission(Permission)`.
2. The access controller indirectly calls the method `implies` of `com.icw.ehf.authorization.policy.AccessPolicy` with the help of further JAAS' objects. `AccessPolicy` is a sub class of the abstract class `java.security.Policy`. The `ProtectionDomain` parameter of the `implies` method holds information about the current user.
3. The `AccessPolicy` delegates again the authorization decision to an `AccessDecisionVoter` instance. First, it calls the method `supports(Permission)`, which states whether a given permission can be handled by the called voter or not. Second, it calls the method `decide(Permission, Principals)` that makes the final decision, whether access will be granted. An implementation of the `AccessDecisionVoter` typically encapsulates the access decision logic of a single access control model (such as the instance-based access control).
4. If an eHF-based application needs more than one access decision voter, then you can make use of an implementation of the abstract class `AccessDecisionEngine`, which is a composite access decision voter. The `AccessDecisionEngine` combines the access decision results of all configured access decision voters into a final access decision result.

Please note that "not permitting" an access request is not the same as explicitly "denying" the request. If only one access decision voter is active in an application, then the final decision result would be to reject the request in both cases. However, if more than one access decision voter is active, then the final result could be different depending on the configured combination algorithm. Therefore, the `decide` method of an `AccessDecisionVoter` does not simply return a boolean value like the `implies` method of the `Policy` class. Instead, it returns a value of the `AccessDecisionResult` type. Possible values of the `AccessDecisionResult` type are:

PERMIT

Permission is granted to the given principal(s).

NO_PERMIT

Permission is not granted to the given principal(s) by this voter. It does not object to other voters' permissions. In this case the access decision voter simply states that he cannot permit the access, but he does not mind if another voter goes on to explicitly permit or deny access.

DENY

Permission is not granted to the given principal(s), and this voter explicitly objects to any other voters' permission. In this case the access decision voter explicitly prohibits the access.

While the eHF access control framework is extensible for realizing specific access control models, it already provides two default access control voter implementations (see [Figure 64](#)). `PermissionManagerImpl` provides the actual implementation of the instance-based access decision logic. `FirstPermitOrDenyDecisionEngine` is a subclass of `AccessDecisionEngine` that iterates through the configured list of voters and returns the first `PERMIT` or `DENY` access decision. If no configured voter returns `PERMIT` or `DENY`, then the final access decision result is `NO_PERMIT`.

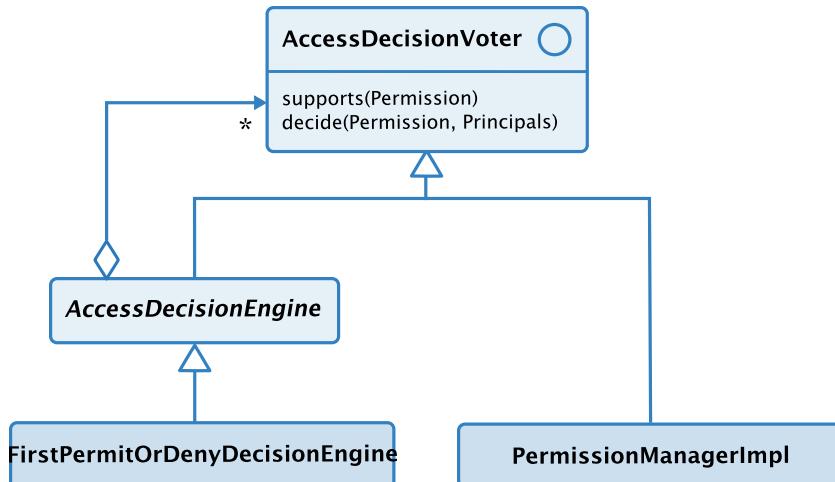


Figure 64: Access control voter default implementations.

Policy Administration Point

The policy administration point consists of the service `PermissionManager` to manage the policy data and to provide that data to the policy decision point. For the instance-based access control, this functionality is implemented by the class `com.icw.ehf.authorization.service.PermissionManagerImpl`. Please note that `PermissionManagerImpl` implements both, the policy administration point and policy decision point mentioned in the section before.

Reusable and Composable Permission Profiles

The eHF Authorization module supports the concept of permission profiles to simplify the usage and management of instance-based access control in an eHF-based application. A permission profile is a named collection of permissions that can be assigned to a user, a role or any other principal in a single assignment operation using the name of the profile.

Each eHF module offers a set of reusable basic permission profiles which represent certain capabilities or use cases of the eHF module that are ideally independent from the eHF-based application in which the eHF module is used.

15.3.2 Domain Model

The eHF Authorization module supports instance-based access control for eHF-based applications by the `PermissionManager`. It provides a fine-grained permission domain model that enables access control down to the level of single domain object instances. [Figure 65](#) shows the domain model. The elements will be covered in this section.

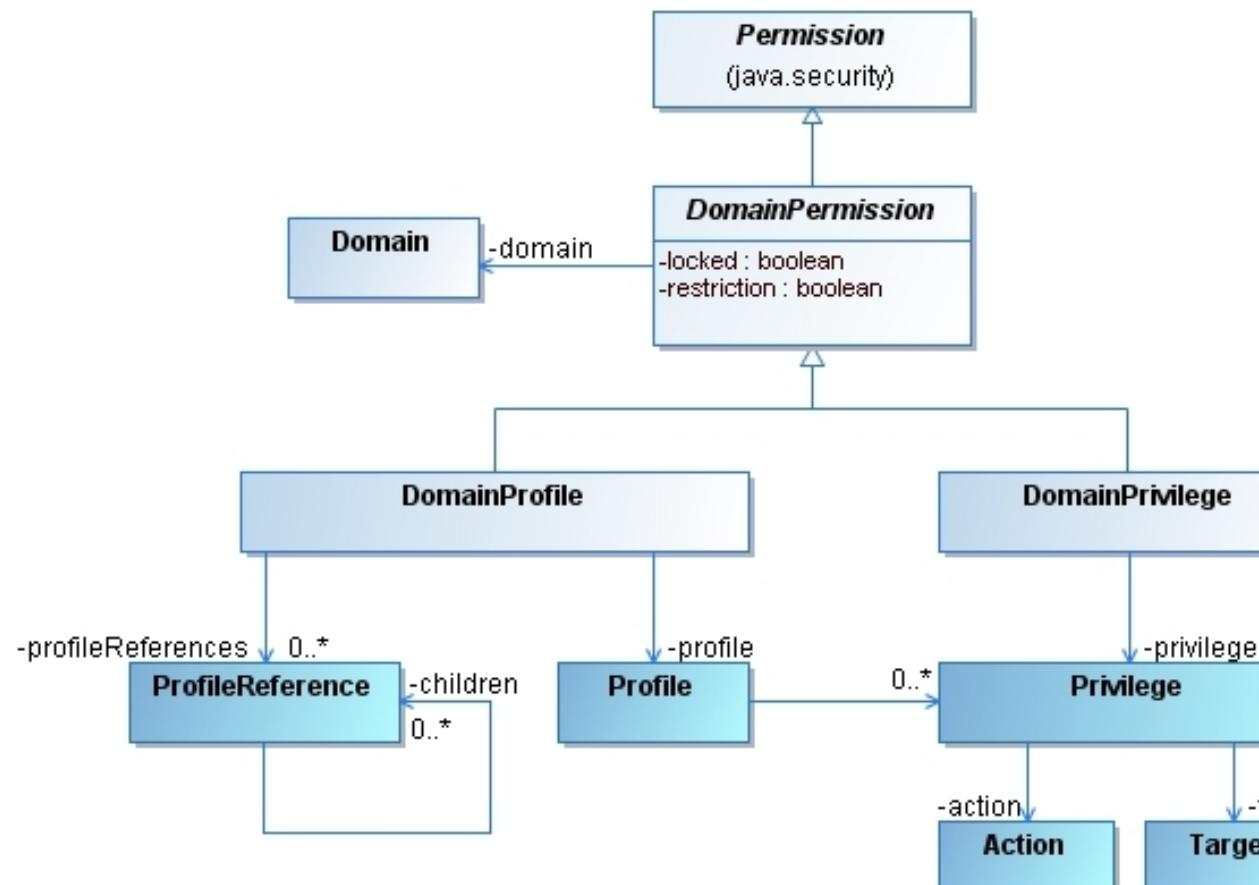


Figure 65: Domain Model of eHF Authorization

Domain Permission

A domain permission associates a permission of JAAS with a domain. As described in the [Architectural Design](#) on page 163, the authorization module is implemented based on JAAS. Using JAAS, applications can create their own permission types as extensions of the Java platform `java.security.acl.Permission` type. The eHF Authorization module provides the `domain permission` class for that purpose.

The *restriction* attribute controls whether the domain permission is permissive or prohibitive, that is, whether this is a permission or a restriction. The value `false` indicates a permission, the value `true` would be a restriction.

Domain

The *domain* is a security-relevant logical grouping of domain objects and can therefore be used to implement instance-based security in a very efficient way. By logically associating a user with a domain, a user can be easily granted access rights to his "own" domain object instances. Please note that the *domain* is called the *scope* in other eHF-based modules. However, this is just another term for the same thing.

Domain Privilege

A *domain privilege* is a concrete implementation of a domain permission and provides a complete representation of a permission in the authorization context: What action (e.g. read, create, update, or delete) is allowed on which target (e.g. domain object or service) within the specified domain (scope).

Action

An *action* can be performed on a target, that is, one or more domain objects or a service. Access to individual actions can be granted or denied. This is a list of the possible actions:

Action	Description
CREATE	Permission to create a new domain object instance.
READ	Permission to read a domain object instance.
UPDATE	Permission to update a domain object instance.
DELETE	Permission to delete a domain object instance.
EXECUTE	Permission to call an operation (other than a general UPDATE) on a domain object instance.
AUTH_CRUDE	Grant or revoke CREATE, READ, UPDATE, DELETE, or EXECUTE permissions on a <i>target</i> .
AUTH_CRUDE_DELEGA	Grant or revoke the AUTH_CRUDE permission on a <i>target</i> .
AUTH_SEED	Grant or revoke the AUTH_CRUDE_DELEGATE or AUTH_SEED permissions on a <i>target</i> .
ADD	Permission to add one domain object to another object.
REMOVE	Permission to remove an object from another object.

Target

A *target* selects one or more protected resources. Typically, this is the whole extent (i.e. all instances) of a specific permission classifier, or one specific domain object instance. A target is defined by the following attributes:

Attribute	Description
Type	The permission classifier of the protected resource. The permission classifier consists of the package and a classifier. For example, the permission classifier <code>my.package.MyClassifier</code> comprises the package <code>my.package</code> and the classifier <code>MyClassifier</code> of a domain object. If the classifier is not set, then the permission

Attribute	Description
	classifier is initialized with the default name <code>Object</code> , for example <code>my.package.Object</code> . Please compare to Usage on page 171 how to define classifier and permission classifier.
Role	This field reflects the role an object plays in an association with another object, which may be used if there are multiple associations between two types. The role clarifies a particular association and can be defined by setting the <code>code</code> tag on an association in the domain model. Whenever a domain object is accessed through navigation of such an association, the latter's <code>code</code> tag is propagated to the former, thus allowing for association-based permission checks.
Context	For a composition, this refers to the owning object's permission classifier. It is unused otherwise.
Identifier	The unique identifier of a particular domain object instance.

Targets can be specified with wildcards, thus selecting all possible values for the wildcarded attribute. Here are some examples in the format (`type, role, context, identifier`) :

Target	Selects
(*, *, *, *)	All domain objects.
(<code>my.package.Program</code> , *, *, *)	All domain objects that are instances of a type with the classifier <code>Program</code> in the package <code>my.package</code> . Typically, this would be the <code>my.package.Program</code> type itself.
(<code>my.package.Object</code> , *, *, *)	All domain objects that are instances of any type in the package <code>my.package</code> that don't have an explicit classifier.
(<code>my.package.Program</code> , *, *, af8d3291-5a66-4bf8-993cf8a9b5296c37)	A specific instance of the classifier <code>my.package.Program</code> with the given ID.
(*, *, <code>my.package.Program</code> , *)	All domain objects that are contained in an instance of a type with the classifier <code>my.package.Program</code> , thus being captured by its context. Note that this may cover instances of different types, and even different classifiers. The given classifier refers to the owning instance.
(<code>com.icw.ehf.usermodel.service.PersonFinder</code> , *, CON, *)	The service <code>PersonFinder</code> in the context <code>CON</code> .

Domain Profile

A *domain profile* defines and groups a set of permissions for one domain.

Profile Reference

A *profile reference* can be used to create a so called "named" or "predefined" permission profile by associating a set of permissions with a name.

Furthermore, a *profile reference* can have a self-reference. This self-reference allows a *profile reference* to hold a set of children, which consequently enables the composition of permission profiles.

A predefined set of *domain privileges* that can be uniquely addressed by a triple of *category*, *name*, and *version*. Whereas a domain profile defines privileges, a profile reference only references them. Profile references are typically predefined through bootstrap or import processes.

Using the *PermissionManager* service, a profile reference can be granted to a user principal or role principal (please compare to [User Management](#) on page 210).

Domain Profile Assignment

Finally, a *domain profile assignment* is the actual assignment of permissions - in the form of one domain profile - to one specific principal of JAAS. It is only through such an assignment that any principal can be granted (or revoked) permissions.

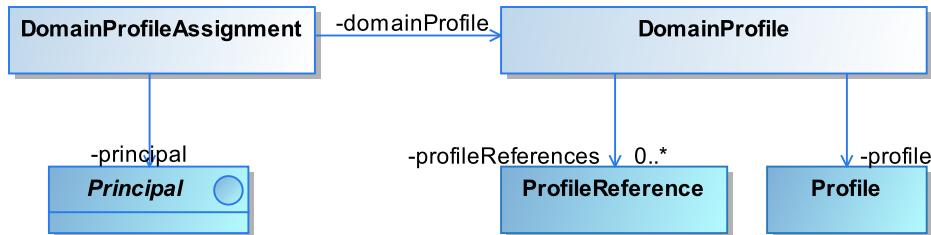


Figure 66: A Domain Profile Assignment connects a Domain Profile with a Principal.

15.3.3 Usage

In eHF-based modules, you have two working points to define the authorization security on the policy enforcement level. On the one hand, you secure service operations by security annotations. On the other, you declare permission classifiers of domain objects in the domain models.

Furthermore, you can use the *SecurityService* and *PermissionManager* on the policy decision and policy administration level, respectively. However, this should be seldom necessary, since the policy enforcement and policy decision work almost automatically if an eHF-based assembly is properly configured.

Security Annotations

Service operations are secured with annotations of the [Security Annotations Framework](#) (SAF). For example, the abstract class `AbstractDomainObjectCrudService` declares annotations for the public operations `create` and `update` as shown in the following listing. The annotations `SecureAction.CREATE` and `SecureAction.UPDATE` enforce write checks, that is, the eHF Authorization checks if the subject has permission to create or update the domain object.

```

public abstract class AbstractDomainObjectCrudService<E extends DomainObject>
    implements ObjectManagerAwareCrudService<E> {

    public E create(@Secure(SecureAction.CREATE) final E domainObject) {
        [...]
    }

    public void update(@Secure(SecureAction.UPDATE) E domainObject,
  
```

```

        UpdateParameter updateParameter) {
    [...]
}
}

```

The next listing shows the usage of security annotations for reading domain objects or collections with `Filter`. The list of domain objects is filtered by the authorization mechanism. Only those are returned for which the subject has read permissions.

```

@Filter
public List<E> loadByScope(final String scope, final boolean includeDependencies)
{
    final ManagerContext context = new ManagerContext(includeDependencies);
    try {
        return getDomainObjectManager().loadByScope(scope, context);
    } finally {
        // support the garbage collector
        context.cleanup();
    }
}

```

When you write custom service operations, you have to declare the appropriate security annotations as required.

Permission Classifier - Defining Secure Targets

A permission classifier is used to define access control on domain objects or services. The permission classifier is asked by the security service and compared to the target of a privilege. (Targets are explained in detail in section [Domain Model](#) on page 168).

You specify a permission classifier in the UML domain model of an eHF-based module. The permission classifier is defined by the tag *classifier* of a domain object (please see the ehf-attribute [stereotype](#) on page 307).

When running the eHF Generator, the annotation `permissionClassifier` is added to the generated domain object. For example, given a domain object `com.icw.ehf.mymodule.domain.MyDomainObject` with a classifier `MyClassifier` in the model, the eHF Generator generates the annotation as shown in the following listing:

```

package com.icw.ehf.mymodule.domain;
[...]
@PermissionClassifier("com.icw.ehf.mymodule.domain.MyClassifier")
public abstract class MyDomainObjectBase extends [...] {
    [...]
}

```

Please note that the permission classifier is made up of the package name, whereas the classifier is specified by the name only. If the classifier is not set in the model, then the permission classifier is initialized with the default name `Object`:

```

package com.icw.ehf.mymodule.domain;
[...]
@PermissionClassifier("com.icw.ehf.mymodule.domain.Object")
public abstract class MyDomainObjectBase extends [...] {
    [...]
}

```

The permission qualifier of a domain object is accessible by the public method `getPermissionClassifier` of the class `com.icw.ehf.common.domain.AbstractBaseDomainObject`. The security service will call this method while processing an authorization request and, based on that, prepare the authorization decision. If your domain object needs custom permission classifier, you could override the implementation of this method.

Using the Security Service

The `SecurityService` is basically just a convenience interface that acts as a facade for JAAS' `AccessController`. Clients that want to check whether the current user is permitted to perform a given action on a given target should use this interface. It is exposed as a bean with the name `securityService` in the Spring context of the eHF Authorization module.

Using the Permission Manager

The `PermissionManager` can be used to query and manipulate permissions. It thus acts as the public interface for the policy administration point and the persistent data model. It is exposed in the authorization module's Spring context as a bean with the name `permissionManager`.

Working with permissions and the `PermissionManager` service basically is all about getting, setting, adding, or removing assignments between profiles and principals of subjects. There are various methods to do this. Please refer to the JavaDoc documentation for the exact signatures and parameters.

For querying `DomainProfileAssignments`, the `PermissionManager` provides overview variants of some methods. These return stripped assignment objects whose profile data is not set. They execute faster than their non-overview counterparts. Clients are advised to use these if detailed profile data is not required.

Finally, the `PermissionManager` provides read access to `ProfileReferences`.

Using Reusable Basic Permission Profiles

A basic permission profile is defined within an eHF module and represents a certain capability or use case of this eHF module. A basic permission profile that is defined in such a way can then be reused by any eHF-based application that uses this eHF module.

For example, the eHF Usermgmt module provides the capability of *Creating a User*. To use this capability, the public API of the eHF Usermgmt module provides accor-dant methods like `createUser` and `createUserForDomain` via the `UserService` interface. The following Figure [Figure 67](#) shows an example of how basic permission profiles can be defined by using the mentioned XML syntax for permission profiles.

```

<authorization>
    <namedProfiles>

        <namedProfile>
            <profile>
                <permission>
                    <target>
                        <type>com.icw.ehf.usermodel.domain.User</type>
                        <role>*</role>
                        <context>*</context>
                        <identifier>*</identifier>
                    </target>
                    <actions>
                        <action>CREATE</action>
                    </actions>
                </permission>
                <permission>
                    <target>
                        <type>com.icw.ehf.encryption.domain.Object</type>
                        <role>*</role>
                        <context>*</context>
                        <identifier>*</identifier>
                    </target>
                    <actions>
                        <action>CREATE</action>
                    </actions>
                </permission>
            </profile>
            <category>basic</category>
            <name>com.icw.ehf.usermodel.createUser</name>
        </namedProfile>

        <!-- other named Profiles... -->

    </namedProfiles>
</authorization>

```

Figure 67: An example for a basic permission profile

The basic permission profile example shown in Figure 1 represents the *Create a User* capability and consists of two instance permissions that must be granted to a principal

so that this principal is able to create a user in an application. Other permissions which are not necessary to execute this use case must not be added to this basic permission profile (*least privilege rule*). The first permission of type com.icw.ehf.usermgmt.domain.User refers to the User domain object owned by the eHF Usermgmt module. The second permission which is of type com.icw.ehf.encryption.domain.Object refers to domain objects owned by the eHF Encryption module. This shows a valid and important characteristic of basic permission profiles. Permissions in a basic permission profile may refer to other eHF modules. In other words, a basic permission profile may have dependencies to permissions from other eHF modules, because not all instance permissions within a basic permission profile must refer to the owning eHF module.

We will use the following conventions to name and identify a basic permission profile (see also Figure Figure 67):

- The category of a basic permission profile will be defined as basic.
- To get a unique profile name we will use a namespace convention for eHF modules: `com.icw.ehf.<module-name>.<capability>` where `<module-name>` is the name of the eHF module (for example usermgmt) and `<capability>` refers the accordant capability or use case, which may or may not be the name of a public API method of the eHF module.

Because of the close relation between the public API methods and the basic permission profiles of an eHF module and because basic permission profiles will be exported by an eHF module (see next section), the basic permission profiles can be regarded as part of the public API of an eHF module. For that reason the JavaDoc documentation of the public API methods of an eHF module will be extended with the information which basic profiles must be granted to a principal to be allowed to execute this public API method. See the following Figure Figure 68 for an example.

```
@Public
public interface UserService extends CrudService<User> {

    /**
     * Creates a new User and assigns the given password to this
     * <p>
     * Required basic profile: <code>com.icw.ehf.usermgmt.createU
     * </p>
     * @param user the User to be created
     * @param password the password to be assigned to the new User
     * @return the created User object
     */
    User create(User user, String password);
}
```

Figure 68: Documentation of Basic Permission Profiles in public API JavaDoc.

Using Composite Permission Profile

The basic permission profiles described in this document provide a significant improvement regarding the implementation of an eHF-based application. The application developer doesn't need to handle permissions and permission assignments on the instance permission level any longer. Instead, he/she can define permission assignments by using the basic permission profiles provided by the eHF modules. However, depending on the complexity of a use case or capability, basic permission profiles may still be a to low-level

abstraction for permission assignments. For more complex use cases it might be helpful to have even higher-level permission abstractions. For example, the eHF Usermgnt module provides the capability of creating a person. To create a person object, a depended user object must also be created. That is, the basic permission profile to create a person object not only contains the person specific instance permissions, but also contains all instance permissions to create a user object. In other words, the basic permission profile to create a person object *contains* all instance permissions to create a user object.

To avoid this kind of permission duplication and to make the concept of *containing* permissions explicit, it would be useful to have composable permission profiles and therewith be able to build permission profile hierarchies.

The following Figure [Figure 69](#) shows an example of the definition of a composite permission profile.

```
@Public
public interface UserService extends CrudService<User> {

    /**
     * Creates a new User and assigns the given password to this
     * <p>
     * Required basic profile: <code>com.icw.ehf.usermgnt.createUser</code>
     * </p>
     * @param user the User to be created
     * @param password the password to be assigned to the new User
     * @return the created User object
     */
    User create(User user, String password);
}
```

Figure 69: Example of a composite permission profile

For this to work, the `com.icw.ehf.usermgnt.createUser` basic profile must be defined and loaded before the `com.icw.ehf.usermgnt.createPerson` profile.

15.3.4 Configuration

The usage of eHF Authorization requires a proper JAAS setup and configuration of the access decision voters in Spring. Furthermore, via the bootstrap process you can specify permission profiles that are imported into the database.

Setting up JAAS

To enable eHF-Authorization in your project, you will have to set up JAAS. Most of this setup should be taken care of automatically if you used an assembly template and the Tomcat setup provided by the ICW IDE. In particular, the JAAS `.login` and `.policy` files are installed correctly in this environment. They are located in the `ICW_IDE/tomcat/conf` directory.

The eHF-specific `AccessPolicy` is registered by a servlet listener. It is included in the `web.xml` through a `listeners.fragment` file that is located in the assembly in the directory `src/main/gen/META-INF/merge/assembly/web-xml`:

```
<listener>
    <listener-class>
```

```
com.icw.ehf.authorization.web.ContextListener
</listener-class>
</listener>
```

The original policy will be configured as the next policy of `AccessPolicy`:



Note: Using JAAS for authorization purposes also requires a proper JAAS authentication setup by registering a login configuration. Once this JAAS setup is complete, you must make sure that all further security-relevant calls are made as a privileged, authenticated subject by using:

```
Subject.doAsPrivileged(
    authenticatedSubject, new PrivilegedAction() {
        [...]
    }
);
```

Please see [Authentication](#) on page 153 for more information.

Spring Configuration

During context initialization, an `AccessDecisionVoter` instance is looked up from a Spring web application context by the name `accessDecisionEngine`. Therefore, you have to configure a bean in the assembly spring configuration that inherits from the abstract class `AccessDecisionEngine`. The following example shows the configuration of the default access decision engine `FirstPermitOrDenyDecisionEngine` that holds an instance of the voter `PermissionManagerImpl` that is exported by eHF Authorization via the Spring identification `accessManagerTarget`:

```
<bean id="accessDecisionEngine"
      class="com.icw.ehf.authorization.voter.FirstPermitOrDenyDecisionEngine">
    <property name="voters">
        <list>
            <ref bean="accessManagerTarget" />
        </list>
    </property>
</bean>
```

A further sample configuration can be found in the eHF Assembly configuration file `ehf-accessdecision-context.xml` located in `src/main/config`.

Providing Basic Permission Profile

Permission profiles are defined in XML documents using a specific XML syntax (see the example in Figure [Figure 70](#)). Permission profiles like the one shown in Figure [Figure 70](#) are loaded into the application's database during the bootstrap or import phase of an assembly build. The import is done by the authorization import processor (`com.icw.ehf.authorization.bootstrap.AuthorizationImportProcessor`).

```

<authorization>
    <namedProfiles>

        <namedProfile>
            <profile>
                <permission>
                    <target>
                        <type>com.icw.ehf.usermodel.domain.User</type>
                        <role>*</role>
                        <context>*</context>
                        <identifier>*</identifier>
                    </target>
                    <actions>
                        <action>CREATE</action>
                    </actions>
                </permission>
                <permission>
                    <target>
                        <type>com.icw.ehf.encryption.domain.Object</type>
                        <role>*</role>
                        <context>*</context>
                        <identifier>*</identifier>
                    </target>
                    <actions>
                        <action>CREATE</action>
                    </actions>
                </permission>
            </profile>
            <category>basic</category>
            <name>com.icw.ehf.usermodel.createUser</name>
        </namedProfile>

        <!-- other named Profiles... -->

    </namedProfiles>
</authorization>

```

Figure 70: An example for a basic permission profile

Basic permission profiles must now also be imported into the database during the assembly import phase. However, the basic permission profile files are defined on module level rather

than on assembly level. To enable now the load of the module-level basic permission profiles during the assembly import phase, the basic permission profile files must be made available within the assembly's classpath. The following solution will be used:

- An eHF module provides its basic permission profile(s) at the location <module-folder>\src\main\resources\META-INF\import\profiles. By adding a module to the assembly as a dependency, the basic permission profile files of the module will then be available within the assembly's classpath.
- The assembly imports the permission profiles provided by the eHF modules as described above during the import build phase. To enable this functionality a new authorization import processor instance is added to the assembly's import context. The new import processor is defined within the file <assembly-folder>\src\main\resources\META-INF\ehf-assembly\ehf-import-moduleprofiles.xml. Within this file the new import processor is defined as shown in the following Figure Figure 71. This definition allows the loading of all basic permission profile files which are stored under the location specified above from all modules that are added as a dependency to the assembly.

```
<!-- authorization module profiles import -->
<ehf:processor id="importModuleProfilesAuthorizationImportProcessor"
    ref="authorizationProcessor" order="200">
    <ehf:resourceLocation value="classpath*:META-INF/import/profiles"/>
    <ehf:resourceLoader patterns="*.xml"/>
    <ehf:parameter class="com.icw.ehf.authorization.bootstrap.AuthorizationImportProcessor"
        <property name="importMode" value="OVERWRITE" />
    </ehf:parameter>
</ehf:processor>
```

Figure 71: An example for a basic permission profile

- The new basic profile import processor is included into the import Spring context of the assembly within the file <assembly-folder>\src\main\resources\META-INF\ehf-assembly-import.xml. The code snippet which is doing this is shown in the following Figure Figure 72.

```
<!-- additionally import module specific profiles -->
<import resource="classpath:/META-INF/ehf-assembly/ehf-import-moduleprofiles.xml" />
```

Figure 72: Adding the basic profile import processor to the assembly's import Spring context

Importing Authorization Profiles

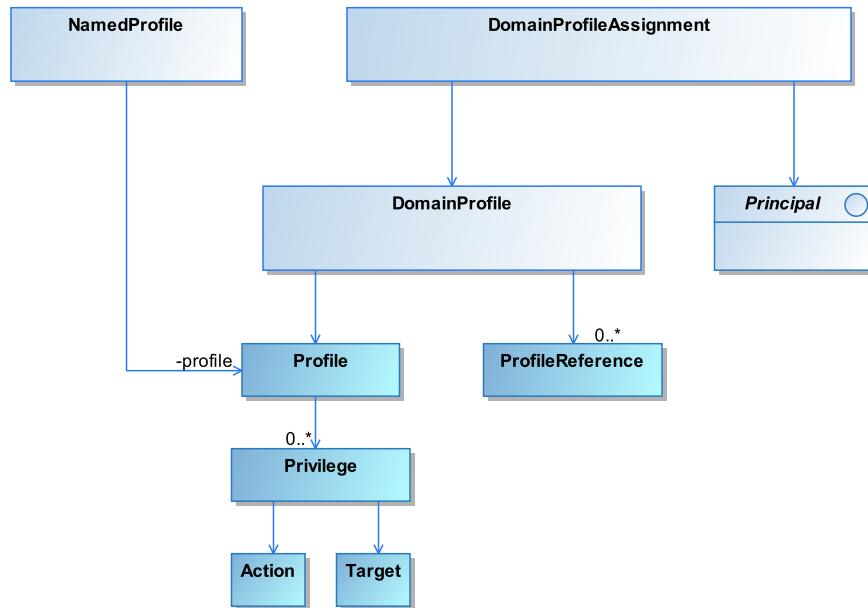
Permission profiles and permission assignments are very often imported during the bootstrap of an assembly. This is typically done by declaring an import processor that is executed during bootstrap or import phases. Usually, the assembly template takes care of this and you will find appropriate declarations in the ehf-bootstrap-authorization.xml and ehf-assembly-import.xml configuration files, respectively. The following XML fragment shows an example of the bootstrap processor for authorization:

```
[...]
<!-- authorization data bootstrap -->
<ehf:processor id="bootstrapAuthorizationImportProcessor"
    ref="authorizationProcessor">
    <ehf:resourceLocation value="classpath:/META-INF/ehf-assembly/bootstrap" />
    <ehf:resourceLoader patterns="authorization-create.xml," />
```

```
authorization-replace.xml,authorization-assignments.xml,  
my-authorization-data.xml" />  
  
<ehf:parameter  
    class="com.icw.ehf.authorization.bootstrap.AuthorizationImportParameter">  
    <property name="importMode" value="OVERWRITE" />  
  </ehf:parameter>  
</ehf:processor>  
[...]
```

You can create your own authorization data in a XML file (for example my-authorization-data.xml) in the META-INF/ehf-assembly/bootstrap or import directories and add the file to the above Spring configuration.

The structure of the XML files corresponds to a large extent with the domain model of eHF Authorization as illustrated in [Figure 73](#) . The XML structure contains two sections: *namedProfiles* and *assignments* . *namedProfiles* are referenced by profile references of the domain model. It provides a category, name, and version that are used as the key by the profile reference. The XML section *assignments* corresponds to the class DomainProfileAssignment .



```

<?xml version="1.0" encoding="UTF-8"?>
<authorization>
  <namedProfiles>
    <namedProfile>
      <profile>
        [...]
      </profile>
      <category/>
      <name/>
      <version/>
    </namedProfile>
  </namedProfiles>

  <assignments>
    <assignment>
      [...]
    </assignment>
  </assignments>
</authorization>
  
```

Figure 73: Profile Assignments

The following list summarizes the most important elements of the authorization declaration file that are not already covered in [Domain Model](#) on page 168 :

AuthorizationData

Represents the extent of data to be imported. It consists of a list of *NamedProfiles* and a list of *Assignments* .

NamedProfile

Defines a *Profile* (including its *Permissions*) that can be uniquely identified by the (given) triple category , name , and version .

Assignment

This is equivalent to a *DomainProfileAssignment* in that it assigns one *DomainProfile* to one *Principal* .

An example authorization data file is shown in the following listings. Reading and producing such files should be straightforward if you keep the authorization model in mind. The below

example defines a named profile (that can later be referenced by a profile reference) with the name *MGR-CDM* in the category *FUNCT* that grants *READ* and *UPDATE* permissions to the domain object *Program*.

```
<?xml version="1.0" encoding="UTF-8"?>
<authorization>
  <namedProfiles>
    <namedProfile>

      <profile>
        <permission>
          <target>
            <type>my.package.Program</type>
            <role>*</role>
            <context>*</context>
            <identifier>*</identifier>
          </target>
          <actions>
            <action>READ</action>
            <action>UPDATE</action>
          </actions>
        </permission>
      </profile>

      <category>FUNCT</category>
      <name>MGR-CDM</name>

    </namedProfile>
  </namedProfiles>
  [...]
</authorization>
```

In the following *assignments* section, a user with the name `my-admin` is granted `AUTH_*` permissions for `Programs` in all domains.

```
<?xml version="1.0" encoding="UTF-8"?>
<authorization>
  <namedProfiles>
    [...]
  </namedProfiles>

  <assignments>
    <assignment>
      <domainProfile>
        <profileReferences />
        <profileStale>false</profileStale>
        <profile>
          <permission>
            <target>
              <type>
                my.package.Program
              </type>
              <role>*</role>
              <context>*</context>
              <identifier>*</identifier>
            </target>
            <actions>
              <action>AUTH_CRUDE</action>
              <action>AUTH_CRUDE_DELEGATE</action>
              <action>AUTH_SEED</action>
              <action>[...]</action>
            </actions>
          </permission>
          <permission>
            [...]
          </permission>
        </profile>
      <domain>
        <id>*</id>
      </domain>
    </assignment>
  </assignments>
</authorization>
```

```

        </domain>
        <restriction>false</restriction>
        <dateRange></dateRange>
    </domainProfile>
    <principal class="user">
        <name>af8d3291-5a66-4bf8-993c-f8a9b5296c48</name>
        <alias>my-admin</alias>
    </principal>
    </assignment>
    [...]
</assignments>
</authorization>
```

Finally, the `my-manager` role is granted the permissions defined in the (FUNCT, MGR-CDM) profile in all domains.

For the `principal` element (near the end of the example below), only `role` and `user` are allowed as values for the `class` attribute. Their `name` tag references their unique ID. The `alias` tag is not evaluated and only makes the file more human-readable.

```

<?xml version="1.0" encoding="UTF-8"?>
<authorization>
    [...]
    <assignments>
        [...]
        <assignment>
            <domainProfile>
                <profileReferences>
                    <profileReference>
                        <category>FUNCT</category>
                        <name>MGR-CDM</name>
                    </profileReference>
                    <profileReference>
                        [...]
                    </profileReference>
                </profileReferences>
                <profileStale>false</profileStale>
                <profile />
                <domain>
                    <id>*</id>
                </domain>
                <restriction>false</restriction>
                <dateRange></dateRange>
            </domainProfile>
            <principal class="role">
                <name>af8d3291-5a66-4bf8-993c-f8a9b5296c37</name>
                <alias>my-manager</alias>
            </principal>
        </assignment>
    </assignments>
</authorization>
```

15.3.5 Extension

The eHF access control framework can be extended or customized to provide own access control models. For example, the eHF Composition module (see [Composition](#) on page 294) makes use of this inherent part of the authorization framework. It extends the instance-based access control by a container-based authorization.

In order to provide a new access control model, you provide a new access decision voter implementation as illustrated in [Figure 74](#). This can be derived either from the interface `AccessDecisionVoter` directly or from the abstract class `AccessDecisionEngine` depending on your requirements. The custom voter implementation has to be declared in the Spring configuration. Furthermore, you add the new voter to the list of voters of the configured access decision engines. Please see the section "Spring Configuration" in the previous chapter [Configuration](#) on page 176.

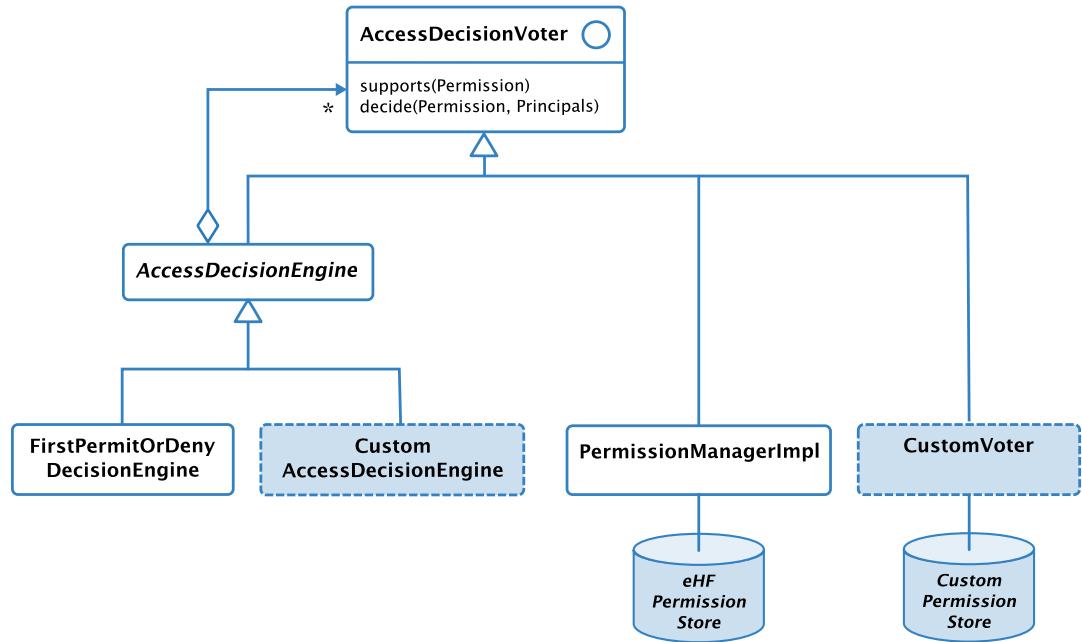


Figure 74: Extension Points for custom Access Decision Voters.

Note:

The specific decision voters can provide their own policy administration point (PAP) to manage and query access permissions including a permission store. However, custom access decision voters can also call module services or reuse the PAP of eHF Authorization.

That way, the container-based access control of the module eHF Composition uses the instance-based permission management of eHF Authorization. The provided `ContainerBasedAccessDecisionVoterImpl` utilizes `PermissionManagerImpl` and the composition service `ContainerService`.

For more information about writing a voter, please consult the document *eHealth Framework - How to implement an Access Decision Voter* (see *ehf-howto-implement-an-access-decision-voter-0.1.pdf* on DVD).

15.3.6 Dependencies

The only important dependency from eHF Authorization has to other eHF-based modules is `ehf-messaging`. It is used for sending messages for auditing purposes (see [Audit](#) on page [184](#)). For example, the administration of permissions creates audit events.

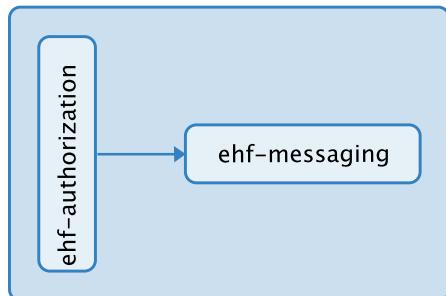


Figure 75: The eHF Authorization module's dependencies.

15.4 Certificate Validation

Certificate validation is a necessary part for all applications dealing with identities based on X.509 certificates. Although a digital certificate contains a validity period, checking this time interval does not suffice to provide reliable system security. Once issued a certificate to an individual, the certificate is valid until the validity interval has expired. To revoke an individual's certificate before the validity period has expired, additional checks are necessary. eHF Certificate Validation provides these services to applications, relying on identity checks, based on digital certificates. Applications can either use the Certificate Validation module or plain JAVA, using Sun JRE 1.5 or higher. The certificate validation based on eHF Certificate Validation will give the application a direct extension point to influence the validation process.

15.4.1 Architectural Design

Validation

An application uses the `CertificateValidatorRegistry` to initiate certificate validation. It can be configured with OCSP and CRL validators. The validators will be added to the registry in the order of their call. The `CertificateValidatorRegistry` implements the *Chain of responsibility* pattern, configured validators are called until one in the chain returns either that the certificate is valid or revoked. If the validator chain is processed until no one else is configured, the result of this validation step is returned.

OCSP Validator

OCSP stands for *Online Certificate Status Protocol* and is defined in RFC-2560. The OCSP validator needs information to request status information about a certificate.

- The certificate for which the status should be obtained.
- A set of trusted certificates to obtain issuer and ocsp responder certificates.
- The url of the ocsp responder.

All these information are provided by components being part of the eHF Certificate Validation module.

CRL Validation

CRL (Certificate Revocation List) is defined in RFC-3280. CRL validation is based on lists containing the information about certificates being revoked before their validity period has expired. To validate against revocation lists the crl validator needs:

- The certificate for which the status should be obtained.
- A set of trusted certificates to obtain issuer and ocsp responder certificates.
- The CRL to use for validation.

eHF-certificate-validation provides an api to obtain the necessary data.

Dependencies

- Spring 2.0
- Apache Commons VFS
- Apache Commons Logging 1.1
- Apache Commons HttpClient 3.0
- Apache Commons Codec 1.3
- Log4J 1.2
- BouncyCastle Crypto Provider jdk1.5-133

Interfaces

- CertificateValidator

Provides a single method returning the validation status of a certificate. How the validation result is retrieved is up to the implementing class. Currently OCSP and CRL validation is supported.

- CrlRepository

An interface intended to hide different strategies to validate CRLs obtained from different sources.

- OCSP-ResponderConfigurer

Provides methods to access information related to OCSP responders and their mapping onto CA certificates. The OcspResponderConfigurer is used to find the OCSP responder URI for a given certificate which is to be validated, as well as the issuer certificate.

Assembly Integration

Certificate validation is ready to use in the eHF assembly. The validation process is kicked by eHF Authentication. It provides authentication interceptors which rule the authentication process. eHF Certificate Validation is configured with default values. These values are set in the following files:

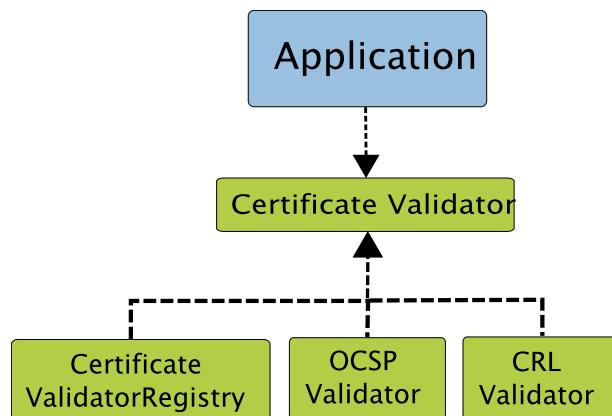


Figure 76: Certificate validation

15.5 Content Scanner

15.5.1 Purpose of eHF Content Scanner

First of all eHF Content Scanner is an API to integrate software like anti virus products into your application. It encapsulates the functionality of these products in its interfaces. It makes your application independent from the actual scanning software used which is usually different depending on your customer's preferences and enables you to use different configurations during installation.

On the other hand eHF Content Scanner already provides some implementations of its own API for specific content scanners (e.g. open source software ClamAV). So you can very quickly build an application with protection against malicious content. eHF Content Scanner has everything on board you need for this.

If your preferred anti virus product is not yet supported, this documentation details all the steps required to implement your own adapter.

15.5.2 Concepts

Clarifies the various problems that eHF Content Scanner aims to address and how these have then been solved.

15.5.2.1 Content Scanner API

The initial goal was to integrate 3rd party anti virus software into the eHF. From the very beginning it was clear that the integration code should enable the usage of different products with completely different characteristics. This led to the decision to design an API rather than an integration component for one or more specific software products.

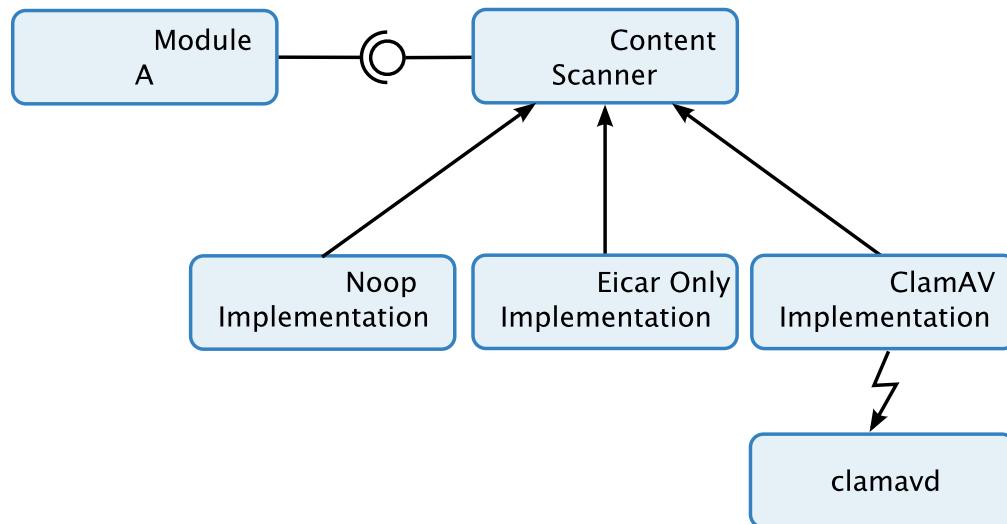


Figure 77: Structure of eHF Content Scanner API

This conveniently decouples the functionality of scanning content from the rest of the application. Different implementations can be used during deployment. Additionally software products not currently supported can easily be added.

15.5.2.2 Streaming Capabilities

The main focus of operation was the deployment in a server environment. Content uploaded by users should be scanned for malicious content. As there can be many users uploading big files, the implementation of such an API needs a low memory footprint. This is best achieved by streaming the content, so we designed this right into the API. Applications which are not running in a server environment also benefit from this approach.

15.5.2.3 Ease of Use

A security feature like scanning for malicious content achieves its goal only when it is actually used and incorporated into the application by the developer. Ease of use is a very important characteristic for an API. First of all the API must be usable with a small number of lines of code. The resulting code must be very clear and explicit, helping everyone to easily understand how everything works.

Another aspect is configuration. It does not help if, as a developer, you need to install and configure numerous components before you can actually use the damn thing. The easiest option is if there is a sensible default behavior. This principle is called "Configuration by Exception". We used this also for the API because there is a default implementation which covers the typical use cases and does not require any configuration.

15.5.2.4 Ready to Use Implementation

eHF Content Scanner also provides an implementation of its own API for the open source ClamAV anti virus software. This is a production ready implementation, which can be used for your server application.

15.5.3 Usage

This section explains how to use eHF Content Scanner in your application.

15.5.3.1 Jump Start

Using the content scanner is, in the simplest cases, as easy as doing the following:

```
InputStream inputStream = some_input_stream;
ContentScanner contentScanner =
    ContentScannerManager.getContentScanner();
try {
    contentScanner.scan(inputStream);
} finally {
    contentScanner.dispose();
}
```

The ContentScannerManager instantiates a ContentScanner using the default configuration file /META-INF/content-scanner-config.xml.

It is necessary to dispose of the ContentScanner instance after you have used it. It is not always necessary for all implementations, but generally most - like the ClamAvRemoteContentScanner - require some form of cleanup such as closing connections or in other cases deleting temporary files. Also to be thread safe it is recommended to call getContentScanner() each time you want to perform a scan and dispose of the instance afterwards. Therefore it is good practice to use a try-finally block for the scan. This way you can execute the cleanup although an exception has been thrown.

15.5.3.2 Content Scanner Manager

The ContentScannerManager is the central entry point of eHF Content Scanner. ContentScannerManager.getContentScanner() reads the configuration from the file /META-INF/content-scanner-config.xml, which must be on the classpath. From this it then returns an instance of ContentScanner with which you can perform your scans.

If the file content-scanner-config.xml is not present in the classpath the default implementation is used. You can also specify your own special configuration file with a different name by calling

```
ContentScanner contentScanner =
```

```
ContentScannerManager.getContentScanner(configFile);
```

which also must be on the classpath. This way you can also switch the configuration at runtime. If you have called `getContentScanner(myConfigFile)` this configuration is used during the succeeding calls of `getContentScanner()`.

With `ContentScannerManager.resetConfiguration()` you can reset the configuration to the original configuration. The next call of `getContentScanner()` then reinitializes the configuration from `content-scanner-config.xml`. You can also change the configuration again by using another `getContentScanner(differentConfigFile)`.

Usually there is no need to call `resetConfiguration()` and `getContentScanner(myConfigFile)` as long as you do not have to change the configuration at runtime. But this method is useful in your tests when you want to use different configurations during your test run.

15.5.3.3 Scanning Methods in ContentScanner

With `ContentScannerManager.getContentScanner()` you get a `ContentScanner` for the actual work. This is an interface defined by the API which each implementation has to fill with life. There are several different methods you can use to scan content:

- `scan(byte[] content)`
- `scan(InputStream input)`
- `scanWhileStreaming(InputStream input)`

Each one has its own characteristics and should be used for different scenarios.

Scanning an Array of Bytes

The method `scan(byte[] content)` takes an array of bytes as an argument and passes it to the scanner software. This is the simplest of the scanning methods and easy to use. But it is only suitable for relatively small amounts of data. Using a byte array means to instantiate all the content in memory. Loading a document of 10MB in a byte array occupies 10MB of your available memory. If your application uses multiple threads (as in a server environment) you can very easily run out of memory.

Here is an example how this method is used:

```
ContentScanner contentScanner =
    ContentScannerManager.getContentScanner();
byte[] bytes = "This is no virus:-)".getBytes();
try {
    contentScanner.scan(bytes);
} finally {
    contentScanner.dispose();
}
```

Scanning an InputStream

In comparison to `scan(byte[] content)` the method `scan(InputStream input)` has a lower memory footprint. The `ContentScanner` reads the data piece by piece from the stream and sends it to the scanning software where it is checked. After all data is read, it closes the `InputStream` and checks the result of the scanner software. The content does not fill up your memory. This method is useful if you want to scan content that you have already persisted (e.g. in a temporary file or in your database).

Here you can see how you can check an `InputStream`:

```
ContentScanner contentScanner =
    ContentScannerManager.getContentScanner();
InputStream inputStream = some_input_stream;
try {
```

```

        contentScanner.scan(inputStream);
    } finally {
        contentScanner.dispose();
}

```

Unfortunately an InputStream can only be used once. So this method is not suitable for situations when your input is an InputStream and you have no access to the source of the stream. For example when you are reading content from a socket during streaming over the Internet. This restriction is overcome by the last of the three scanning methods.

Scanning an InputStream while streaming

This methods takes an InputStream as an argument and returns an InputStream. Now with this InputStream you can do whatever your application is supposed to do. While your application is reading the data from the stream it is simultaneously scanned. The stream is consumed only once:

```

ContentScanner contentScanner =
    ContentScannerManager.getContentScanner();
InputStream inputStream = some_input_stream;
try {
    inputStream =
        contentScanner.scanWhileStreaming(inputStream);

    // read here from your inputStream

    inputStream.close();
} finally {
    contentScanner.dispose();
    IOUtils.closeQuietly(inputStream);
}

```

When the stream is closed by your application, the ContentScanner transparently checks the result of the scanner software.

All three methods mentioned above throw a PotentialMaliciousContentException if a virus or other malicious content is detected or a ContentScannerNotAvailableException if the anti virus software is not available or e.g. the remote scanning host is not reachable.

15.5.3.4 Other Methods in ContentScanner

Checking Availability of a Scanner

The method `isContentScannerAvailable()` is especially useful when you are using an implementation which communicates with a remote host or if a specific single scanning process is running on your machine. It determines if the communication can be established and the anti virus software is accepting content for scanning.

The implementation of this method may vary depending on the type and character of the anti virus software being used. An implementation that uses file based scanning on your local hard disk might always return `true`.

Date of Last Update

The method `getDateOfLastUpdate()` returns the date when the virus definitions of the anti virus software were last updated. If a file has been scanned recently you can use this method to determine if it should be scanned again. If the definitions have not been updated since the last scan, there is no need to scan it again.

Disposing

The method `dispose()` is used for cleanup after the scan process has finished. Once again it depends on the specific implementation as to what the behavior actually is. Some may need to remove temporary files, others may need to close connections to remote machines while some others will do nothing at all. But as mentioned above it is necessary

to call this method after you have finished the actual scan. You can never know which implementation of the ContentScanner is configured at runtime. And this implementation might require some cleanup after the scan process. If you do not call `dispose()` this required cleanup routine of the implementation in use is not executed. Better to be on the save side and always call `dispose()`. This way your code is free of any assumptions on the behavior of the used implementation and you can configure another one any time you like.

The method is usually called in a `finally` block. This way you ensure it is called whether an exception has been thrown or not.

15.5.3.5 Testing Your Application

In order to test your application you will have to feed it some malicious content - or better yet something that claims to be malicious content but really is not. For this you can use the Eicar test virus. Unfortunately handling these files is tricky when you have anti virus software installed locally on your machine. Usually you can not download these files without deactivating the anti virus software, because it jumps right in and warns you. It is supposed to do that.

Therefore eHF Content Scanner provides the utility class `EncryptedFileUtils` which eases the trouble with test virus files. Here is an example how it is used:

```
ContentScanner contentScanner = ContentScannerManager.getContentScanner();
InputStream inputStream = EncryptedFileUtils.
    decryptResourceWithAes256( "/eicar.com.aes" , " /eicar.com.aes.key" );
try {
    contentScanner.scan(inputStream);
    fail("PotentialMaliciousContentException should have been thrown!");
} catch ( PotentialMaliciousContentException e ) {
    assertNotNull(e.getMessage());
    assertNotNull(e.getTypeOfMaliciousContent());
} finally {
    contentScanner.dispose();
}
```

This piece of code is an example from our tests. It creates an `InputStream` that streams directly the Eicar test virus from an AES encrypted file. This way the test virus passes the common anti virus software. You can use this in a similar way to test your application without any troubles.

15.5.3.6 Dependencies

The following is a list of runtime dependencies required by eHF Content Scanner:

3rd party library	Version
aspectjrt	1.5.3
commons-collections	3.2
commons-configuration	1.5
commons-io	1.4
commons-lang	2.3
commons-logging	1.1.1
commons-net	1.4.1
junit	4.4
log4j	1.2.15

15.5.4 Configuration of eHF Content Scanner

This section describes the configuration of eHF Content Scanner and its various implementations.

The content scanner is usable without any configuration at all. It uses a default configuration which is the EicarOnlyContentScanner. This is the most sensible configuration during development.

You need to add the file /META-INF/content-scanner-config.xml to your classpath with the appropriate configuration in order to use other implementations of the content scanner API. Syntax and options of the different implementations are the topics of the following chapters.

15.5.4.1 Configuration of the EicarOnlyContentScanner

This section describes the configuration of eHF Content Scanner with the EicarOnlyContentScanner.

The EicarOnlyContentScanner is not intended for production use. It is a scanner especially for development and testing, because it only detects the [Eicar test signatures](#)). The Eicar test signatures are files that are recognized by all virus scanner products as malicious content, but without carrying any damage or replication routines in the code. The intention behind these test signatures is to have a safe way to test your virus scanner products. This is useful in your development environment if you want to test the applications behavior when a virus is detected and you do not want to use a real virus for this purpose.

The EicarOnlyContentScanner is an in memory scanner. It can be easily configured and does not require installation of separate scanner software. Actually this scanner is the default in ehf-content-scanner if no other configuration is in place. It logs a warning message on each attempt to scan content reminding you that it should not be active in a production environment.

Here is an example of a configuration file for the EicarOnlyContentScanner:

```
<?xml version="1.0" encoding="UTF-8" ?>
<contentscanner>
    <factory>com.icw.ehf.contentscanner.eicaronly.EicarOnlyContentScannerFactory</factory>
    <logEnabled>false</logEnabled>
</contentscanner>
```

By default the EicarOnlyContentScanner logs a message on each scan invoked to indicate that using it is not a good choice in production. With the logEnabled property you can influence this. When it is set to false the logging of these warning messages is suppressed.

15.5.4.2 Configuration of the NoopContentScanner

This section describes the configuration of eHF Content Scanner with the NoopContentScanner.

The NoopContentScanner is a implementation of the content scanner API which does absolutely nothing - not even logging a warning. It is intended for scenarios where scanning for viruses is not intended.

On first glance this might appear a bit useless. But usually you have applications that have the usage of the content scanner built in. How can you turn off virus scanning at runtime? Change the source code? Not a good idea. Instead you can configure the NoopContentScanner for this purpose.

The content of the content-scanner-config.xml file for the NoopContentScanner is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<contentscanner>
    <factory>com.icw.ehf.contentscanner.noop.NoopContentScannerFactory</factory>
</contentscanner>
```

15.5.4.3 Configuration of ClamAvRemoteContentScanner

This section describes the configuration of eHF Content Scanner with a remote ClamAV scanning host.

First of all you need at least one host on which the ClamAV (clamavd) daemon is installed, running and accepting connections on a specific port. It does not matter if it is a single host, a group of hosts, the local host or a remote system. The content scanner uses a TCP connection to communicate with the ClamAV daemon. For detailed instructions on installing and configuring ClamAV see <http://www.clamav.net/>.

For your own application you must add a `/META-INF/content-scanner-config.xml` file to your classpath. It should have the following structure:

```
<?xml version="1.0" encoding="UTF-8" ?>
<contentscanner>
    <factory>com.icw.ehf.contentscanner.clamav.ClamAvRemoteContentScannerFactory</factory>
    <hostlist>
        <host>
            <!-- host1 -->
            <name>hostname_1</name>
            <port>port_number_1</port>
        </host>
        <host>
            <!-- host2 -->
            <name>hostname_2</name>
            <port>port_number_2</port>
        </host>
        [...]
        <host>
            <!-- hostN -->
            <name>hostname_N</name>
            <port>port_number_N</port>
        </host>
    </hostlist>
</contentscanner>
```

The host selection process for a particular content scanner instance is as follows: first a host is randomly selected from the list in order. Then the health status of the selected host is verified with a ClamAV PING status command. If the selected host does not respond within the timeout specified by the parameter `ping-timeout`, the next host in line is tried until all hosts from the list have been processed. The first host that answers positively to a ClamAV PING status command is used.

Configuration parameters:

1. The `timeout` parameter sets the timeout (in milliseconds) for a particular connection to a remote ClamAV daemon. The default value is 10 seconds (10000). This parameter can be set for all hosts globally and/or per host. The individual host setting overrides the global setting.

```
<?xml version="1.0" encoding="UTF-8" ?>
<contentscanner>
    <factory>com.icw.ehf.contentscanner.clamav.
ClamAvRemoteContentScannerFactory</factory>
    <timeout>5000</timeout>
```

```

<hostlist>
    <host>
        <name>hostname_1</name>
        <port>port_number_1</port>
        <timeout>10000</timeout>
    </host>
    [...]
</hostlist>
</contentscanner>

```

2. The *ping-timeout* parameter sets the timeout (in milliseconds) used to check the availability of a remote ClamAV daemon. The default value is 3 seconds (3000). This parameter can only be set globally for all hosts.

```

<?xml version="1.0" encoding="UTF-8" ?>
<contentscanner>
    <factory>com.icw.ehf.contentscanner.clamav.
ClamAvRemoteContentScannerFactory</factory>
    <ping-timeout>15000</ping-timeout>
    <hostlist>
        [...]
    </hostlist>
</contentscanner>

```

3. The *keepalive* parameter (*true/false*) sets whether a particular content scanner keeps its connection to a remote ClamAV daemon open in between multiple scans. The default value is *false*. The parameter can be set for all hosts and/or per host. The individual host setting overrides the global setting.

```

<?xml version="1.0" encoding="UTF-8" ?>
<contentscanner>
    <factory>com.icw.ehf.contentscanner.clamav.
ClamAvRemoteContentScannerFactory</factory>
    <keepalive>false</ping-timeout>
    <hostlist>
        <host>
            <name>hostname_1</name>
            <port>port_number_1</port>
            <keepalive>true</keepalive>
        </host>
        [...]
    </hostlist>
</contentscanner>

```

In a cluster environment we recommend that you install the ClamAV on each cluster node, and add `localhost` as the first host entry in the list followed by the other cluster nodes.



Note: There are several configuration options for the clamavd which influence the behavior of the TCP communication between the ClamAvRemoteContentScanner and the daemon. For example:

- `TCPSocket`
- `TCPAddr`
- `MaxConnectionQueueLength`
- `StreamMaxLength`

You can usually find these options in a file named `clamav.conf` on the host where the daemon is installed. The default values for these options of ClamAV may be not sufficient for your requirements. You or your administrator needs to review and edit the file carefully. Refer to the documentation of `clamavd` for more information.

15.6 Security Token Service

Overview

Usually, client systems convey user credentials on whose behalf they act to every single endpoint. In case of intermediate systems like an Enterprise Service Bus (ESB), or any other middleware systems, every intermediary has to lead-trough the credentials to the endpoint and therewith the knowledge about the user secrets as well. Endpoints themselves authenticate the user and after successful authentication the actual "business work" is performed.

The basic idea of the eHF Security Token Service is "Security as a Service". In the broadest sense, this means: Instead of implementing various security mechanisms into each service and service consumer - with the consequence, that the user credentials have to be distributed in the whole infrastructure - the security enforcement burden is shifted to a shared security service. To achieve that, the eHF Security Token Service module exposes a Security Token Service. A protocol for the Security Token Service (STS) is defined in an OASIS specification named WS-Trust. The implementation of the eHF Security Token Service is based on this WS-Trust specification.

After integrating the STS in the system environment, the aspect of authenticating users on base of their credentials is centralized to that single service. The Security Token Service itself creates so-called assertions after successful authentication. The assertion or token – we use these terms synonymously – may be transported to endpoints directly or via intermediaries without unveiling the actual credentials of the user. As the Security Assertion Markup Language (SAML, also specified by OASIS) has become the definitive standard for the exchange of authentication and authorization information and Web Single Sign On (SSO) solutions in the enterprise identity management, we use SAML as format for the Assertions.

The typical lifecycle of a service call using a Security Token consists of the following four steps:

1. The client requests a token from the STS
2. The STS authenticates the user and responds with a security token in the form of a SAML assertion (in case of successful authentication of the user)
3. The client invokes the business service. The token is also conveyed to the service with the client's request (currently in the HTTP Header)
4. The business service validates the token and executes the business logic on behalf of the user identified by the security token.

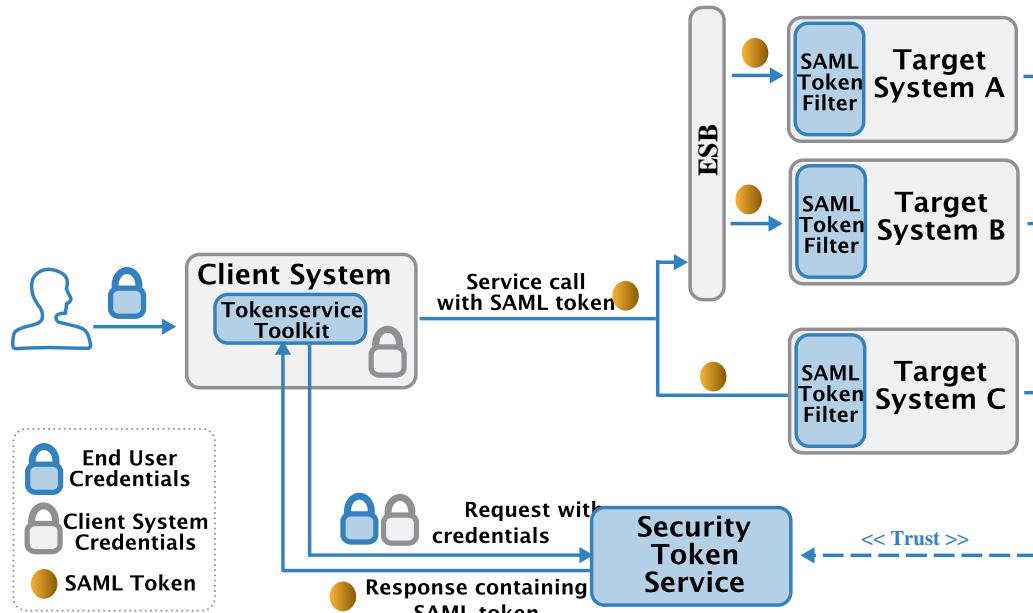


Figure 78: Lifecycle of a service call with tokens

As you can see in the picture above, the eHF Security Token Service solution consists of several components:

1. The Security Token Service, which implements a WS-Trust compliant STS. It issues and renews tokens for provided user credentials.
2. The eHF Tokensevice Toolkit, a convenient Java-based library which encapsulates the WS-Trust communication between client systems and the Security Token Service.
3. The SAML Token Filter resides at all target systems which provide the business services. This filter is able to validate requests which carry a previously issued security token.

Issuing security tokens for users is currently done based on their credentials which proof the user's identity. The eHF Security Token Service supports username/password pairs and X.509 client certificates as valid credentials. Additionally, previously issued tokens can be renewed (Token Renewal) and tokens issued by another trusted STS can be re-issued (Token Exchange). The entity that invokes the Security Token Service is usually a system, acting on behalf of the actual user. That system also authenticates at the STS, but in an additional authentication layer and with a technical user identity. This authentication is performed on the SSL transport layer via x.509 client certificates. Furthermore the eHF Security Token Service module provides a token filter, used at target systems which expose service endpoints accepting requests with security tokens. The token filter is a SAML login filter which validates client requests, containing security tokens in their header, to verify that the requesting user has been authenticated by a trusted Security Token Service. The SAML login filter is a servlet filter that processes SAML tokens in HTTP requests for authentication. It extracts the token from the HTTP request, decodes and validates it. After successful validation, the contained user identifier is used for authentication. The filter reuses the existing eHF Authentication module, including its support for different user management backends.

15.6.1 Architectural Design

On the server side, the eHF Security Token Service consists of the two components STS and SAML login filter. The first one issues tokens whereas the latter is used in

target systems to authenticate using SAML tokens. The server-side components are complemented by the eHF Token Service Toolkit which allows easy communication with the STS for client systems.

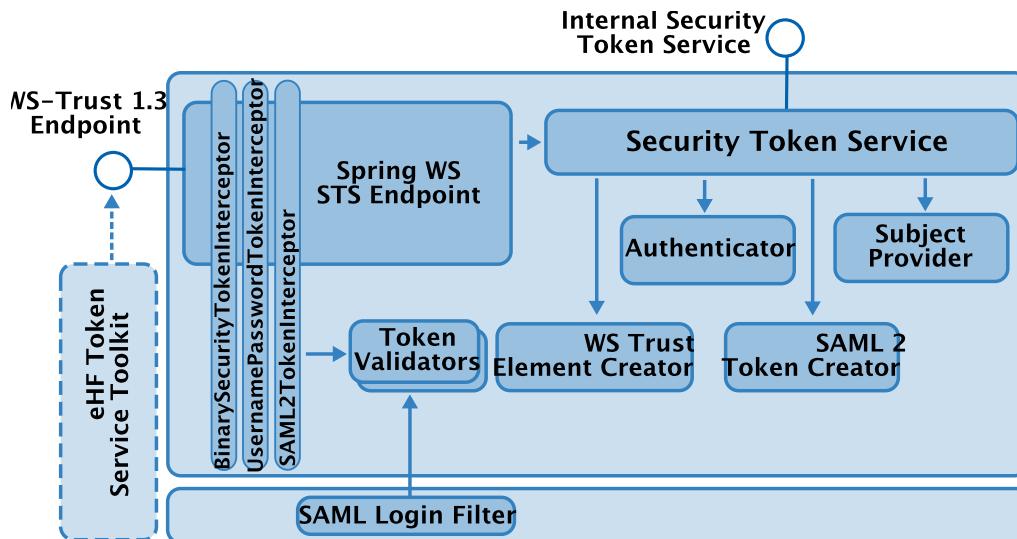


Figure 79: Security Token Service Architecture

Service Interfaces

The eHF Security Token Service is realized as a Web Service that conforms to the WS-Trust specification. In contrast to other eHF platform Web Services, it is not realized with the Axis 1.4 framework but with the [Spring Web Services](#) framework. The eHF Security Token Service Web Service conforms to the WS-Trust 1.3 specification of OASIS. The specification defines an XML schema which makes use of some additional schemas from other WS-* specifications such as WS-Security, WS-Policy and WS-Addressing. This results in quite a complex WSDL definition.

The eHF Security Token Service supports a defined subset of the functionality that is possible with WS-Trust. Java-based clients are recommended to use the eHF Token Service Toolkit for STS communication so they are shielded from the actual Web Service interface definition and WS-Trust interaction complexity. Since the STS interface is WS-Trust compliant, any client system is able to interact with the STS without the assistance of the eHF Token Service Toolkit. The WSDL of the eHF Security Token Service is available at:

```
http://<host>/<servlet-context>/tokensevice/ws-trust.wsdl
```

However, this WSDL is hardly sufficient for access with an automatically generated client. For once the STS does not fully implement all the WSDL. Second, the client system must additionally authenticate with HTTP Basic or a client certificate. Third the service also expects authentication of the user for whom the token shall be issued via a WS-Security "UsernameToken" which also has to be added to the request.

Spring Web Services exposes the endpoint of the STS Web Service via a dispatcher servlet that is mapped to the URL pattern /tokensevice/*. The servlet is defined in the eHF Security Token Service module and contributed to an eHF assembly.

Beside the WS-Trust interface, the eHF Security Token Service exposes an internal interface which can be used by other eHF modules within the application where the STS is embedded. The service com.icw.ehf.tokensevice.service.InternalSecurityTokenService also provides methods to issue and renew tokens.

The internal STS directly operates with SAML assertions. WS-Trust compliant message exchange is not required since the internal STS can only be called within system boundaries without any network communication.

Token Interceptors

The eHF Security Token Service uses Spring WS to expose the WS-Trust endpoint. Communication in WS-Trust is done exclusively by sending Request Security Token (RST) requests to the STS which get answered with Request Security Token Responses (RSTR). The token subject, the user for whom a token gets issued, is carried as WS-Security header information inside an RST. The token interceptors process those WS-Security header and extracts the end user's credentials. The credentials, either username/password, X.509 certificate or a SAML assertion, get placed inside the message context in order to be accessible for the Security Token Service Implementation.

Token Validators

One motivation behind using security tokens is a scenario where there exist potentially untrustworthy intermediaries who have access to the message and can misuse the credentials that are sent along with it. Using a security token instead of user credentials significantly reduces the risk and consequences of such an exposure – not least because extra care is taken to reduce the potential of token misuse by an unauthorized party.

Therefore, each token is validated in multiple ways in order to make sure that the token receiver can trust it. Only if all validations succeed, the token is considered valid and further processed. The validations are performed in the order of ascending cost and risk. The validations in their order of occurrence are:

1. Freshness validation – make sure the token is not yet too old according to the configured maximum age setting and the expiration date within the token itself.
2. Audience validation – make sure the receiver is a member of the intended audience that is set in the token.
3. Replay detection – detects the repeated reception of the same token.
4. Signature validation – makes sure the signature is valid, confirming the integrity and the authenticity of the issuer of the token.

The appliance of these combined validations ensures a good protection against attacks such as usage of fake tokens and especially the reuse of an obtained and possibly modified token.

Token Issuing

Based on an incoming RST, a RSTR response is created by the eHF STS which contains a valid SAML2 assertion. The construction of WS-Trust elements and SAML2 assertions is performed by the WS Trust Element Creator and SAML2 Token Creator components. Before issuing a token, the subject's identity has to be verified through user authentication at some user store.

This end user authentication is encapsulated in the eHF `com.icw.ehf.tokenservice.authentication.Authenticator`. Each implementation adapts to a different user store. This interface is an extension point which allows you to adapt the eHF Security Token Service to a custom user store.

The internal Security Token Service interface issues tokens for currently authenticated users. If the STS is embedded into an eHF application, this application is able to issue tokens for authenticated users by leveraging the internal interface. The authenticated user respectively subject gets determined by a `com.icw.ehf.tokenservice.service.SubjectProvider` implementation.

Deployment options

In an integrated solution, the eHF Security Token Service typically gets deployed as a stand-alone application as centralized credentials store. Nevertheless, it is possible and sometimes useful to embed the eHF STS into an existing eHF application. In this deployment scenario, this eHF application acts as the central credentials store in which all other participating systems have to trust.

Supported WS Security Profiles

The WS-Trust specification requires that the user credentials are carried as WS Security Extension (WSSE) headers. The eHF Security Token Service can process username/password, X.509 certificates and SAML2 assertions as end user credentials.

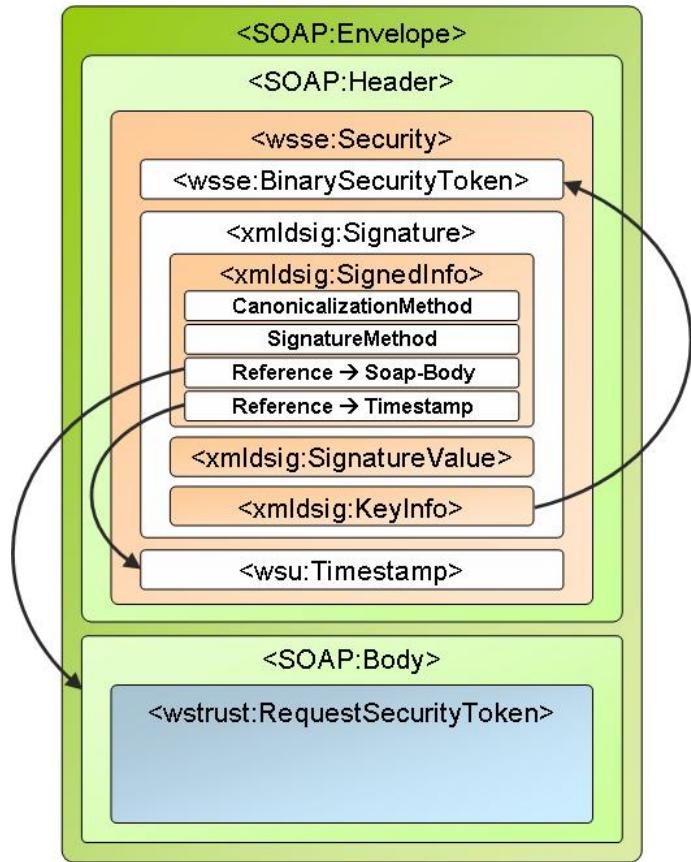
Username and password have to be transmitted in the WS Security Extension header as WSSE username token.

```
<wsse:Security>
  <wsse:UsernameToken>
    <wsse:Username>username</wsse:Username>
    <wsse:Password>password</wsse:Password>
  </wsse:UsernameToken>
</wsse:Security>
```

X.509 client certificates have to be transmitted as WSSE binary security token together with a digital signature.

```
<wsse:Security>
  <wsse:BinarySecurityToken>
    MIIDCTCC...
  </wsse:BinarySecurityToken>
  <ds:Signature>
    ...
  </ds:Signature>
</wsse:Security>
```

The implementation to authenticate end user identities based on their X.509 certificates conforms to the WS Security Specification. This means, that the certificate is contained in an `<BinarySecurityToken>` element that belongs to the private key used to sign the message. The recipient of the message – the Security Token Service – validates the signature based on the algorithms specified in the signature's `<SignatureMethod>` element. The `<SignedInfo>` element also contains `<Reference>` elements that are pointers to the message parts covered by the signature. Note that the signature is an indirect signature. The `<SignatureValue>` element contains the Base64 encoded signature of the digest of the canonicalized `<SignedInfo>` element, whereas the `<SignedInfo>` element contains the digital fingerprints (the digests) of the data actually secured by the signature. See the following schema for the structure of the request message.

**Figure 80:** X.509 Client Certificate RST

Compared to a WS Security Header with a UsernameToken, a BinarySecurityToken and digital signature - as proof of ownership - are much more complex. The complexity to create and more than that to process such a security header is pretty high. This is the reason why the eHF Token Service Token Service module uses an existing WS Security Engine. The [WSS4J API](#) is internally used to process the security header. Apache WSS4J is an implementation of the OASIS Web Services Security Framework (WS-Security). WSS4J is a Java library that can be used to sign and verify SOAP Messages with WS-Security information. WSS4J will use Apache Axis and Apache XML-Security projects and is interoperable with JAX-RPC based server/clients and .NET server/clients. We use WSS4J to verify the security header on WS-Trust RST Messages (Request Security Token). The verification is done by an interceptor extracting the credentials and depositing them in the message context when the processing of the BinarySecurityToken/XML Signature was successful. The application flow for issuing a token based on username/password or X.509 certificate (BinarySecurityToken) is the same, with the difference that each variant is handled by its own interceptor.

Even SAML assertions issued by another STS can be used as credentials to issue new SAML assertions. In this case, the eHF Security Token Service has to trust the issuing STS in order to return a token. The complete SAML token as credential has to be transported in the WSSE header as well:

```
<wsse:Security>
  <saml:Assertion>
    ...
  </saml:Assertion>
```

```
</wsse:Security>
```

The eHF Security Token Service will iterate through all registered token validators to validate the token before the STS issuing is triggered.

If you use the eHF Token Service Toolkit for the communication with the eHF STS, the toolkit will abstract from all low-level WS-* parts, so you as a client developer do not have to care about the different authentication schemes or any WS-Trust or SAML elements.

Token Encoding Format

The WS-Security specification defines a standard way to secure Web Service invocations that defines how to put a security token inside a SOAP header. However, the eHF Security Token Service does not yet make use of this standard mechanism because it would limit to standard SOAP Web Services and lock down other transport protocols (e.g. REST-style services). Instead, the SAML Token is transmitted as the value of the custom HTTP header X-ICW-SAML. The value is created by compressing and Base64-encoding the SAML token XML document. An HTTP header with a token then looks like this:

```
GET /protected.html HTTP/1.1
Host: service.icw.com
...
[additional headers]
...
X-ICW-SAML: hVLLbsTwEPyV...
```

In spite of compression the token remains quite large. 1200 characters would be a common size for a signed and encoded SAML token.

15.6.2 Usage

To use the eHF Security Token Service, it is recommended to use the eHF Token Service Toolkit which comes with the eHF Token Service Toolkit module. It provides a Java-based API to obtain and renew SAML tokens. `com.icw.ehf.tokenservice.webapi.client.SecurityTokenServiceClient` represents the client which is responsible for communicating with the STS. It provides several methods for accepting enduser credentials and return valid SAML tokens. The network communication is performed by the toolkit and all interaction with the STS is totally transparent to client systems.

As tokens are defined in XML, the `com.icw.ehf.tokenservice.Saml2Token` interface is a wrapper which provides convenient methods for accessing various fields of a SAML token. There is intentionally no API to manipulate the token, as even one toggled bit may invalidate the token's digital signature, which in turn would lead to an unusable token. SAML tokens are usually passed along to the protected resource and no further processing of the obtained Token takes place at the client. For this reason, the interface provides the method `toCompressedBase64String()` to create the encoded form of the token that is expected at the token filter.

15.6.2.1 STS client configuration

The eHF Security Token Service issues SAML tokens for endusers. But for obtaining SAML tokens from an STS, the client systems themselves need to authenticate at the STS. The eHF Security Token Service supports the following three authentication schemes: Currently there are three different variants to secure the communication between the client system and the STS. All variants make use of established HTTP-based security mechanisms. In every case, the enduser's credentials are nested in a WS-Security-Element (wsse) in the SOAP header of the request, which in turn is nested into an HTTP header. The three variants differ in the way the client system authenticates itself at the STS and if transport layer security (TLS) is added. Every variant could be used with either the pre-configured Spring-managed client or with a programmatically configured client.

- **HTTP Basic Authentication:** the client system uses HTTP Basic authentication without any encryption. This mode is intended to be used in test and development

- environments only. The unencrypted user credentials can easily be eavesdropped and decoded by an attacker.
- Transport Level Security (TLS) with server authentication (server certificate only): In this scenario, the client gets connected to the server based on an SSL connection using server authentication. This ensures confidentiality through encryption at least. Here, only the server authenticates while the server accepts any client on the transport layer. The missing authentication of the client system is performed via HTTP Basic Authentication. After the establishment of the SSL connection the client system's technical user credentials will be transmitted securely via HTTP Basic Authentication to the STS. This variant represents the minimal security level that should be used to obtain tokens, as neither the credentials of the technical user nor the credentials of the enduser can be intercepted by a third party.
 - Transport Level Security (TLS) with mutual authentication (server and client certificate): This variant provides the highest applicable level of security on the transport layer at the moment. The client system connects to the server using SSL and is authenticating with its client certificate. This ensures encryption on the transport layer and authenticates the client system to the STS based on the client's X.509 certificate. To establish an SSL connection using mutual authentication, the client system certificate (the whole key pair, which means public and private key) must be accessible at the client side. Analogous to the second variant, the trusted server's public key must reside in a truststore. In contrast to variant 2, the locally configured key- and truststore always overrides already configured system-wide key- and truststore.

Furthermore, the URL of the Security Token Service has to be configured at client side.

15.6.2.2 Setting up an STS client

The `com.icw.ehf.tokensevice.webapi.client.SecurityTokenServiceClientFactory` facilitates the creation of a `SecurityTokenServiceClient`. An instance of the `SecurityTokenServiceClient` can be configured via a Spring context definition and a separate property file or just programmatically with the help of a `SecurityTokenServiceClientFactory`. You might use the Spring-configured way, if the configuration of the client instance is rather static and you need just one client instance with a stable configuration. If you need more instances, maybe with separate configurations at the same time, you might want to set up your client instances with the help of a `SecurityTokenServiceClientFactory`. Furthermore, the programmatically configured client provides mechanisms to configure a proxy server – if desired with separate authentication – all requests for tokens are routed through. In the following, we'll give a short overview about both approaches.

15.6.2.2.1 Spring-configured client:

The eHF Token Service Toolkit module already comes with a pre-configured `SecurityTokenServiceClient` with bean name `stsClient` in file `ehf-tokensevice-ws-client-context.xml`. The configuration file `stsclient.properties` controls all options of the eHF Token Service Toolkit client.

Client with HTTP Basic Authentication

To use HTTP Basic authentication for communication with the STS requires the following configuration within the `stsclient.properties` file:

```
#HTTP BASIC AUTHENTICATION
stsclient.uri=http://127.0.0.1/ehf/tokensevice/
=====
# Credentials for the BASIC AUTHENTICATION
#
stsclient.basicauth.username=stsclient
```

```
stsclient.basicauth.password=stsclient
```

By setting the desired protocol scheme `http` as part of the URL in property `stsclient.uri`, the HTTP Basic authentication scheme is used. The URI describes the location of the STS. `stsclient.basicauth.username` and `stsclient.basicauth.password` represent the client system's credentials which are used to authenticate at the eHF Security Token Service.

Client with TLS and HTTP Basic Authentication

Here, the client system as well authenticates with HTTP Basic authentication, but the connection is secured via SSL. The following configuration is required:

```
#HTTPS with server authentication
stsclient.uri=https://127.0.0.1/ehf/tokenservice/
=====
# Credentials for the BASIC AUTHENTICATION #
=====
stsclient.basicauth.username=stsclient
stsclient.basicauth.password=stsclient
=====
# public key from the trusted STS #
=====
stsclient.ssl.truststore=classpath:/META-INF/stsclient_truststore.jks
stsclient.ssl.truststorePass=changeit
```

Here, you have to specify the protocol scheme `https` to establish an SSL connection between client and STS. Again, the properties `stsclient.basicauth.username` and `stsclient.basicauth.password` represent the client system's credentials for authentication at the STS. Because we use an SSL connection which incorporates the STS's certificate, the client system has to trust the STS. This trust relation is realized through a Java truststore, which contains the STS certificate. `stsclient.ssl.truststore` holds the location of the truststore file whereas `stsclient.ssl.truststorePass` represents the password for accessing the truststore.

Client with Mutual TLS Authentication

To communicate over an SSL channel with mutual authentication, the following configuration is required on the client side:

```
#HTTPS with mutual authentication
stsclient.uri=https2way://127.0.0.1/ehf/tokenservice/
=====
# public key from the trusted STS #
=====
stsclient.ssl.truststore=classpath:/META-INF/stsclient_truststore.jks
stsclient.ssl.truststorePass=changeit
=====
# client certificate #
=====
stsclient.ssl.keystore=classpath:/META-INF/stsclient_keystore.jks
#
# password to access the keystore
stsclient.ssl.keystorePass=changeit
#
# password to access the key itself
stsclient.ssl.keyPass=changeit
```

To indicate mutual authentication on transport level, the protocol prefix `https2way` has to be used. In this scenario, both client system and STS provide their certificates in the SSL handshake. On the client side, we again have to trust the STS by referencing the appropriate Java truststore through the properties `stsclient.ssl.truststore` and `stsclient.ssl.truststorePass`. Additionally, the client system has to mix in the

private key of its certificate. The private key has to be maintained in a Java keystore file which is referenced by property `stsclient.ssl.keystore`. The keystore typically is secured by a password, which has to be provided in property `stsclient.ssl.keystorePass`. Access to private keys inside a keystore is secured by an additional password. To let the Token Service Toolkit access the private key, the key password has to be defined in property `stsclient.ssl.keyPass`.

A request to an eHF Security Token Service can be done through this pre-configured Spring bean:

```
// setting up the appropriate spring ctx, configuring the client
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "classpath:/META-INF/ehf-tokenservice-ws-client-context.xml");

// this is the client interface
SecurityTokenServiceClient stsClient =
    (SecurityTokenServiceClient) ctx.getBean("stsClient");

// The Tokens are returned by the client in case of successful
// authentication of the end-users credentials
Saml2Token samlToken = stsClient.requestSamlToken("user1", "user1");
```

15.6.2.2.2 Programmatically configured client:

The factory approach helps to create `SecurityTokenServiceClient` instances programmatically without using a configuration file and should be used when more dynamic is needed, or when the communication to the STS should go through a proxy server. This approach results in slightly more client code than the Spring alternative. The configuration of the clients which are created by the factory will be retrieved from a mandatory `com.icw.ehf.tokenservice.webapi.client.SecurityTokenServiceClientConfiguration` object. Clients created with this factory are 'protocol-aware', which means that depending on the supported protocol contained in the `SecurityTokenServiceClientConfiguration`, the factory is setting up the client instance exclusively for this protocol. In contrast to the Spring-based way, the created clients are not pre-configured with the URL of the Security Token Service, so the client system must set the service URL at the client instance by invoking `SecurityTokenServiceClient.setServiceUrl(String)`. The URL may be altered between two STS requests. When not only the URL but also the proxy to use or even the protocol of the Security Token Service changes, a new instance of the client must be constructed every time.

The following code illustrates, how a client configuration is created programmatically for transport level security with mutual authentication. The resulting STS client instance contains the same configuration as the Spring-configured alternative above:

```
// configuration container object for 'https' with server auth
SecurityTokenServiceClientConfiguration config =
    new SecurityTokenServiceClientConfiguration(
        SupportedProtocol.SSL_SERVER_AUTH);

// path to the truststore relative to the classpath without any prefix
config.setTruststoreLocation("/truststore.jks");
config.setTruststorePassword("changeit");

// path to the keystore relative to the classpath without any prefix
config.setKeystoreLocation("/stsclient_keystore.jks");
config.setKeystorePassword("changeit");
config.setKeyPassword("changeit");

// create client instance from configuration
SecurityTokenServiceClient client =
    SecurityTokenServiceClientFactory
        .createProtocolAwareClient(config);
```

```
// set STS url
client.setServiceUrl("https://tokenservice.icw.com");
```

15.6.3 Configuration

15.6.3.1 Configuration of the Security Token Service

The runtime behaviour of the eHF Security Token Service can be influenced by configuration properties. The relevant configuration properties can be found in file `configuration.tokensevice.properties` of an eHF assembly. If a property is not specified in the assembly configuration, the module's default will be taken.

Name	Description	Default
<code>tokensevice.issuer</code>	The value of the <code><Issuer></code> element of created assertions. It should be an identifier for the STS system, preferably its URL.	<code>http://tokensevice.icw.com</code>
<code>tokensevice.signAssertions</code>	Defines, whether the STS digitally signs issued assertions. Signing every issued assertion is recommended for production. By signing tokens, other parties can validate the token's integrity and authenticity of the issuer.	true
<code>tokensevice.tokenValiditySeconds</code>	Defines the token validity in seconds. Each issued token is only valid for a given time period. This property defines the validity interval starting from the issuing time instance. On the service provider site, the validity constraint should always be checked by the SAML login filter.	30
<code>tokensevice.keystore.filename</code> , <code>tokensevice.keystore.password</code> , <code>tokensevice.key.alias</code> , <code>tokensevice.key.password</code>	Those properties reference the private key of the Security Token Service. The private key is used for establishing SSL connections with client systems and is used for creating the digital signature of tokens. <code>tokensevice.keystore.filename</code> references the Java keystore file, <code>tokensevice.keystore.password</code> holds the keystore's password. A keystore may contain several keys, which are uniquely identifiable through their aliases, so <code>tokensevice.key.alias</code> defines the alias for the STS private key whereas <code>tokensevice.key.password</code> specifies the password for the STS private key. The default configuration only references a test key, so the private key information is required to be overridden for productive use to reflect the public key infrastructure of your solution.	Reference to default STS test key shipped with the STS module. Required to be changed for production!
<code>tokensevice.authenticator</code>	Before issuing a token, the enduser's identity has to be verified. This is done by an authenticator. The eHF STS ships with a default authenticator implementation <code>jaasAuthenticator</code> . It triggers a JAAS login for the given enduser. This means, that a JAAS <code>LoginModule</code> implementation has to determine whether the user credentials are valid or not. The eHF user management implements such a <code>LoginModule</code> which consults its user store to derive a decision. The eHF context module provides the <code>identityMappingAuthenticator</code> option which	jaas Authenticator

Name	Description	Default
	enables the context mapping scenario. Incoming user credentials in the context of one system are mapped to an identity of a second system context. Further custom authenticators can also be plugged in.	
tokenservice.jaas.realm	If the jaasAuthenticator option is selected, the STS needs to know the JAAS realm for triggering a JAAS login. The realm property has to match one of the JAAS realms which are defined in application server's JAAS configuration (e.g. catalina.login).	ehf
tokenservice.renewBearer	Decides whether renewal of tokens with subject confirmation bearer is allowed or not. Tokens with subject confirmation holder-of-key provide the highest level of security because a cryptographic proof of possession can be performed. For tokens with subject confirmation bearer, a token provider is always considered to be its possessor. In default mode, such tokens will not be renewed. For secured environments (e.g. intranets), subject confirmation bearer might be suitable although.	false

15.6.3.1.1 Context Management Support:

An eHF-based client system is able to transfer the authentication context of a locally authenticated user to another application. To do this, an eHF-based client system uses the internal security token service (internal STS) provided by the eHF Security Token Service module to issue a token for the locally authenticated user.

To enable the usage of the internal STS, an eHF-based client system must define dependencies to the eHF Security Token Service and the eHF Tokenservice Toolkit modules. Further, the eHF-based client system must have access to a Java key store that holds a private key and a corresponding X.509 certificate which are used to create holder-of-key SAML tokens and signed WS-Trust request messages.

The internal STS gets the identifier of an authenticated user (e.g. the username) via an implementation of the com.icw.ehf.tokenservice.service.SubjectProvider interface. As a default, the internal STS uses the UserPrincipalSubjectProvider. If the eHF-based client system does not use eHF-authentication and eHF-usermgmt, then a custom SubjectProvider can be configured as shown in the following Spring configuration snippet.

```
<ehf:inject target-ref="internalSecurityTokenService">
    <ehf:propRef path="subjectProvider" ref="customSubjectProvider" />
</ehf:inject>
```

After the source token has been successfully issued by the internal STS, this source token is then send to an (externally) deployed STS to request the target token. The external STS must be configured to work in "context management mode". The authenticator of the STS must be configured to alternativeAuthenticator. The following snippet of the configuration.tokenservice.properties demonstrates how this can be done.

```
# The authenticator implementation used by the STS
tokenservice.authenticator = alternativeAuthenticator
```

The eHF context module must be defined as a dependency in the STS assembly. The truststore of the external STS must contain the X.509 certificate of the internal STS to verify the signature of the source token and the holder-of-key certificate of the request message.

The tokensevice fingerprint parameter of the external STS must be configured accordingly in the configuration.tokenfilter.properties file as shown below.

```
# certificate fingerprint of the internal sts (keystore of ehf-tokensevice
module)
tokenfilter.sts.fingerprint=wofWlnbiinM/k/hIHsm/vslagri=
```

15.6.3.1.2 Endorsing Xerces and Xalan:

The OpenSAML library that is used by the eHF Security Token Service module depends on the use of an XML parser different from the one included in the Sun JDK. In order to fulfill this requirement it is necessary to override the JDK-internal implementation using a mechanism called [Endorsed Standards Override Mechanism](#). The alternative parser that has been tested for use is Apache Xerces and its Xalan counterpart as XSLT implementation. Their current version is implemented in four jars that will have to be endorsed. They are:

- xml-apis-2.9.1.jar
- xercesImpl-2.9.1.jar
- serializer-2.9.1.jar
- xalan-2.7.0.jar

If you received a release zip of an eHF application, you'll find these JARs in its directory endorsed. The actual endorsement can be performed in two different ways. One way is to put the JAR files in the dedicated JDK folder <JAVA_HOME>/jre/lib/endorsed. This however endorses the contained JARs for all invocations of this JVM. Better to put them into a separate folder that can be specified as a system property when invoking the JVM as so:

```
java -Djava.endorsed.dirs=<endorse folder> ...
```

The Tomcat servlet container delivered with eHF already comes with those libraries endorsed. All those JARs are stored at common/endorsed and the startup script endorses them at Tomcat startup by pointing to the common/endorsed folder.

15.6.3.2 Configuration of the Security Token Filter

The SAML login filter has to be configured for every service provider system which accepts tokens, issued by the eHF Security Token Service. The SAML login filter a standard servlet filter and is mapped to URLs that shall be protected in the same way as the other login filters which are provided by the eHF Authentication module.

An eHF assembly defines fragments which are merged into the web.xml. For the configuration of the SAML login filter, only src/main/config/merge/module/web-xml/filter.fragment and src/main/config/merge/module/web-xml/filter.mapping.fragment are relevant. filter.fragment holds all filter definitions, so we have to add the filter definition of the SAML login filter. The eHF Token Service module already provides two pre-configured login filters as Spring beans named samlLoginFilter and optionalSamlLoginFilter. The first one requires a SAML token in the HTTP header. The second filter can be passed without a SAML token in the header and is intended to be used in conjunction with other login filters where multiple authentication schemes should be supported. So, the filter definition in the filter.fragment has to either be

```
<filter>
  <filter-name>samlLoginFilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>
```

or

```
<filter>
  <filter-name>optionalSamlLoginFilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
```

```
</filter>
```

. Note, that the filter names are relevant because they match the exposed filter beans of the eHF Token Service module.

In the next step, the defined filter has to be mapped to application URLs by adding a filter mapping in the `filter.mapping.fragment` file:

```
<filter-mapping>
  <filter-name>samlLoginFilter</filter-name>
  <url-pattern>/services/*</url-pattern>
</filter-mapping>
```

The exemplary mapping above would enforce SAML token based authentication for all requests whose target URL matches the URL pattern above.

Once the filter is set up in the `web.xml`, we can influence the runtime behaviour of the token filter. The token filter is configured via the property file `configuration/configuration.tokenfilter.properties` inside an eHF assembly.

Name	Description	Default
tokenfilter.maxAgeSeconds	The maximum age in seconds that valid tokens are allowed to have. The age is calculated from the <code>IssueInstant</code> attribute of the assertion which represents the token's creation time. The age check is only performed if this property is set to a value larger than 0.	45
tokenfilter.clockSkewSeconds	Set this property in order to compensate time differences between the issuing system and the validating system. Its value is used as the number of seconds of tolerance in time comparisons, so that e.g. a token may be [clockSkewSeconds] older than the maximum age allowed without being rejected.	5
tokenfilter.expectedAudience	Defines the <code>entityID</code> of the service provider. The term <code>entityID</code> comes from the SAML specification and describes an identifier for a system entity that provides, consumes or otherwise participates in SAML-based services. The <code>entityID</code> uniquely identifies SAML communication participants and is typically the HTTP URL of the system.	(no default)
tokenfilter.requiresAudience Restriction	If <code>true</code> , the SAML login filters will reject tokens that do not contain an audience restriction.	false
tokenfilter.requiresSignature	Determines whether all incoming SAML tokens have to be signed in order to not get rejected by the token filter. Setting this property to <code>false</code> would skip the signature validation. Note, that the digital signature is recommended because it ensures the token's integrity and proofs the issuing entity.	true
tokenfilter.sts.fingerprint	The SHA1 fingerprint of the eHF Security Token Service certificate which issues the SAML tokens. The fingerprint is taken to retrieve the STS certificate out of the pool of trusted certificates to validate the signature. Please adapt this value according to the X.509 certificates of your Public Key Infrastructure.	50Wa8G NBeNdY a5+81b KziQ7M C4o=
tokenfilter.verbose.errors	Whether to include detailed error messages into the response when token validation fails. This is very helpful for debugging clients and needed for testing different error	false

Name	Description	Default
	conditions but not recommended in production as it also gives hints to a possible attacker.	
tokenfilter.jaas.realm	The JAAS realm used by the token filter for authentication. The realm is configured in the JAAS login configuration (for example in tomcat's <code>conf/catalina.login</code> file).	ehf

Token Cache configuration

The replay detection validation needs to remember the unique assertion IDs of the received tokens in order to detect a duplicate submission of a token. They need not to be remembered indefinitely but only as long as a token would be valid, because expired tokens are rejected anyway. An additional caveat is that in a distributed deployment every node must have knowledge of the tokens received by any other node. What is needed therefore is a caching mechanism that is able to expire its entries after some time and that can also be distributed if needed. Instead of implementing our own caching mechanism, the eHF Token Service uses ehcache – a caching implementation that is also supported by Hibernate. This cache can be configured via the file `ehcache-tokenservice.xml` that is searched for in the classpath under the `META-INF` folder. The eHF Token Service module looks for a cache configuration named `tokenCache` inside this file. An eHF assembly contains the ehcache configuration file under `src/main/config/ehcache-tokenservice.xml` where you can modify the ehcache configuration. If the configuration file is not found in the classpath at runtime, the following default configuration from inside the eHF Token Service module is used:

```
<cache name="tokenCache"
    overflowToDisk="false"
    timeToLiveSeconds="120"
    maxElementsInMemory="6000"
    eternal="false"
    memoryStoreEvictionPolicy="FIFO" />
```

In a distributed production setting it will be necessary to set up a distributed cache that is synchronized among all nodes. Also the cache should be made persistent so that cache entries are not lost at JVM restarts. Please refer to the [ehcache documentation](#) for these settings.

Session cookies and reauthentication

Just as the other login filters of eHF Authentication, the SAML Token Filter will create a server-side session and will set a session cookie in the first response after authentication. Clients may choose to include this cookie in subsequent requests in order to continue to use the once established session. This way they do not need to include a token again. It is paramount that this session cookie must not be sent over insecure connections because anyone who knows the session cookie can impersonate the user during the duration of the session. If the filter detects a token and a session cookie in a request, it will invalidate the existing session and then perform a renewed authentication based on the new token – the cookie that identifies the valid existing session is ignored. If successful, a new session is established and a new session cookie will be created.

15.6.4 Dependencies

eHF Token Service Toolkit

The eHF Token Service Toolkit does not have any references to any eHF module and can be utilized in any Java-based client application for easily communicate with an eHF STS.

eHF Security Token Service

The eHF Security Token Service has a direct dependency to the eHF Token Service Toolkit which contains shared components which are both used on client and server side.

The eHF Commons Security module provides the infrastructure for a commonly used and centralized truststore. The eHF STS uses this infrastructure to store trusted certificates.

Since end users can authenticate using their X.509 client certificates, the eHF STS has a dependency to the eHF Certificate Validation module, which performs the certificate validation.

The eHF User Management is a dependency because it provides the default user store for performing a JAAS-based authentication for end users to proof their identity.

The eHF Authorization module is a dependency because we leverage execute permissions to regulate access to the `issue` and `renew` STS methods.

The SAML login filter also is an eHF message producer which creates events for invalid SAML tokens, so the eHF STS needs a dependency to the eHF messaging module.

15.7 User Management

15.7.1 Overview

The eHF User Management component offers identity management functionality and supports the creation and administration of users. Users can be managed either globally or in the context of an organization hierarchy. The eHF User Management allows you to assign roles to users as well as to organizations. An authorization component can use this functionality to enable role based access control (RBAC) where users and organizations inherit permissions from roles. The eHF User Management component is a lightweight implementation for the management of user data that can easily be utilized by standalone business applications. It is not meant to be a full-fledged enterprise identity management solution. However, eHF Security supports the integration of 3rd party identity management systems through well-defined service provider interfaces. The following figure shows the services that are offered by the eHF User Management.

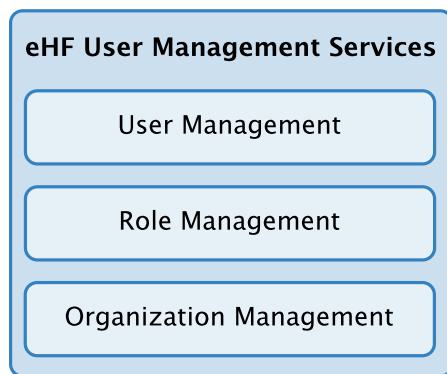


Figure 81: User Management

The User Management service provides functions to create and administer user accounts. Typically this service is used by higher level customer registration processes to enable account management for business applications. User account data includes user identifiers, personal data, contact data and payment information (all payment related signatures have been deprecated because of legal constraints, which state that payment related data has

to be stored in dedicated and certified systems). User secrets (e.g., password) can be randomly created, changed and validated against a syntax policy. User account life cycle operations such as de-activation and re-activation are supported as well.

The Role Management service allows you to manage roles and to organize roles in hierarchies. Roles typically represent the duties and rights of individuals (users) within organizations. The management and administration of permissions in a business application can be implemented very effectively by assigning permissions to roles instead of directly to users. In this context, users can inherit permissions from roles, which is the core principle behind role based access control (RBAC). A role may even inherit permissions from parent roles (hierarchical RBAC).

The Organization Management service can be used to manage hierarchically structured organizations. Nodes in the hierarchy represent sub-organizations, departments, and so on. Users can be members of one or more organizations and roles can be assigned to organizations. A user who is a member of an organization inherits the permissions that are assigned to the organization. The member inherits also all permissions that are assigned to parents of this organization. In this context, permissions may be assigned either directly to an organization or via roles that are assigned to an organization.

15.7.2 Architecture

The following diagram outlines an architectural overview of the eHF User Management component. The eHF User Management provides a User Management API, a Role Management API and an Organization Management API. All these API's can be accessed through Java interfaces as well as through Web Service interfaces.

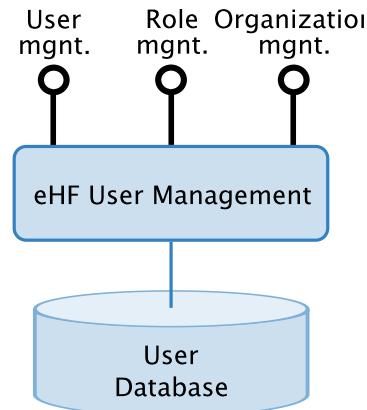


Figure 82: User Management

User Management API

eHF User Management provides a User Management API to create and administer user identity data like the user's identifiers and secrets as well as a user's personal data like addresses or payment information (all payment related signatures have been deprecated because of legal constraints, which state that payment related data has to be stored in dedicated and certified systems). User identity data and personal data is stored in the User Database, which can be any JDBC database supported by Hibernate 3.x. The eHF UserManagement API also supports a wide range of criteria and identifier based finder methods that enable business applications and processes to effectively retrieve user data from the User Database.

Role Management API and Organization Management API

eHF User Management provides a Role Management API and an Organization Management API. Role and organization data is stored in the User Database, which can be any JDBC database supported by Hibernate 3.x. Both the Role Management API and the Organization Management API offer finder methods to efficiently retrieve role and organization data from the User Database.

Domain Model

The following diagram outlines the domain model of the eHF User Management component.

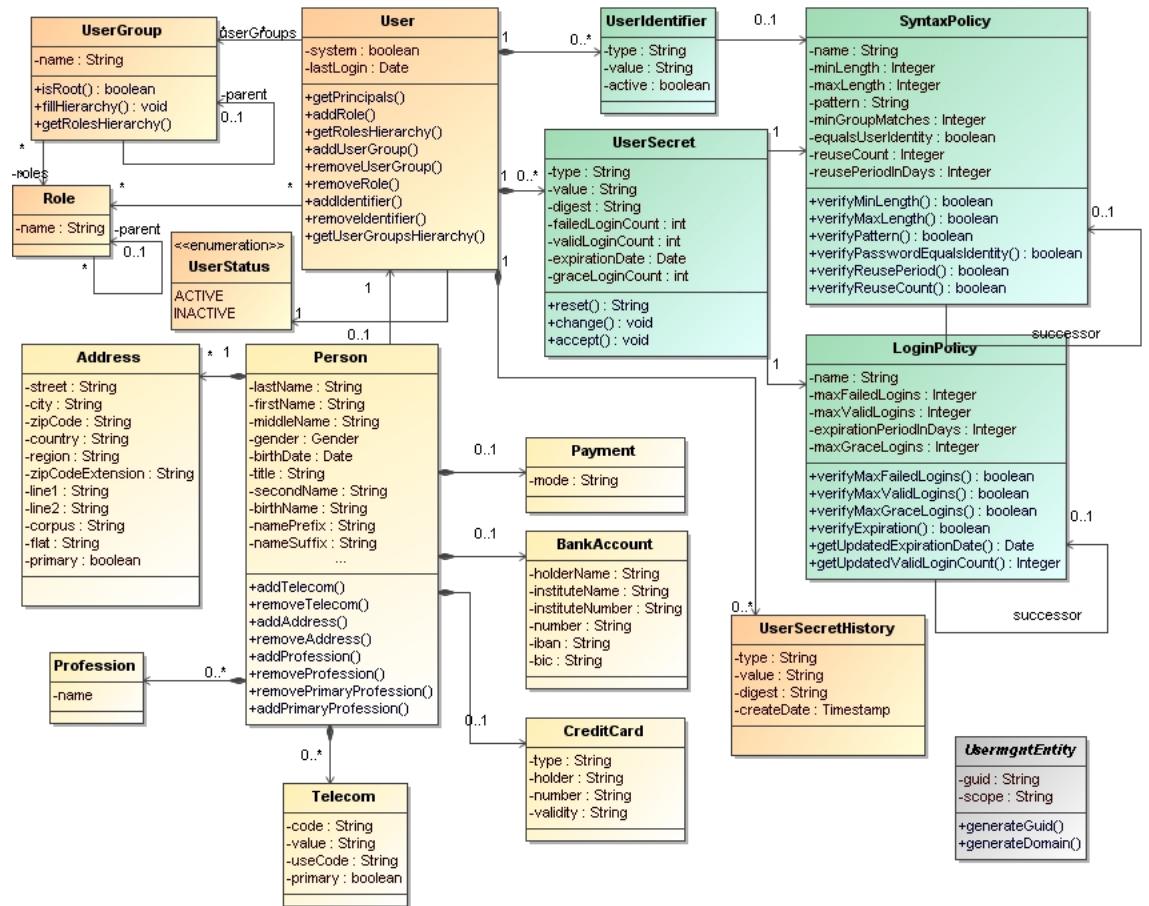


Figure 83: User Management

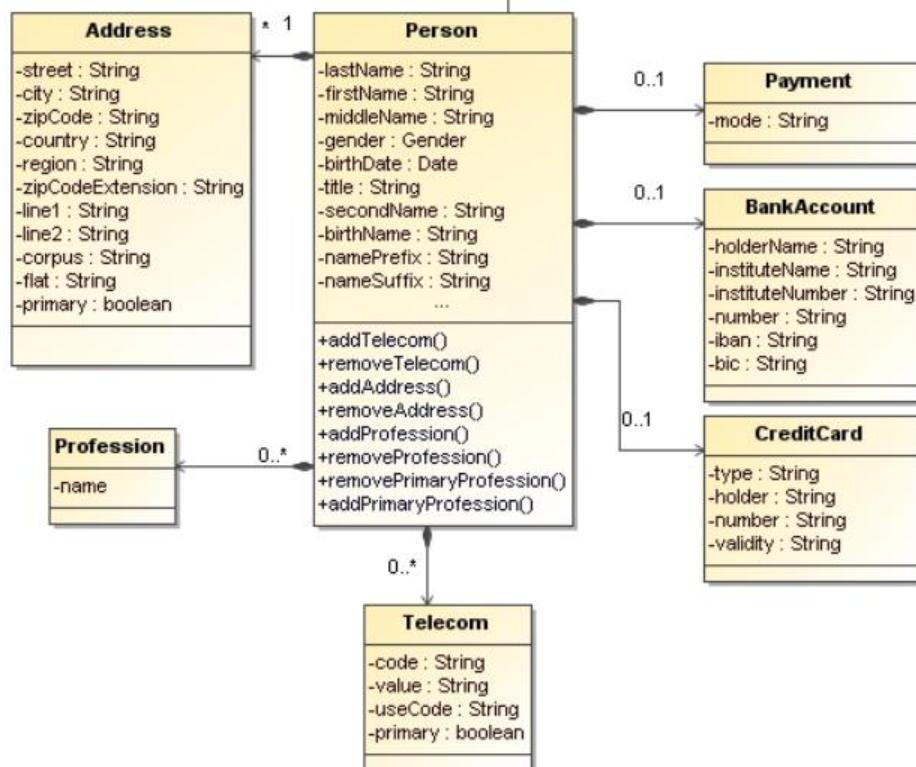


Figure 84: Person

payment data (all payment related signatures have been deprecated because of legal constraints, which state that payment related data has to be stored in dedicated and certified systems) for using business applications and services and so on.

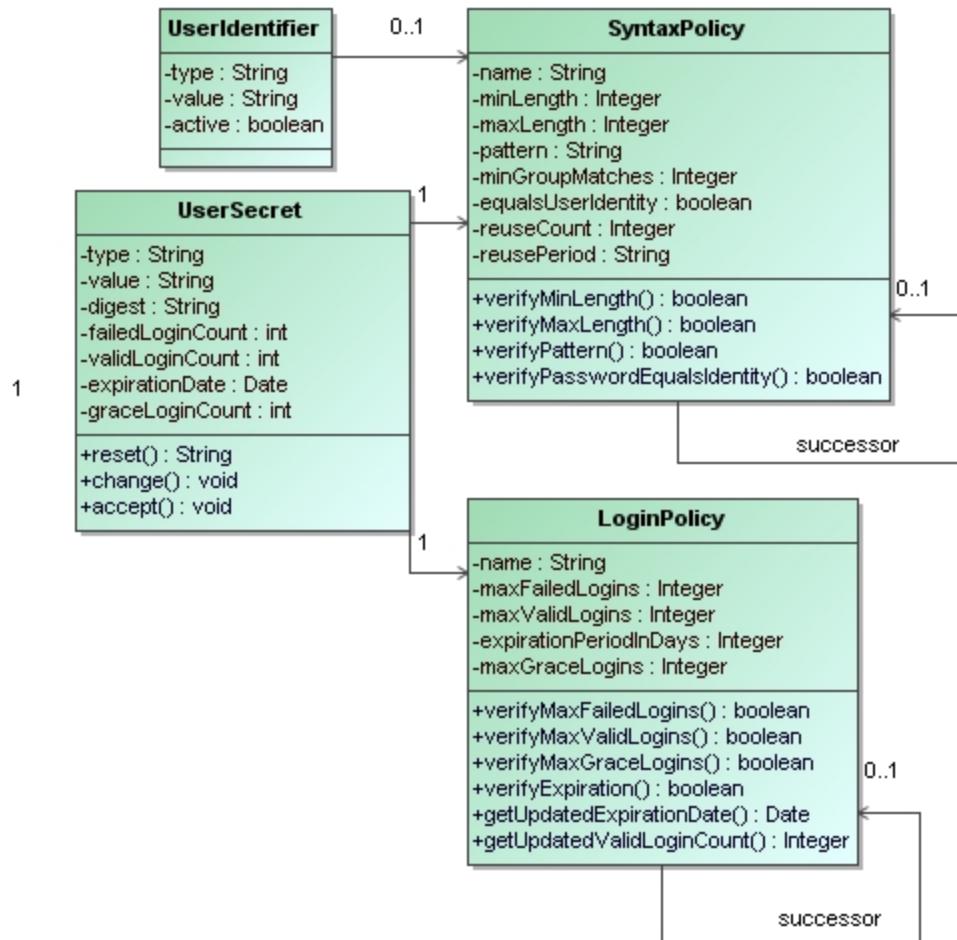
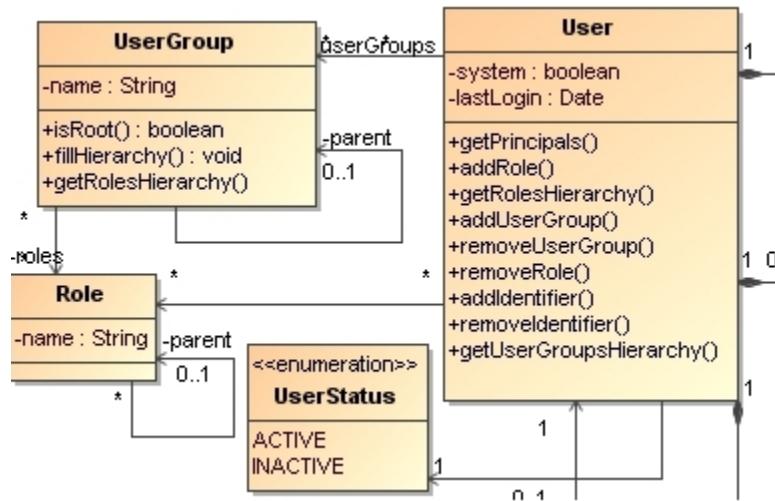


Figure 85: User Identifiers and Secrets

User is the entry point for user identity data. User contains information about a user's status, that is, whether the user's account is currently active and if the user is a system user. Multiple identifiers of different types can be assigned to a single user. Identifiers can optionally be validated against a syntax policy. The eHF User Management also allows the management of a secret for a user, called a user secret. A user secret is for example a password or a pin (personal identification number). User secret management includes creating a random secret value for a user, changing a user's secret value and validating user secrets against a customizable syntax policy. Further, the validity of a user secret can be verified against a customizable login policy that maintains an expiration period as well as limits for successful logins and failed login attempts.

**Figure 86:** Roles and User Groups

A user can be optionally assigned to one or more roles. An authorization component like eHF Authorization can leverage this information to enable role based access control (RBAC). The role's parent reference makes it possible to build role hierarchies.

A user can be assigned to organizations to reflect a user's functions and positions within organizations. Organization hierarchies are enabled by the organization's parent reference. eHF User Management further allows to assign roles to organizations, which can be used to significantly simplify the administration of users and user permissions.

Authentication Adapter

The `AuthenticationAdapter`, provided by the user management, is an implementation of the `ehf-authentication`'s [AuthenticationAdapter interface](#) on page 153. It is a stateful login service that validates the given user credentials against the eHF user management datastore.

The `AuthenticationAdapter` interface is the only interface which must be implemented to integrate a 3rd party identity management system.

15.7.3 Usermanagement and Security Annotation Framework

The eHF Usermanagement uses the [Security Annotations Framework](#) (SAF). SAF is an instance-level access control framework based on Java 5 annotations and AOP technologies (Spring AOP, AspectJ). SAF is a mechanism for authorization policy enforcement, i.e. the SAF `SecurityInterceptor` intercepts the call of annotated code and enforces authorization.

The Java snippet below shows the class `com.icw.ehf.usermgmt.security.hibernate.SecurityInterceptor` using SAF annotations for checks on data access level. For example `SecureAction.CREATE` enforces write checks, i.e. the eHF Authorization checks if the user has permission to create this object.

```

public class SecurityInterceptor extends EmptyInterceptor {

    public boolean onSave(@Secure(SecureAction.CREATE)
        Object entity, Serializable id, Object[] state, String[] propertyNames,
        Type[] types) {
        ...
    }
}
  
```

```

public boolean onFlushDirty(@Secure(SecureAction.UPDATE)
    Object entity, Serializable id, Object[] currentState, Object[]
previousState,
    String[] propertyNames, Type[] types) {
    ...
}

public void onDelete(@Secure(SecureAction.DELETE)
    Object entity, Serializable id, Object[] state, String[] propertyNames,
Type[] types) {
    ...
}
}

```

The next Java snippet shows the interface `com.icw.ehf.usermgnt.service.PersonService` using SAF annotations for read checks on the service level with *Filter*. The Set of Telecom objects are filtered by the authorization mechanism, only those are returned for which the user has read permissions.

```

public interface PersonService extends CrudService<Person> {
    @Filter
    public Set<Telecom> getTelecoms() {
        ...
    }
    ...
}

```

The XML snippet below shows the Spring configuration. The calls are forwarded to the Access Manager of eHF Authorization.

```

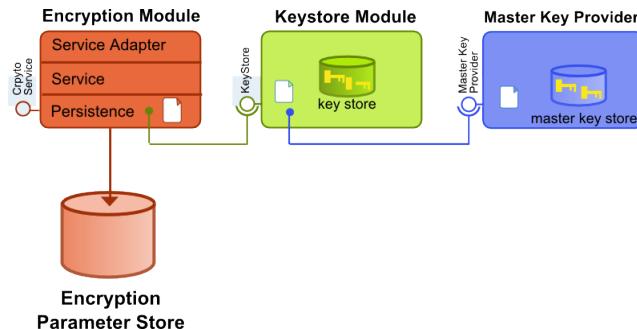
<!-- Annotation Driven Security -->
<sec:annotation-driven
    interceptor-order="2"
    access-manager="accessManager"
    support-aspectj="true"
/>

<!-- Access Manager -->
<bean id="accessManager"
    class="com.icw.ehf.usermgnt.security.AccessManagerImpl"
/>

```

15.8 Encryption

In eHF several modules together provide the encryption infrastructure to support pseudonymization. The eHF Commons Encryption module defines basic classes, utility classes and the services (i.e. in terms of Java interfaces). The eHF Encryption module implements the encryption and decryption services. Internally, it utilizes various cryptographic engines. The eHF Keystore module provides a repository of keys required by cryptographic engines. The repository and the keys inside are protected by key encrypting keys provided by the Master Key Provider. [Figure 87](#)) gives an overview of the encryption infrastructure.

**Figure 87:** Overview of the encryption infrastructure

15.8.1 Commons Encryption Module

The Commons Encryption module defines the basic classes (such as target and result of a cryptographic operation, and encryption-related exceptions), cryptographic services and utility classes.

CryptoTarget and CryptoResult

A **CryptoTarget** defines an attribute of a domain object which is to be encrypted or decrypted. If multiple attributes of a domain object need to be encrypted, multiple targets are required. A **CryptoTarget** has the following properties:

Property	Documentation
moduleId	Identifies the module which the owning domain object of this attribute belongs to.
ownerClass	Class name of the owning domain object.
attributeName	Name of the attribute.
dataType	Data type of the attribute.
value	Value of the attribute.
scope	Scope of the owning domain object.
id	Identifier of this particular domain object instance.
classification	Specifies the required security level for this target.

A **CryptoResult** consists of the following two properties:

- **target** Target of the operation.
- **result** Resulted cipher text or plain text depending on the operation performed.

CryptoService

Cryptographic services are specified in the interface **CryptoService**. Its operations are defined as follows:

```

/**
 * Encrypts a variable number of targets and returns the encryption results.
 * @param encryptionTarget A number of encryption target.
 * @return An array of encryption result.
 * @throws CryptoException Thrown in case the encryption can not be performed.
 */
CryptoResult[] encrypt(CryptoTarget... target) throws CryptoException;
  
```

```

/**
 * Encrypts the targets with the given context and returns the encryption results.
 * @param context Encryption context (e.g. for a query).
 * @param encryptionTarget A number of encryption target.
 * @return An array of encryption result.
 * @throws CryptoException Thrown in case the encryption can not be performed.
 */
CryptoResult[] encrypt(EncryptionContext context, CryptoTarget... target) throws
CryptoException;

/**
 * Decrypts a variable number of targets and returns the decrypted results.
 * @param cryptoTarget A number of decryption targets.
 * @return An array of decryption result.
 * @throws CryptoException Thrown in case the decryption can not be performed.
 */
CryptoResult[] decrypt(CryptoTarget... target) throws CryptoException;

```

Cryptographic services can be provided either on the same machine as the application modules or on dedicated servers accessable through the network. Network latency can cause a significant performance penalty. Hence, `CryptoService` supports bulk encryption and decryption to reduce the number of requests. For example, only one request is issued for all encryption-relevant attributes of a domain object.

`CryptoEngine`

Cryptographic engines implement cryptographic algorithms (e.g. AES, SHA1) which the encryption infrastructure depends on. Cryptographic implementations can be offered by different providers, for example, Sun's Java Cryptography Extension (SunJCE) or Bouncy Castle. To ensure implementation independence, eHF Commons Encryption defines the interface `CryptoEngine`. Its operations are defined as follows:

```

/**
 * Performs an encryption on the given plaintext with the specified parameter.
 * @param plaintext Target to be encrypted.
 * @param param Parameter to use for the encryption.
 * @return Result of the encryption. In case the plain text is null
(uninitialized),
 * a null is returned as result.
 */
byte[] encrypt(byte[] plaintext, CryptoParameter param);

/**
 * Wraps the given {@link InputStream} into an encrypting {@link InputStream}.
 * @param plainStream Plainstream to be encrypted.
 * @param param Parameter to use for the encryption process.
 * @return A wrapped {@link InputStream} that delivers the encrypted content.
 * In case the plainstream is null(uninitialized), a null is returned as result.
 */
InputStream encrypt(InputStream plainStream, CryptoParameter param);

/**
 * Performs a decryption on the given target with the specified parameter.
 * @param cipherText Target to be decrypted.
 * @param param Parameter to use for the decryption.
 * @return Result of the decryption. In case the ciphertext is
null(uninitialized),
 * a null is returned as result.
 */
byte[] decrypt(byte[] cipherText, CryptoParameter param);

/**
 * Wraps the given {@link InputStream} into a decrypting {@link InputStream}.
 * @param cryptedStream Cipher stream to be decrypted.
 * @param param Parameter to use for the decryption process.
 * @return A wrapped {@link InputStream} that delivers decrypted content.

```

```
* In case the cipher stream is null(uninitialized), a null is returned as result.

*/
InputStream decrypt(InputStream cryptedStream, CryptoParameter param);
```

Cryptographic engines take plaintext as byte sequences and return ciphertexts also as byte sequences. However, application data to be encrypted is often of types such as String, Integer etc. Hence, a conversion of target values to byte sequences is required before the actual encryption. In eHF, encrypted values of data types which implement Serializable are stored as Strings in the database. After an encryption, the resulted ciphertext as a byte sequence is encoded into a String in base-64 representation. Before a decryption, the String in base-64 is decoded and translated back to a byte sequence. After a decryption, the resulted plaintext as a byte sequence is converted to the original type of the plaintext. The eHF Commons Encryption module provides utility classes for the type conversions. Currently, converters for the types String, Integer, Long, Boolean, Java Date, and ByteArray are provided. They are used by the Encryption module, transparently for the application modules.

Apart from the API to encrypt and decrypt byte sequences, CryptoEngine provides operations to encrypt and decrypt streams of data received from the file system or the network. With the API for streaming, there is no need to get all the data into buffers before it can be encrypted or decrypted.

KeyStore

The protection of sensitive data strongly relies on the security of the keys used for encryption. Usually, the keystore, a repository of key data, is located in a highly protected environment. All accesses to the keystore is guarded by authentication and authorization. The number of entities that need access to the key is kept to the minimum. In eHF, only cryptographic engines and a key manager have access to the keys in the keystore. The rest of the encryption infrastructure and the application modules reference to the keys via key aliases.

The eHF Commons Encryption module defines two interfaces for key accesses, KeyStore and ManagedKeyStore.

- KeyStore provides the operation `loadKeyByAlias(alias)` for acquiring the actual key which is associated to a given key alias. The operation `containsKey(alias)` returns whether a key with the given alias exists in the keystore. The number of keys in the keystore is given by the operation `size()`.
- ManagedKeyStore extends the functionality of KeyStore by providing additional operations for key manipulation. The operation `storeKey(alias, key)` inserts a new key into the keystore, while the operation `deleteKey(alias)` removes an existing key with the given alias from the keystore.

A trusted administrator uses a KeyManager to generate keys and manage keys. A key manager must verify that the administrator is authorized to conduct the desired operations. Only a key manager has access to the interface ManagedKeyStore, while cryptographic engines have access to the interface KeyStore.

CryptoException

TODO

15.8.2 Encryption Module

The eHF Encryption module provides the implementation of `CryptoService`. The internal architecture of the Encryption module is shown in [Figure 88](#). Upon receiving an encryption or decryption request consisting of a number of `CryptoTargets`, the `CryptoService` component retrieves cryptographic parameters (such as algorithms, keys and initialization vectors) from the `ParameterProvider` component, and utilizes the `CryptoEngine`

component to perform the actual encryption and decryption. After a successful encryption, a `CryptoReceipt` consisting of the cryptographic parameters used for the encryption is persisted. The receipts are retrieved by the `ParameterProvider` component for subsequent decryptions.

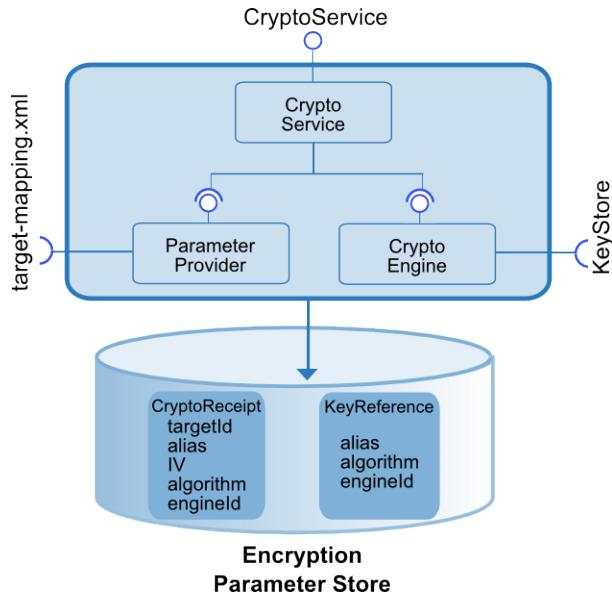


Figure 88: Internal architecture of the Encryption module

Key references

As mentioned before, the number of entities that need access to the keys shall be minimized. With the exception of the `CryptoEngine` component, all other components of the Encryption module interact with references to the keys. Each key is generated for a particular algorithm. It is referenced by one alias and assigned to one cryptographic engine. The Encryption module persists a key reference table which keeps track of the information (`alias`, `engineId`, `algorithm`) for each key. The table is updated whenever new keys are generated or existing keys are deleted.

Crypto receipts

The number of `CryptoReceipts` to be persisted depends on the scopes of keys and IVs. If a key is selected based on a classification, only one receipt is persisted for all attributes of the same classification. The narrowest scope, i.e. instance-based key or IV, requires for each domain object instance a receipt. Each `CryptoReceipt` has a property `targetId` which is calculated based on the properties (such as classifications or scopes) which are used to select the key or IV.

Determining encryption and decryption parameters

Upon receiving a `CryptoTarget` in an encryption or decryption request, the `ParameterProvider` component consults the configuration in the `target-mapping.xml` file to determine the scope of the key and IV to be used.

In the case of an encryption, the `ParameterProvider` generates new keys and IVs if no `CryptoReceipt` for the required scope is available. The decryption of a ciphertext requires the same cryptographic parameters used for the previous encryption. Thus, a `CryptoParameterNotFoundException` will be thrown, if no `CryptoReceipt` can be found for a decryption.

Encrypting query parameters

Using application-level encryption, query parameters must be encrypted before the query is issued to the database. Since we query against data, which is assumed to be encrypted and existent in the database, neither new key nor new IV will be generated for a query. If no `CryptoReceipt` can be found to encrypt the query parameter, the query is considered as invalid and an `InvalidQueryException` will be thrown.

The following gives a typical example of an invalid query. The attribute `createDate` has been encrypted using an instance-based key. A search request which only consists of the `createDate` in plaintext, but no identifier of this `IntakeEntry` cannot succeed.

Since the generator also generates the code for queries, it passes an `EncryptionContext` object to the Encryption module, indicating whether the encryption request is for a persistence operation or for a query.

15.8.3 Keystore Module

The eHF Keystore module provides an implementation of `KeyStore` and `ManagedKeyStore`. Currently, it is based on Sun's default implementation in the `java.security` package, which uses a file to store the keys. Additionally, the Keystore module implements the interface `KeyManager` to generate keys.

15.9 Pseudonymization

eHealth applications use (centralized) repositories to store personal and medical information of patients. Privacy of the stored data and patients-controlled authorization of their trusted health care providers and relatives are key requirements for eHealth applications. Usually, medical studies rely on anonymized medical data which no more have a link to the associated patient. However, patients would benefit from getting informed about actual findings of a study in their healing process.

Pseudonymization is an approach that provides a form of traceable anonymity of the stored data. Instead of completely removing identification information from the medical data, identification information is transformed into a piece of information (i.e. pseudonym) which cannot lead to the patients without knowing a certain secret. Consequently, legal, organizational or technical procedures are required to establish the association between medical data and the related patients.

Encryption is one of the mechanisms for building pseudonyms. We distinguish between database-level encryption and application-level encryption. The main objective of a database-level encryption is to prevent access to the data in plaintext in case of a theft of the database or the backup media. It is transparent for applications. However, it does not restrict accesses of privileged, malicious database administrators (in short DBA). In contrast to database-level encryption, application-level encryption is not transparent to the application. Application data will be encrypted before it is persisted to the database. Application-level encryption is independent from database technologies and protects the data from malicious DBAs. In eHF, we use application-level encryption. As illustrated in [Figure 89](#), the Medicine-Cabinet module invokes the cryptographic service `CryptoService` provided by the Encryption module, before pseudonymization-relevant data is persisted to the database.

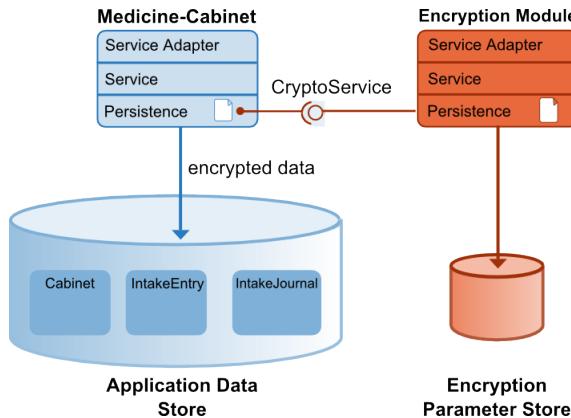


Figure 89: Pseudonymization using application-level encryption

In eHF, application-level encryption is accompanied by a model-driven pseudonymization approach. From the viewpoint of an application developer, enabling pseudonymization consists of the following two steps:

- classifying pseudonymization-relevant attributes in the model, and
- parameterizing security levels of the classifications by configuration.

From the perspective of eHF, pseudonymization is supported by

- the generator, which generates code to encrypt and decrypt values of pseudonymization-relevant attributes both for persistence and for queries, and
- the underlying encryption infrastructure.

15.9.1 Classification

Different categories of data needs to be protected by different security levels. On the one hand, the weakest security level may not fulfill the privacy requirements for certain highly sensitive data. On the other hand, using the strongest security level for all data would unnecessarily waste resources and decrease the runtime performance of the whole system. Applications have the best knowledge about the security levels for their data to be protected.

In eHF, an attribute becomes relevant for the pseudonymization only if it is classified in the model. A classification assigns the data to different categories.

- A classification implies a semantic grouping of data. For example, the credit card number and expiration date could be grouped together, because it is likely that when one changes, the other changes as well.
- A classification can be used to indicate the required security level. The selection of most appropriate cryptographic algorithm and encryption parameters (e.g. length of keys) can be based on the classification.

Classification is a concept which is not constrained to a domain object class or a module. Instead, a classification such as the "scope" is valid across the border of domain object classes and modules.

15.9.2 Configuration

A classification in the model maps attributes of domain objects to different categories. It is a prerequisite for applying different security levels to different categories of data. Different cryptographic parameters (e.g. algorithms and keys) can be used to encrypt data of different classifications. From the perspective of cryptography, the amount of data encrypted with the same key shall be limited, in order to limit the effectiveness of known ciphertext attacks which rely on a large collection of data encrypted with the same key. Hence, the best practice is to use dedicated keys to encrypt differently classified data. We refer keys

which are selected based on classifications as classification-based keys. In [Figure 90](#), a classification-based key is used to encrypt the attributes `isoDate` and `canonicDate`.

Moreover, one classification can be mapped to multiple keys. For example, different keys can be used for attributes which have the same classification but reside in different domain objects or modules. In [Figure 90](#), a module-based key is used to encrypt the attribute `scope` which has the classification `scope`.

Going a step further, even inside one domain object class, dedicated keys can be used for different attribute instances. The `IntakeEntry` table in [Figure 90](#) shows two scopes, `Langstrumpf` and `Nilson`. The first three instances of the attribute `comment` in the table belong to the scope `Langstrumpf`, and are encrypted with the same key, while the two instances belonging to the scope `Nilson` are encrypted with another key. Keys, which are selected based on the scopes of associated data, are referred to as scope-based keys. For each instance of the attribute `createDate`, a dedicated instance-based key is used.

IntakeEntry					
	date	scope	freeText	c-date	
classification	createDate	scope	comment	isoDate	canonicDate
			Langst.		
			Langst.		
			Langst.		
			Nilson		
			Nilson		

IntakeEntry					
	createDate	scope	comment	isoDate	canonicDate
key scoping					

instance-based
module-based
scope-based
classification-based

key
key
key
key

Figure 90: An example of classification and key scoping

Since many protected data in the database has a pre-defined format or a very narrow value range, data is highly predictable and thus vulnerable for chosen plaintext attacks. For example, the result of an HIV test is either positive or negative. An attacker having access to the encryption service could issue requests to encrypt the two possible results of an HIV test. By comparing the resulted ciphertexts with the ciphertexts in the database, the attacker can derive the plaintexts. Block ciphers use random initialization vectors (IV) which are XORed with the plaintexts before their encryption. However, the same plaintext still leads to the the same ciphertext if the same key is used. Statistical methods can be used to guess the values of the plaintexts. Using unpredictable and unique IVs for each instances of plaintexts is a way to solve this problem.

The scope of a key or an IV defines the scope in which the key or the IV is valid. In eHF, the scope of a key can be a particular classification, a module, a domain object class, a scope or a particular instance. If not specified, for each new key, a new IV is created. This results in an IV having the same scope as the key. Moreover, a narrower IV scope than the

related key scope can be used. In the example of an HIV test, a classification-based key and instance-based IV are appropriate to encrypt the results. The mapping from classifications to cryptographic parameters is defined in a configuration file `target-mapping.xml`.

15.9.3 Generator support

To support pseudonymization, each attribute of a domain object is represented by two representations (see [Figure 91](#)). The external representation represents the state of the attribute in plaintext to serve the perspective of an application. The internal representation represents the state of the attribute from the perspective of the database, which is encrypted if the attribute is pseudonymization-relevant.

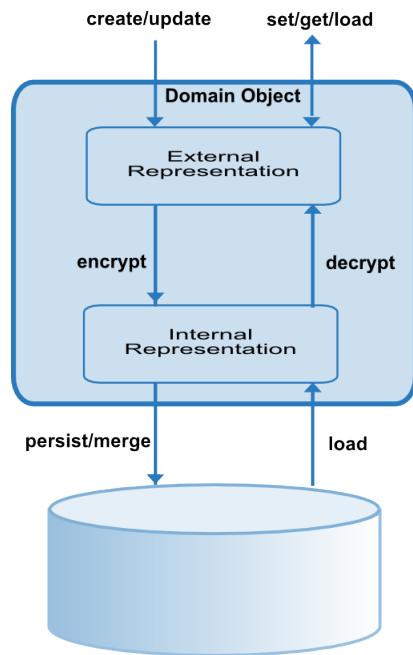


Figure 91: Internal and external representation of a domain object

When invoking the setter and getter methods of a domain object, an application interacts with the external representation of the domain object. Before the state of a domain object is persisted to the database either by creation or by update, a synchronization from its external to internal representation occurs. Attribute values as plaintexts in the external representation will be encrypted, and the resulted ciphertexts are stored into the internal representation.

Whenever a domain object is loaded from the database, only its internal representation has been initialized by the database. A synchronization from internal to external representation decrypts the ciphertexts in the internal representation, and fills the external representation with the resulted plaintext.

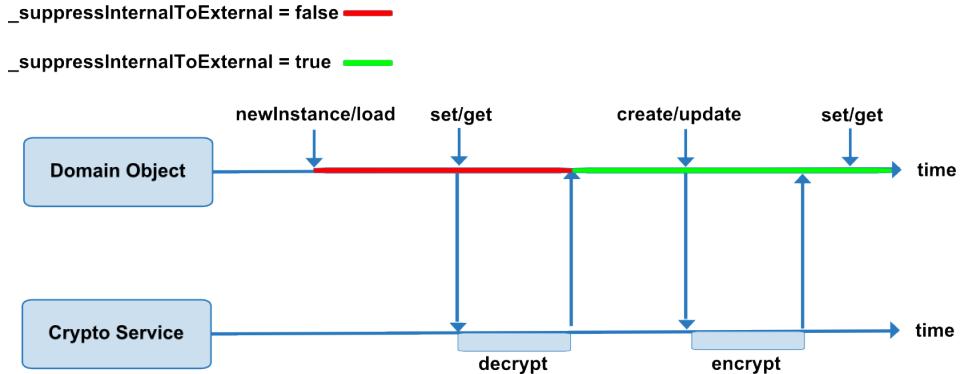


Figure 92: Interaction between a domain object and CryptoService

Since each synchronization between the two representations is associated with either an encryption or a decryption which may cause a performance penalty, we use a relaxed consistency model between the two representations. This is controlled by the flag `_suppressInternalToExternal` in the domain object. Figure 92) shows the interaction between a domain object and the CryptoService in response to method invocations on the domain object.

- Whenever a domain object is instantiated, the flag `_suppressInternalToExternal` is initialized as false. After a domain object has been loaded from the database (a load operation instantiates the domain object), the first setter/getter method on the domain object triggers a decryption to synchronize the external representation with the internal representation consisting of encrypted values in the database. The first setter/getter method sets the flag `_suppressInternalToExternal` to be true, such that subsequent setters/getters do not require a decryption anymore.
- With no exception, each create/update operation on a domain object triggers an encryption to synchronize the internal representation with the external representation.

The generator generates the code in domain objects and DAOs to control the synchronization logic. In particular, for each create, load, and update operation, the DAO calls the `synchronize()` method of the involved domain object which subsequently invokes its own `synchronizeRepresentation()` method. The `synchronizeRepresentation()` method is also called in the setter/getter methods.

In addition to the synchronization purpose, encryption and decryption are used for database queries. For pseudonymization-relevant attributes of a domain objects (which are classified in the model), the generator generates the code to create targets of cryptographic operations, and to trigger their modulation. Later in the application modules, the modulation invokes cryptographic services.

When data from a database is encrypted, the data type of the associated column needs to be modified. For example, an integer becomes a String after the encryption. The generator performs the necessary schema modification.

15.10 Commons Identity

The eHF Commons Identity module provides an application programming interface (API) for the administration of user data as well as for the administration of data that belongs to the individual entity that represents the user in the real world. Here, this individual entity can be a person, a system, a device, an organization, etc.

The design goal of the eHF Commons Identity module is to decouple an eHF-based application from the implementation and the technology of the underlying user data store

which can be a relational database, a LDAP system, etc. That is, the eHF Commons Identity module does not provide the implementation to access a user data store. Instead, the eHF Commons Identity module provides the interfaces and the domain model for the user data administration domain. The actual implementation for the access a user data store respectively a user management system is provided by modules like for example the eHF LDAP Identity module, that enables to access user data managed within a LDAP system.

The PXS system is a good example for an application that needs this kind of flexibility to satisfy the different user data store requirements of their customers.

15.10.1 Architectural Design

The eHF Commons Identity module provides a number of service APIs that can be used by eHF-based applications for the administration of user data. Currently, the supported service APIs are `UserService`, `PersonService` and `UserGroupService`. In the future, the provided functionality will be extended according to the incoming requirements. As the eHF Commons Identity module does not provide the functionality to access the physical user data stores, the module contains only the interfaces and not the implementation these interfaces. The implementation is provided by other modules like the eHF LDAP Identity module for example.

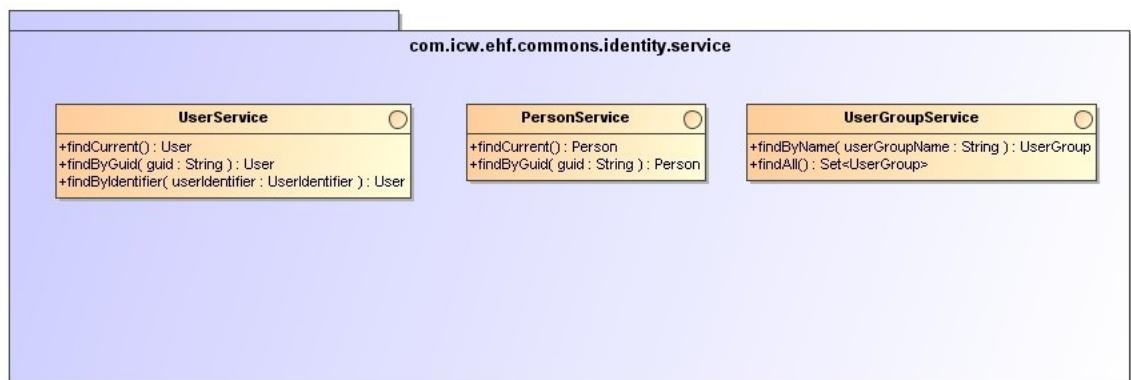


Figure 93: Architecture of Commons Identity

15.10.2 Domain Model

As stated above, the eHF Commons Identity module enables an eHF-based application to manage and find **User** data as well as data of the real-world individual related to the **User**. Currently, the only supported real-world individual in eHF Commons Identity **Person**. However, in the future, eHF Commons Identity will also support other real-world individual entities like system, device and organization, depending on the incoming requirements.

A **User** can be identified by different identifiers like a user name or a X.509 certificate etc. within a single system or across different systems. The **UserIdentifier** object takes care of this.

To support role-based access control (RBAC) for applications, one or more **Roles** can be assigned to a **User**. **Roles** can be organized in hierarchies to support hierarchical role-based access control (HRBAC).

In the same way as role-based access control is supported, the eHF Commons Identity module supports group-based access control. An arbitrary number of **UserGroups** can be assigned to a User. UserGroups can also be organized in hierarchies.

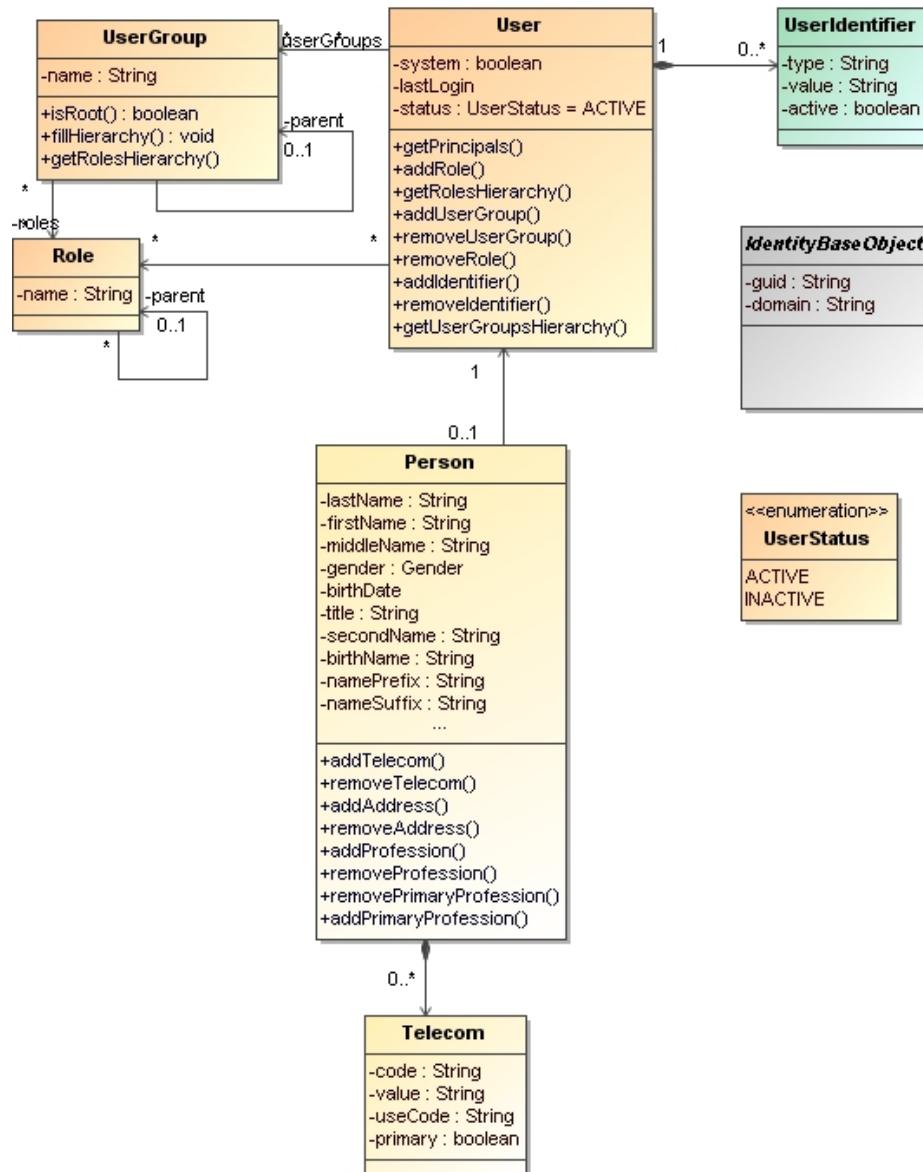


Figure 94: Domain Model of Commons Identity

15.10.3 Usage

As the eHF Commons Identity module only provides the interfaces for the administration of user data, an eHF-based application uses the eHF Commons Identity module always in combination with an implementation module. Therefore, please refer to the usage documentation of the respective implementation module. See for example the documentation of the eHF LDAP Identity module.

15.10.4 Dependencies

15.10.4.1 Internal

The eHF Commons Identity module has dependencies to the following eHF modules:

- eHF Commons

The Maven2 pom:

```

<dependency>
    <groupId>com.icw.ehf</groupId>
    <artifactId>ehf-commons</artifactId>
    <version>${project.version}</version>
    <scope>compile</scope>
    <classifier>runtime</classifier>
</dependency>

```

15.10.4.2 External

The eHF Commons Identity module has no dependencies to third party libraries.

15.11 LDAP Identity

eHF LDAP Identity module is an extension of the eHF Commons Identity module. It extends the eHF Commons Identity module with LDAP connectivity, so that personal identifiable information can be stored in a LDAP system like [ApacheDS](#) or ADAM..

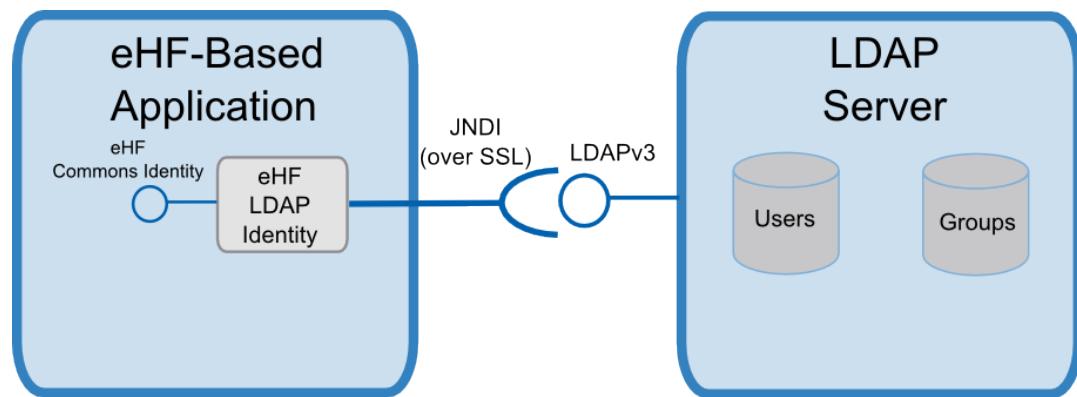


Figure 95: Overview

Furthermore, eHF LDAP Identity module holds an LDAP-specific implementation of AuthenticationAdapter from the eHF Authentication module.

15.11.1 Architectural Design

eHF LDAP Identity implements the interfaces of eHF Commons Identity.

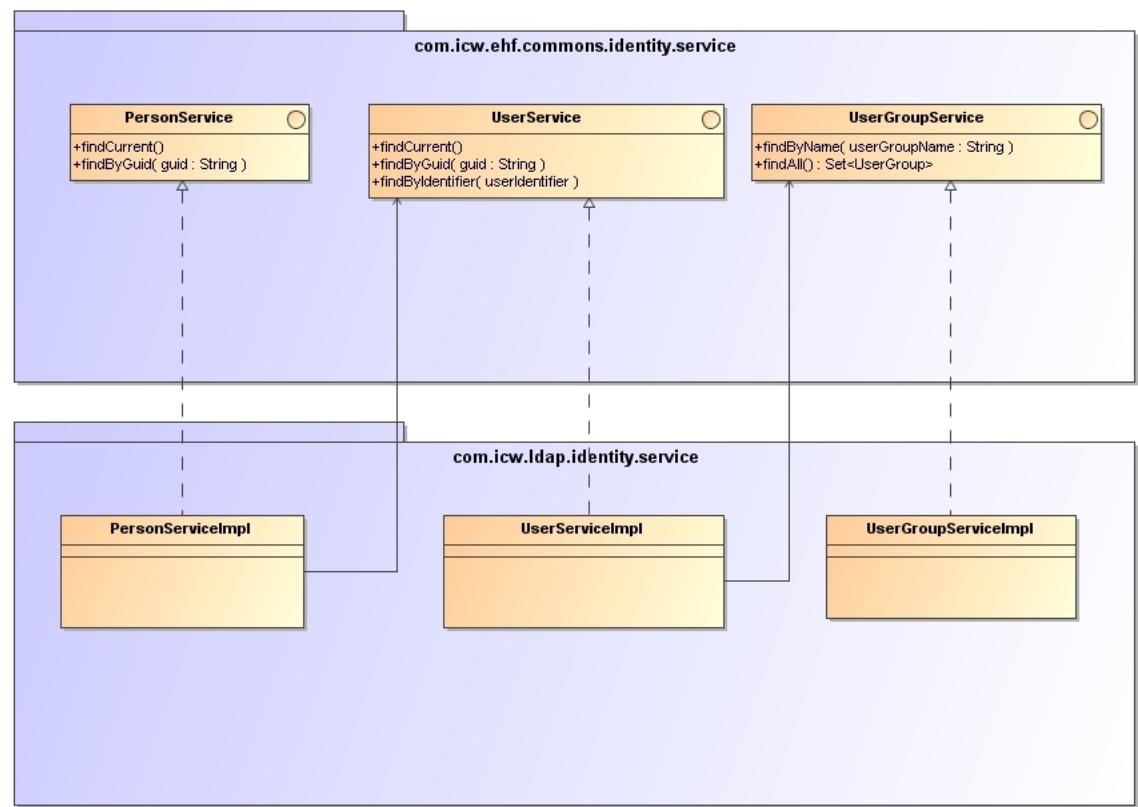


Figure 96: Architecture

Furthermore, **PersonServiceImpl** uses the **UserService** and the **UserServiceImpl** uses the **UserGroupService** of eHF Commons Identity. This is useful because a **Person** object contains a **User** object, and a **User** object can contain many **UserGroup** objects.

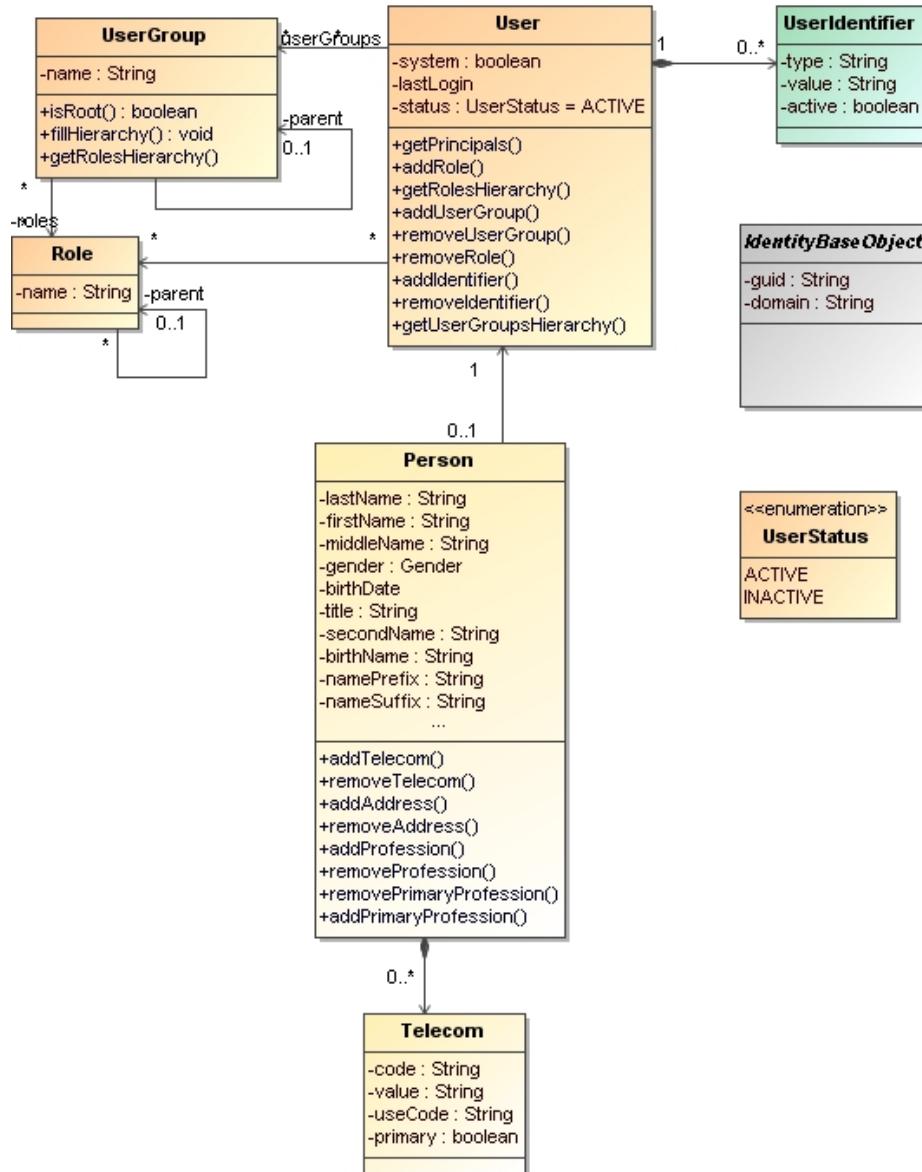


Figure 97: Commons Identity Domain Model

Furthermore, eHF LDAP Identity has an implementation of `AuthenticationAdapter` from the eHF Authentication module, called `AuthenticationService`. Currently, it supports only the user-name-and-password-login.

15.11.2 Usage

The eHF LDAP Identity module can be only used in combination with eHF Commons Identity.

15.11.3 Configuration

Three things must be configured for the LDAP identity module. First, the mapping between LDAP attributes, and object fields, and the module configuration have to be configured in the file `ehf-ldap-identity.properties` (location is `src/main/resources/META-INF`

). Then, the connection to the LDAP system has to be configured. Mostly, it is configured on assembly level in the file `ehf-system-context.xml` or for testing on module level in the file `ehf-ldap-identity-test-context.xml`. Finally, you need a LDAP-specific JAAS configuration in your servlet container (in case of Tomcat it is the file `catalina.login` in the folder `$TOMCAT_HOME/conf`).

The following points show two sample configurations, in fact for ApacheDS and for ADAM of Microsoft:

- ApacheDS:

```
#ehf-ldap-identity.properties
#LDAP attribute configuration
user.guid=entryUUID
user.domain=entryUUID
user.identifier.username=uid
user.identifier.guid=entryUUID
user.identifier.domain=entryUUID

object.userGroup.name=Roles
userGroup.guid=entryUUID
userGroup.name=cn

person.guid=entryUUID
person.domain=entryUUID
person.firstName=givenName
person.lastName=sn
person.title=title
person.telephone.value=telephoneNumber
person.email.value=mail

#System configuration
userGroup.base.dn=ou=parteien,ou=gruppen
system.domain=system
userGroup.objectclass=groupOfNames
member.attribute=memberOf
ldap.server.type=ApacheDS
```

```
<!-- ehf-ldap-identity-test-context.xml -->
<bean id="ldapTemplate" class="org.springframework.ldap.core.LdapTemplate">
    <property name="contextSource">
        <bean class="org.springframework.ldap.core.support.LdapContextSource">
            <property name="url" value="ldap://localhost:10389" />
            <property name="base" value="dc=bundestag,dc=de" />
            <property name="userDn" value="uid=jackerma,ou=personen,dc=bundestag,dc=de" />
            <property name="password" value="password" />
            <property name="dirObjectFactory" value="com.sun.jndi.ldap.obj.LdapGroupFactory"/>
            <property name="baseEnvironmentProperties">
                <map>
                    <entry key="java.naming.factory.state" value="com.sun.jndi.ldap.obj.LdapGroupFactory" />
                </map>
            </property>
        </bean>
    </property>
</bean>
```

```
#catalina.login
ehf{
    logoutHandler.1 = com.icw.ehf.authorization.service.PermissionLogoutHandler
        adapter=com.icw.ehf.ldap.identity.service.AuthenticationService
        userServiceName=ldapUserService
```

```

        userNamePrefix="uid="
        userNamePostfix=",ou=personen,dc=bundestag,dc=de"
        providerUrl="ldap://localhost:10389/dc=bundestag,dc=de"
        additionalAdapter.1=com.icw.ehf.usermgnt.service.AuthenticationService
        additionalAdapter.1.userServiceNames=usermgntUserService;
    }
}

```

- ADAM:

```

#ehf-ldap-identity.properties
#ADAM attribute configuration
user.guid=objectGUID
user.domain=objectGUID
user.identifier.username=userPrincipalName
user.identifier.guid=objectGUID
user.identifier.domain=objectGUID

userGroup.guid=objectGUID
userGroup.name=cn

person.guid=objectGUID
person.domain=objectGUID
person.firstName=givenName
person.lastName=sn
person.title=title
person.telephone.value=telephoneNumber
person.email.value=mail

#System configuration
system.domain=system
userGroup.objectclass=group
object.userGroup.name=Roles
member.attribute=memberOf
ldap.server.type=ADAM

```

```

<!-- ehf-ldap-identity-test-context.xml -->
<bean id="ldapTemplate" class="org.springframework.ldap.core.LdapTemplate">
    <property name="contextSource">
        <bean class="org.springframework.ldap.core.support.LdapContextSource">
            <property name="url" value="ldap://localhost:389" />
            <property name="base" value="dc=bundestag,dc=de" />
            <property name="userDn" value="jackerma@bundestag.de" />
            <property name="password" value="password" />
            <property name="baseEnvironmentProperties">
                <map>
                    <entry key="java.naming.ldap.attributes.binary" value="objectGUID" />
                </map>
            </property>
        </bean>
    </property>
</bean>

```

```

#catalina.login
ehf{
    logoutHandler.1 = com.icw.ehf.authorization.service.
PermissionLogoutHandler
        adapter=com.icw.ehf.ldap.identity.service.AuthenticationService
        userServiceNames=ldapUserService
        providerUrl="ldap://localhost:10389/dc=bundestag,dc=de"
        additionalAdapter.1=com.icw.ehf.usermgnt.service.AuthenticationService
        additionalAdapter.1.userServiceNames=usermgntUserService;
}

```

```
};
```

For further information about the configuration possibilities and meanings, please have a look to the following how to's:

- How to enable the eHF LDAP integration in an eHF-based assembly
- How to enable PXS specific LDAP integration functionality

15.11.4 Dependencies

15.11.4.1 Internal

The eHF LDAP identity module has dependencies to following eHF modules:

- eHF Commons Identity
- eHF Commons
- eHF Authentication

The Maven2 pom:

```
<dependency>
    <groupId>com.icw.ehf</groupId>
    <artifactId>ehf-commons</artifactId>
    <version>${project.version}</version>
    <scope>test</scope>
    <classifier>runtime</classifier>
</dependency>
<dependency>
    <groupId>com.icw.ehf</groupId>
    <artifactId>ehf-commons-identity</artifactId>
    <version>${project.version}</version>
    <scope>compile</scope>
    <classifier>runtime</classifier>
</dependency>
<dependency>
    <groupId>com.icw.ehf</groupId>
    <artifactId>ehf-authentication</artifactId>
    <version>${project.version}</version>
    <scope>compile</scope>
    <classifier>api</classifier>
</dependency>
```

15.11.4.2 External

The eHF LDAP Identity module has following dependencies to third party libraries:

- SpringLDAP
- ApacheDS (for tests)

The Maven2 pom:

```
<dependency>
    <groupId>org.springframework.ldap</groupId>
    <artifactId>spring-ldap-core</artifactId>
    <version>1.3.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.sun</groupId>
    <artifactId>ldapbp</artifactId>
    <version>1.0</version>
</dependency>
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-all</artifactId>
    <version>1.5.6</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.apache.directory.client.ldap</groupId>
```

```
        <artifactId>ldap-client-api</artifactId>
    </exclusion>
    <exclusion>
        <groupId>org.apache.directory.shared</groupId>
        <artifactId>shared-ldap-schema</artifactId>
    </exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>org.apache.directory.server</groupId>
    <artifactId>apacheds-server-integ</artifactId>
    <version>1.5.6</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId> org.apache.directory.client.ldap</groupId>
            <artifactId>ldap-client-api</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.directory.shared</groupId>
            <artifactId>shared-ldap-schema</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

16 Infrastructure Modules

16.1 Terminology

16.1.1 Basic Architectural Principles

In eHF the semantic of medical information is represented with codes. In order to work with codes several aspects have to be considered:

- The semantics must be structured into codes and related concepts.
- The concepts must be acquirable and searchable.
- The information must be persisted and bound to an observation of some kind.

The retrieval of codes and concepts as well as the structuring of this information is provided by the Code Services. The key features of the API are listed in the following sections.

16.1.1.1 Design Goals

We applied best practices from domain-driven design like mapping the domain knowledge into the code so that the central concepts about the terminology domain are still recognizable in the source code. In other words concepts like code systems and their properties, for example the object identifier (oid), are named accordingly. See the section [Identification of Concepts](#) on page 239 .Further we have followed a service oriented approach, leading to a flexible but still easy to use API. The services are named after their primary purpose in the class name. For example, the finding of concepts is provided by the `CodeFinderService` . Another important part of the above mentioned service focus is the organization of the data passed to and received from the services. These objects are mere data objects that either identify a code concept or carry parameter value but do not model relationships. For example all codes of a code system are obtained by calling the `CodeFinderService.find(..)` method instead of loading a code system and accessing a getter to retrieve the codes.From the beginning we have emphasized performance aspects, because some catalogues are huge and the combinations of search parameters lead to complex statements.

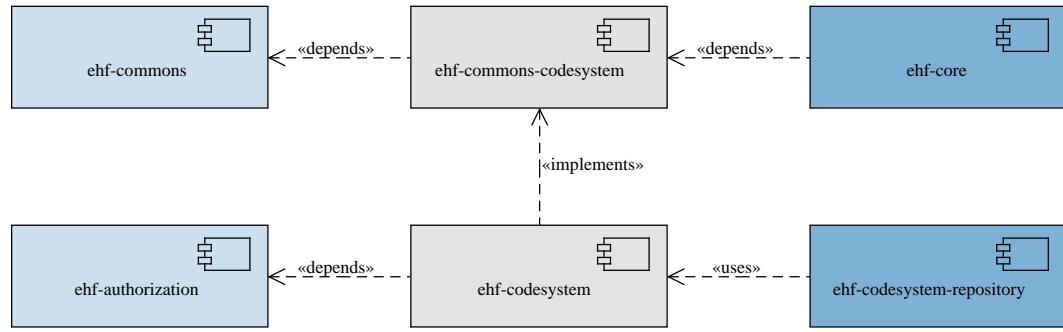
16.1.1.2 Persistence of Coded Information

Depending on the usage of the terminology functionality, persistence occurs in different ways:

- The codes and concepts are stored for retrieval purposes within a database. The model and entities for this data is persisted by the `ehf-codesystem` module. A dedicated importer parses the XML representation of the code concepts and stores the data in the database using common CRUD Services. Such an import is performed before the deployment of an application. See the section [Code Content Repository](#) on page 239 for more information.
- For the modification of the code concepts during runtime, a dedicated service, the `CodeAuthoringService` , is provided by the Public API. In the section [Authoring of Codes at Runtime](#) on page 248 the details are explained.
- The semantic information of a domain object (for example the type of a diagnosis) is persisted as a coded attribute (`Code` , `EmbeddedCode`). This functionality is provided by the `ehf-core` module. See the section [Coded Attributes](#) on page 108 for more information.

16.1.1.3 Modules and Packages

The implementation of the code concepts within eHF is distributed among several modules.

**Figure 98:** Module Runtime Dependency

16.1.1.3.1 ehf-commons-codesystem:

The *ehf-commons-codesystem* module defines the [Contracts of the Public API](#) on page 238 for retrieving, searching and accessing the code concepts. Amongst other features, it defines:

- Service signatures for the code services (`CodeFinderService`, ...)
- Required data for the identification of code concepts (`CodeQualifier`, ...)
- Criteria for the parametrization of queries (`CodeCriteria`, ...)
- Containers for handling result sets.

The following packages can be found in the *ehf-commons-codesystem* module.

Package	Description
criterions	Defines the criteria parameter for the services.
exception	Defines the exception definitions for the services.
identifier	Interfaces and abstract classes describing necessary information for identification of codes.
qualifier	Specifies a hierarchy for qualifying codes and code collections.
service	Contains the container and parameter definitions for the services.
test	Support classes and mocks for using code services in tests.
transfer	Describes wrapper classes for the response of the services.
util	Contains helper classes.

Table20. Packages of *ehf-commons-codesystem*

16.1.1.3.2 ehf-codesystem:

The *ehf-codesystem* module provides runtime classes for the interfaces defined in *ehf-commons-codesystem* and the persistence of the code content repository. It implements:

- The code services (`CodeFinderServiceImpl`, `FinderDao`, ...)
- A data model for the persistence of the code concepts (Generator/UML model, ...)
- A data representation of codes (XML schema definition)

- An importer for copying the XML representation into the database model (`CodeImportProcessor`, ...).

This module depends on some external beans which are stated in the import section of the Spring module context file.

Bean	Interface
securityService	com.icw.ehf.authorization.service.SecurityService
transactionManager	org.springframework.transaction.PlatformTransactionManager
sessionFactory	org.hibernate.SessionFactory
hibernateTemplate	org.springframework.orm.hibernate3.HibernateOperations

Table21. Required Beans

The following beans are provided by the module and are defined in the export section.

Bean	Interface	Function
bootstrapHandler	com.icw.ehf.codesystem.repository.handler.Handler	Bootstrap
reverseTransformatorHandle	com.icw.ehf.codesystem.repository.handler.Handler	Bootstrap
validatorHandler	com.icw.ehf.codesystem.repository.handler.Handler	Bootstrap
extensionValidatorHandler	com.icw.ehf.codesystem.repository.handler.Handler	Bootstrap
repositoryProcessor	com.icw.ehf.codesystem.repository.processor.Processor	Bootstrap
codeResolverService	com.icw.ehf.commons.codesystem.CodeResolverService	Service
codeFinderService	com.icw.ehf.commons.codesystem.CodeFinderService	Service
codeValidatorService	com.icw.ehf.commons.codesystem.CodeValidatorService	Service
codeAliasService	com.icw.ehf.commons.codesystem.CodeAliasService	Service
codeAuthoringService	com.icw.ehf.commons.codesystem.CodeAuthoringService	Service

Table22. Provided Beans

Using the exported beans requires the import of the module context in your Spring configuration file (for example in your assembly).

```
<import resource="classpath:/META-INF/ehf-codesystem-module-context.xml" />
```

16.1.1.3.3 ehf-codesystem-repository:

This module contains the XML representation of the code systems and code sets required by eHF. Further it defines a file named 'import-filter' which determines the categories that

are imported into the database. The section [Code Content Repository](#) on page provides further information.

16.1.1.3.4 ehf-core:

The *ehf-core* module allows the storage of coded information as attributes of domain objects (like the confidentiality code of a Medication). It provides:

- Hibernate mappings for codes as embedded entities (*Code* , *EmbeddedCode* , ...)
- Helper classes to determine the range of codes for a coded attribute (*CodedAttributeQualifier* , *CodedAttributeService* , ...).

See the section [Coded Attributes](#) on page 108 for further details.

16.1.1.3.5 3rd Party Libraries:

The following table shows the required 3rd Party libraries. Only those dependencies that are introduced by the modules are listed. Further dependencies for tasks like transaction handling and database drivers need to be provided as well.

dependency	<i>ehf-commons-codesystem</i>	<i>ehf-codesystem</i>	<i>ehf-codesystem-repository</i>
spring-2.5.6.jar	required	required	transitive
commons-collections-3.2.jar	required	required	transitive
commons-logging-1.1.1.jar	required	required	transitive
dom4j-1.6.1.jar		required	transitive
jaxen-1.1.1.jar		required	transitive
xstream-1.2.2.jar		required	
xpp3_min-1.1.4c.jar		required	
ehcache-1.2.4		required	transitive
ejb3-persistence-1.0.0.jar		required	transitive
hibernate-3.2.6.ga-icw-p3.jar		required	transitive
hibernate-annotations-3.3.0.ga.jar		required	transitive
hibernate-commons-annotations-3.3.0.ga.jar		required	transitive
antlr-2.7.6rc1.jar		required	transitive

Table23. 3rd Party Libraries

16.1.2 Contracts of the Public API

In general, the Code Services API provides methods for accessing controlled vocabularies and classifications using the aforementioned concepts. The API, amongst others things, is used for validation purposes, ensuring the consistency of the overall content stored in the system and supports the demands on localization, internationalization and general customization. The service contract is defined in the *ehf-commons-codesystem* module whereas the *ehf-codesystem* module is a concrete implementation of this contract. The classifications and vocabularies are stored in the *ehf-codesystem-repository*. See the [Code Content Repository](#) on page for more details.

The Code Services API contains several public service interfaces:

- `CodeAliasService` Register and lookup of aliases for code systems and code categories
- `CodeAuthoringService` Create or extend code systems and code sets, modify code categories
- `CodeFinderService` Find code concepts and their related aspects
- `CodeResolverService` Retrieve codes and their display-values in different translations
- `CodeValidatorService` Validate code concepts against the application content

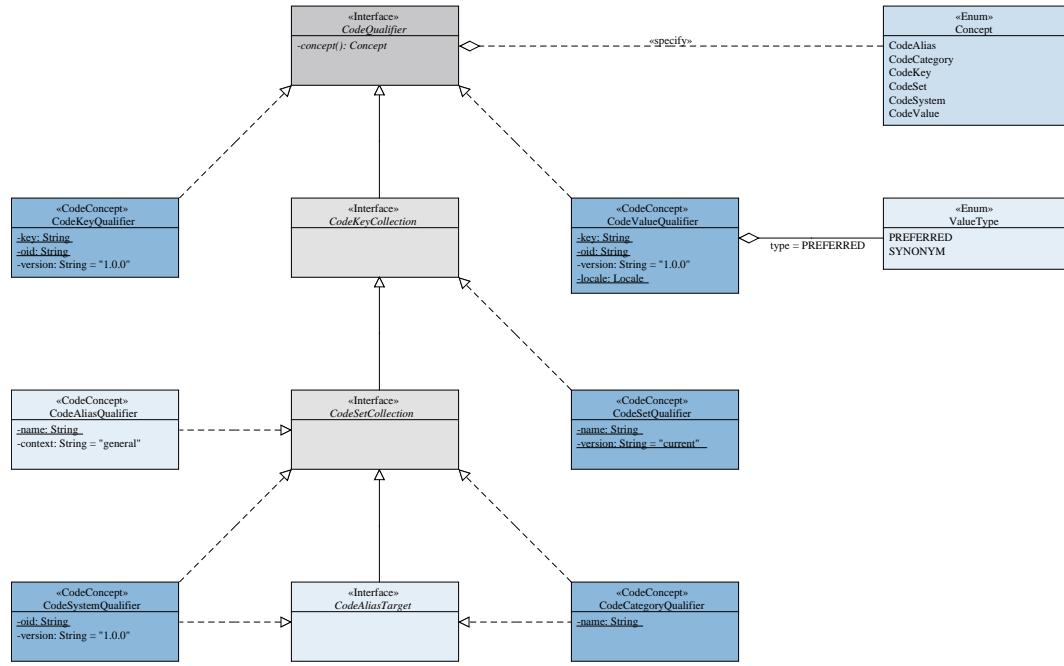
Before we address the details of the service methods we will give a brief introduction to the system contracts. Invoking a Code Service requires passing a code concept as an argument. The section [Identification of Concepts](#) on page 239 provides an overview of the identifiable concepts. Once the concept is determined, you may continue with the [Retrieval of Concepts](#) on page 241 or by [Using Concepts via Aliases](#) on page 247 to define your own identifier of a code concept. If concepts are missing, there is the possibility of [Authoring of Codes at Runtime](#) on page 248. Also you may do a [Validation of Concepts](#) on page 251. [Further Service Aspects](#) on page 252 provides additional information on cross-cutting concerns. The section [Handling defective Requests](#) on page 255 explains service exception. Finally [Examples for using the API](#) on page 256 are given.

16.1.2.1 Identification of Concepts

Every code concept is determined by some sort of `CodeQualifier`. This is an important aspect since it is used to address concepts in the system. The available concepts can be found in the `qualifier` package of `ehf-commons-codesystem`. The simplest concept is a single code that is represented by a `CodeKeyQualifier`. A code set is represented by a `CodeSetQualifier` which in turn represents a `CodeKeyCollection`. In the same way a code system is represented by a `CodeSetCollection` because code sets refer to a code system that is also a `CodeKeyCollection`. And finally a code category is identified by a `CodeCategoryQualifier` which is also represented by a `CodeSetCollection` since a code category is defined as a union of code sets.

16.1.2.1.1 CodeQualifier :

The `CodeQualifier` identifies a concept but does not reference a related concept. For example the codes of a code set can not be acquired by calling some method of a `CodeSetQualifier`. Such a functionality is provided by the Code Services.

**Figure 99:** CodeQualifier Hierarchy

Concept Identifier	Attributes	Description
CodeQualifier		Every code concept is determined by some sort of CodeQualifier .
CodeKeyQualifier	oid, version (opt.), key	Determines a specific code.
CodeKeyCollection		Every concept that specifies one or more codes is determined by some sort of CodeKeyCollection .
CodeSetQualifier	name, version	Determines a specific code set.
CodeSetCollection		Every concept which specifies one or more code sets is determined by some sort of CodeSetCollection .
CodeSystemQualifier	oid, version (opt.)	Determines a specific code system.
CodeCategoryQualifier	name	Determines a specific code category.
CodeAliasQualifier	alias, context (opt.)	Determines an alternative identifier for a code concept that is marked as a CodeAliasTarget .
CodeAliasTarget		Currently aliases for the concepts code

Concept Identifier	Attributes	Description
		system and code category are supported.

Table24. Overview of CodeQualifier

A CodeQualifier is created by calling the static method `create` with the desired attributes. For example to create a CodeSystemQualifier use the following code snippet:

```
String oid = "1.2.3.4";
String version = "2009"; // optional, the default is "1.0.0"

CodeSystemQualifier codeSystem = CodeSystemQualifier.create(oid, version);
```

A CodeQualifier returns a textual representation of its content by calling the according method. The returning string is a concatenation of the underlying concept and qualifier attributes delimited by the pipe ('|') character.

```
String textRepresentation = codeSystem.textRepresentation(); // = "CodeSystem|1.2.
3.4|2009"
```

Noteworthy is the fact that all CodeQualifier override the `hashCode` and `equals` methods in such a way, that different instances of a CodeQualifier with the same content are considered equal. Even the identity is ensured, that means every identified concept exists only once.

```
CodeSystemQualifier codeSystem1 = CodeSystemQualifier.create("oid1");
CodeSystemQualifier codeSystem2 =
CodeSystemQualifier.create("oid1");

assertEquals(codeSystem1, codeSystem2);

assertTrue(codeSystem1 == codeSystem2);
```

16.1.2.2 Retrieval of Concepts

The following code snippet provides an example for finding codes within a collection of codes.

```
// Determine a CodeKeyCollection
CodeKeyCollection codeKeyCollection = CodeSetQualifier.create("name");

// Specify search parameters
CodeCriteria<Find> codeCriteria = CodeCriteria.createFindCriteria("search term");

// Trigger the find request
CodeContainer<CodeKeyQualifier> container = codeFinderService.
find(codeKeyCollection, codeCriteria);
```

This section will address the different aspects of the previous code lines.

16.1.2.2.1 CodeFinderService :

The finder services provide functionality to browse the relation between code concepts, for example, find all codes within a code category. All return values of these services contain

sufficient information to qualify the code concepts but do not provide further information like the translations for codes.

Service	Parameter	Result	Description
find	CodeKeyCollection , CodeCriteria<Find>	CodeContainer<CodeKeyQualifier>	Finds codes within a collection of codes.
findCodeSets	CodeKeyCollection	List<CodeSetQualifier>	Finds all code sets representing a collection of codes.
findCodeSets	CodeCategoryQualifier , AssignmentStatus	List<CodeSetQualifier>	Finds all code sets with the given status for a code category.
findCodeSystem	CodeSetQualifier	CodeSystemQualifier	Finds the parent code system of a code set.
findCurrentCodeSet	CodeSetQualifier	CodeSetQualifier	Finds the current version of a code set.
findCodeCategories	void	List<CodeCategoryQualifier>	Finds all code categories available in the system.
findCodeSets	void	List<CodeSetQualifier>	Finds all code sets and all their versions that are available in the system.
findCodeSystems	void	List<CodeSystemQualifier>	Finds all code systems available in the system.

Table25. CodeFinderService

For example to find all active code sets for a code category use:

```
CodeKeyCollection codeKeyCollection = CodeCategoryQualifier.create("name");
List<CodeSetQualifier> codeFinderService.findCodeSets(codeKeyCollection,
AssignmentStatus.ACTIVE);
```

Finding all code sets for that code category independently of their AssignmentStatus is done using the null argument.

```
List<CodeSetQualifier> codeFinderService.findCodeSets(codeKeyCollection, null);
```

16.1.2.2.2 CodeResolverService :

These services offers the retrieval of code description and their translation respectively. Further it provides search capabilities on the key entries and the coded value terms.

Service	Parameter	Result	Description
resolve	CodeKeyQualifier , Locale	CodedValue	Returns a CodedValue object containing the preferred value for the given code.
resolve	CodeKeyCollection , CodeCriteria<Resolve>	CodeContainer<CodedValue>	Returns an array of all codes belonging to the given CodeKeyCollection and criteria.
search	CodeKeyCollection , CodeCriteria<? super Search>	CodeContainer<CodedValue>	Queries the system for codes and translations which adhere to the given Search criteria.

Table26. CodeResolverService

For example to obtain all codes of a code set with a US translation use:

```
CodeKeyCollection codeKeyCollection = CodeSetQualifier.create("name");
CodeCriteria<Resolve> codeCriteria = CodeCriteria.createResolveCriteria(Locale.US);
CodeContainer<CodedValue> container = codeResolverService.resolve(codeKeyCollection, codeCriteria);
```

16.1.2.2.3 Common Signature for the Retrieval:

All services, which yield a collection of codes as a result follow a similiar service signature:

```
CodeContainer<? extends CodeIdentifier> nameOfService ( CodeKeyCollection,
CodeCriteria<? extends Query> )
```

Currently three service methods use this signature:

1. CodeResolverService.resolve \- Retrieves all codes and translation
2. CodeResolverService.search \- Retrieves all codes and translation matching a given search pattern
3. CodeFinderService.find \- Retrieves all codes matching a given search pattern

The next sections explain the parameter types in detail.

16.1.2.2.4 Restrict and Control the Service Output:

The CodeCriteria allows the parameterization of the services listed in the previous section. For example, you can determine the localization of the returned descriptions or specify search terms. The CodeCriteria comes with a default value configuration which can be adapted, in order to restrict the result set of a standard query. The available settings are explained in the next table using the following column captions:

Criterion Name of the criterion **Parameter** Parameter of a criterion **Type** Data type of a parameter **Range** Value range of a parameter **Default** Default settings provided by the system (may be overwritten) **Description** Additional information for a criterion or its parameter

Criterion	Parameter	Type	Range	Default	Description
SearchTerm	pattern	String		%	'
	options	EnumSet	CASE_SENSITIVE	Ø	
SearchType	searchTypes	EnumSet	BY_KEY, BY_VALUE	Ø	
	locale	Locale	'	de	
SubSystem	valueTypes	EnumSet	PREFERRED, SYNONYM	Ø	
	identifier	String		null	deprecated
Translation	traversStructure	boolean		true	deprecated
	locale	Locale		de	
Sort	fallback	Enum	BY_COUNTRY, BY_LANGUAGE, BY_DEFAULT, NO_FALLBACK	BY_DEFAULT	Interpretation of the locale.
	type	Enum	BY_KEY, BY_VALUE, SEMANTIC	BY_KEY	
ValueType	order	Enum	ASCENDING, DESCENDING	ASCENDING	
	valueTypes	EnumSet	PREFERRED, SYNONYM	PREFERRED	
Range	index	int	0 ... n	0	
	length	int	\-1 ... n	100	

Table27. Overview of Code Criteria

Depending on the service only a subset of criteria is useful. Therefore the CodeCriteria is a generic class and the respective subset is defined by one of the following Query types:

- Resolve Criterion relevant in the CodeResolverService.resolve service
- Search Criterion relevant in the CodeResolverService.search service
- Find Criterion relevant in the CodeValidatorService.find service

The applicability of a criterion for a service is given in the next table. The required column states whether a criterion is required. If the request is missing a mandatory criterion, the service will create one with its parameters set to the default values.

Criterion	resolve	search	find	required
SearchTerm	✗	✓	✓	optional
SearchType	✗	✓	✗	mandatory
SubSystem	✓	✓	✓	optional
Translation	✓	✓	✗	mandatory

Criterion	resolve	search	find	required
Sort	✓	✓	✓	optional
ValueType	✓	✓	✗	mandatory
Range	✓	✓	✓	optional

Table28. Application of Code Criteria

A criterion is created by passing the desired values to the constructor - for other settings the defaults are used respectively.

```
TranslationCriterion translation = new TranslationCriterion(Locale.GERMANY);
```

Once the single criterion is created, they are collected in the CodeCriteria object which takes the criterions as constructor arguments. Also the matching generic type must be determined.

```
CodeCriteria<Resolve> codeCriteria = new CodeCriteria<Resolve>(translation);
```

The CodeCriteria provides some static create methods for common parameters. For example the previous code lines can be shortened:

```
CodeCriteria<Resolve> codeCriteria = CodeCriteria.createResolveCriteria(Locale.US);
```

According to the table above the following code snippet will not compile.

```
// A TranslationCriterion cannot be used in the find(..) method
CodeCriteria<Find> codeCriteria = new CodeCriteria<Find>(translation);
```

One major achievement of this parameter container is the flexibility of its configuration. With the settings it is possible to adapt the default configuration of the CodeCriteria values even system-wide.

Exemplarily the SearchTypeCriterion is explained here. It defines the data elements (database columns) to which a search term is applied. The possible settings and their effects are listed in the following table.

searchTypes	valueTypes	locale	search in C_KEY	search in C_VALUE	search in C_VALUETYPE
∅	∅	as given	✓	✓	PREFERRED, SYNONYM
∅	PREFERRED	as given	✓	✓	PREFERRED
∅	SYNONYM	as given	✓	✓	SYNONYM
BY_KEY	∅	as given	✓	✗	PREFERRED
BY_KEY	PREFERRED	as given	✓	✗	PREFERRED
BY_KEY	SYNONYM	as given	✓	✗	PREFERRED

searchTypes	valueTypes	locale	search in C_KEY	search in C_VALUE	search in C_VALUETYPE
BY_VALUE	Ø	as given	✗	✓	PREFERRED, SYNONYM
BY_VALUE	PREFERRED	as given	✗	✓	PREFERRED
BY_VALUE	SYNONYM	as given	✗	✓	SYNONYM

Table29. SearchTypeCriterion

See the CodeCriteria and the package criterions in *ehf-commons-codesystem* for more details.

16.1.2.2.5 Container-wrapped Result Set:

The result of the service request is wrapped in a CodeContainer , in order to deliver a comprehensive and complete response. The returned container contains not only the requested set of concepts - being restricted with the CodeCriteria argument, but also the amount of concepts that comply to the request but were filtered due to a range restriction (RangeCriterion). Overall the CodeContainer object encapsulates:

- The CodeQualifier of the desired concept
- The CodeCriteria with the actual query criteria
- A result set of CodelIdentifier objects
- The number of codes before applying any range filtering

The CodelIdentifier interface defines the necessary properties for a code to be unique. They are the same as in the Code concept and the CodeKeyQualifier respectively. Currently there are two implementations of the CodelIdentifier interface which serve as result objects for the finder and resolver service:

- CodeKeyQualifier \ Identifier for a code key concept.
- CodedValue \ Transfer object which describes a resulting ResolvedCode and additionally a value, its type and localization.

The benefits are that the context information of the request is bound to the results (original parameters \!= effective parameters). Both the default values and locale fallback scenarios can be obtained (for example, en_US \> en \> de). You can do the paging of results together with a RangeCriterion .

```
// Restrict the result set to 200 entries
int idx = 0;
int len = 200;
RangeCriterion rangeCriterion = new RangeCriterion(idx, len);
CodeCriteria<Find> codeCriteria = new CodeCriteria<Resolve>(rangeCriterion);

// Get the first 200 codes of a code system
CodeContainer<CodeKeyQualifier> container = codeFinderService(codeSystemQualifier,
    codeCriteria);

// Check if the code system has more than 200 codes
int numResultingCodes = container.numCodes();
int numAvailableCodes = container.numAvailableCodes();

if (numAvailableCodes > numResultingCodes) {

    // Get the next 200 codes
    idx += len;
    rangeCriterion = new RangeCriterion(idx, len);
    container = codeFinderService(codeSystemQualifier, codeCriteria);
}
```

16.1.2.3 Using Concepts via Aliases

Aliases are used as alternative identifiers. For example instead of using an object identifier directly via a CodeSystemQualifier it is possible to register an alias for that particular concept. Often an alias provides better readability. The available services for handling aliases are

- Register an alias for a code system or a code category
- Look up specific aliases
- Retrieve all aliases known to the system

An alias is represented by a CodeAliasQualifier which takes an alias name and an optional context information. The context serves as a name space where the alias name is unique. A CodeAliasQualifier can be registered for a code concept that implements the CodeAliasTarget marker interface. Currently these are all children of the CodeSetCollection which is either a CodeSystemQualifier or a CodeCategoryQualifier . The following code snippet creates an alias and registers it for a code category:

```
CodeCategoryQualifier codeCategory = CodeCategoryQualifier.create("CategoryName");

CodeAliasQualifier codeAlias = CodeAliasQualifier.create("AliasName",
    "AliasContext");

codeAliasService.register(codeAlias, codeCategory);
```

The alias creation and its code category registration were successful when no exception was thrown. You can lookup the underlying concept - bound to the alias - by calling:

```
CodeAliasTarget aliasTarget = codeAliasService.lookup(codeAlias);
```

You may also use the CodeAliasQualifier for other services wherever they expect a CodeSetCollection as an argument. See the section [Alias Lookup](#) on page 254 for more.

If the CodeAliasQualifier is missing the context property, the default context is assumed. The default context is named "GENERAL" and allows the registration of several different alias names for the same concept. This is not allowed if non-default contexts are given as they take only one alias for a specific code concept. Therefore the call of the CodeAliasService in the next code snippet fails with an InvalidConceptException .

```
CodeAliasQualifier codeAliasTwo = CodeAliasQualifier.create("AliasNameTwo",
    "AliasContext");

try {
    // The register service will throw an InvalidConceptException
    codeAliasService.register(codeAliasTwo, codeCategory);
}
catch(InvalidConceptException e) {
    // Only one alias per context allowedIn the context 'AliasContext'
}
```

16.1.2.3.1 CodeAliasService :

This service defines methods for the administration of aliases for a CodeAliasTarget . As described before, only one alias per context is allowed for a target. This restriction does not apply to the default respectively empty context.

Service	Parameter	Result	Description
register	CodeAliasQualifier , CodeAliasTarget	void	Register an alias for a CodeAliasTarget .
unregister	CodeAliasQualifier	void	Unregister an alias.
lookup	CodeAliasQualifier	CodeAliasTarget	Retrieve the CodeAliasTarget for the given alias.
find	CodeAliasTarget	List<CodeAliasQualifier>	Retrieve all aliases register for the given CodeAliasTarget .
find	CodeAliasTarget , String	List<CodeAliasQualifier>	Retrieve all aliases register for the given CodeAliasTarget for an alias context.
find	Class<T extends CodeAliasTarget>	Map<CodeAliasQualifier, T>	Retrieve all aliases registered for the concept given by the CodeAliasTarget type.
find	Class<T extends CodeAliasTarget> , String	Map<CodeAliasQualifier, T>	Retrieve all aliases registered for the concept given by the CodeAliasTarget type and an alias context.

Table30. CodeAliasService

Finding all aliases that are registered for a code system can be obtained this way:

```
Map<CodeAliasQualifier, CodeSystemQualifier> aliasesForCodeSystems =
    codeAliasService.find(CodeSystemQualifier.class);
```

16.1.2.3.2 Predefined Aliases:

The import process of the *ehf-codesystem* modules creates an alias in the "NAME" context for a code system by using the name given in the XML definition file. Therefore you might use a CodeAliasQualifier as a convenient notation instead of the object identifier's numerical series. The codesystem name can be retrieved by filtering the context attribute of already loaded code system aliases or by loading them directly like this:

```
Map<CodeAliasQualifier, CodeSystemQualifier> aliasesForCodeSystems =
    codeAliasService.find(CodeSystemQualifier.class, "NAME");
```

16.1.2.4 Authoring of Codes at Runtime

The code base of an application can be managed and altered during runtime by using the CodeAuthoringService . The services provide methods for:

- Creating and extending a code system
- Creating and extending a code set
- Assigning code sets to code categories.

The modification of code sets is subject to a versioning control. Only the current version of a code set can be altered. The service always creates a new version of a code set whenever a code is added or removed from it. Therefore the latest version is also the current version. Currently the following version scheme is applied:

- The version number consists of three digits separated by a dot (x.y.z).
- A new code set starts with the version (1.0.0).
- The addition of codes to a code set increases the *minor* number by one (1.y.0).
- The removal of codes results in an increase of the *major* number by one and the resetting of the *minor* and *revision* numbers to zero (x.0.0).
- The *revision* number is currently unused (1.0.z).

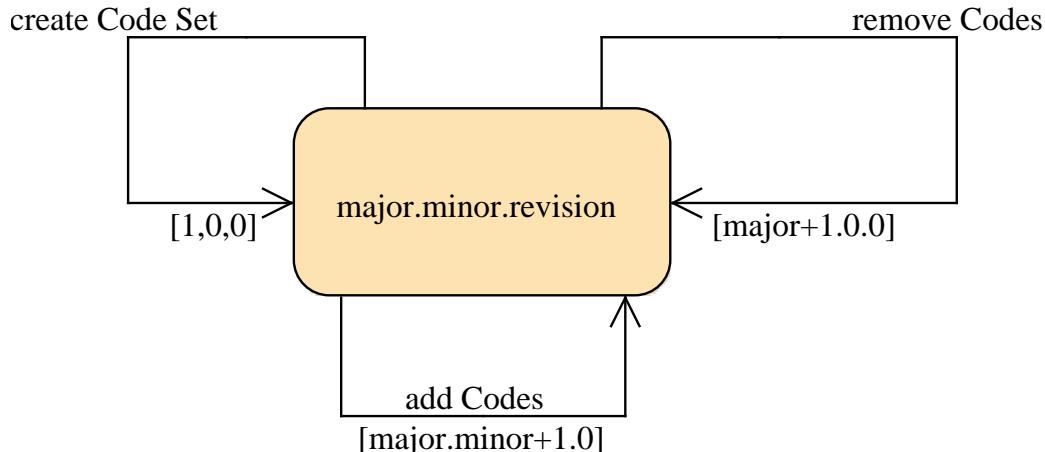


Figure 100: Code Set Versioning

Further restrictions are:

- A code set and all its versions can only be taken from one code system and oid respectively. This means a change of the code set to another code system requires a different code set name.
- A code set is assigned to a code category based on the version. In case some version of the code set is not assigned explicitly, then its codes are not part of the code category.
- The assignment of a code set version to code category always has a status. See the AssignmentStatus for possible settings.

16.1.2.4.1 CodeAuthoringService :

The service provides the following methods

Service	Parameter	Result	Description
create	CodeSystemQualifier	void	Creates a new code system.
create	Set<CodedValue>	Set<CodedValue>	Creates new codes and their values.
update	Set<CodedValue>	Set<CodedValue>	Updates the values of existing code values.
create	CodeSetQualifier , CodeSystemQualifier	CodeSetQualifier	Creates a new code set adding

Service	Parameter	Result	Description
			all codes of the given code system.
create	CodeSetQualifier , Set<CodeKeyQualifier>	CodeSetQualifier	Creates a new code set version adding all given codes.
add	CodeSetQualifier , Set<CodeKeyQualifier>	CodeSetQualifier	Extends the code set with the given collection of codes.
remove	CodeSetQualifier , Set<CodeKeyQualifier>	CodeSetQualifier	Removes the given collection of codes from a code set.
assign	CodeSetQualifier , AssignmentStatus , CodeCategoryQualifier	CodeSetQualifier	Assigns the given code set to the given code category.

Table31. CodeAuthoringService

See the javadoc for detailed information. For example the service for adding codes to a code set ignores those codes that already exist. The remove method behaves analogous.

Creating a new code set based on a new code system requires the following steps:

```
// Create code system qualifier
CodeSystemQualifier codeSystem = CodeSystemQualifier.create("oid", "version");

// Create code values
CodedValue codedValue1 = CodedValue.create(codeSystem, "key1", ValueType.
PREFERRED, Locale.US, "value1");
CodedValue codedValue2 = CodedValue.create(codeSystem, "key2", ValueType.
PREFERRED, Locale.US, "value2");

// Create set of code values
Set<CodedValue> codedValues = new HashSet<CodedValue>();
codedValues.add(codedValue1);
codedValues.add(codedValue2);

// Create the code system in the DB
codeAuthoringService.create(codedValues);

// Create code set qualifier
CodeSetQualifier codeSet = CodeSetQualifier.create("name");

// Create the code set in the DB with all codes of the given code system
codeAuthoringService.create(codeSet, codeSystem);
```



CAUTION: Attention

The altering of the code base using this service might invalid the results of previous requests. So please be careful when caching code content locally.

16.1.2.4.2 Logging:

All changes to the code base are logged in an internal journal. Currently the userId, the timestamp, the service and its given parameter and the results are stored respectively. An error during the persistence of the journal data results in a CodeServiceException but does not rollback the actual authoring changes.



Important: Note

There is no service available to the user to retrieve the journal. Therefore a database administrator is required to track changes to the code content.

16.1.2.4.3 Authorization:

As changes to the code base alters the outcome of a service request and the system behavior, unauthorized access is intercepted and results in a java.security.AccessControlException .The authoring process requires execution rights to the services. The available targets can be specified by using the full classname of the CodeAuthoringService.Qualifier constants. These constants implement the CodeServiceQualifier interface which specifies a method to retrieve the declaring service class. For the CodeAuthoringService there is a target for every service method (the target name aligns with the method name and parameters).

- createCodeSystem ,
- createCodeValue ,
- updateCodeValue ,
- createCodeSetImplicit , // all codes of the code system
- createCodeSetExplicit , // given set of codes
- addCodesToCodeSet ,
- removeCodesFromCodeSet ,
- assignCodeSetToCodeCategory ;

A permission for a code author, which grants access to each method, would look like

```
<permission>
  <target>
    <type>com.icw.ehf.commons.codesystem.CodeAuthoringService.Qualifier</type>
    <role>*</role>
    <context>*</context>
    <identifier>*</identifier>
  </target>
  <actions>
    <action>EXECUTE</action>
  </actions>
</permission>
```

Because the *ehf-codesystem* module is using the com.icw.ehf.authorization.service.SecurityService , it depends on *ehf-authorization*. See the respective documentation for more details like the user creation and the permission import.

16.1.2.5 Validation of Concepts

In order to find out whether a code is a part of a code set, all codes should be retrieved and checked, if the code is part of the result set. There is an easier way using the CodeValidatorService .

```
CodeSetQualifier codeSet = CodeSetQualifier( "name" );
CodeKeyQualifier codeKey = CodeSetQualifier( "key" , "oid" );
boolean isElement = codeValidatorService.elementOf(codeKey, codeSet);
```

This service is used by the eHF CRUD Services internally to verify that coded attributes are within the specified range.

16.1.2.5.1 CodeValidatorService :

The validator services provide methods for the verification of relations between concepts or the actual existence of a concept. Therefore all services return a boolean value.

Service	Parameter	Result	Description
exists	CodeQualifier	boolean	Validates if the given CodeQualifier is known to the system.
elementOf	CodeKeyQualifier , CodeKeyCollection	boolean	Validates if the given CodeKeyQualifier is an element of the CodeKeyCollection .
elementOf	CodeSetQualifier , CodeSetCollection	boolean	Validates if the given CodeSetQualifier is an element of the CodeSetCollection .
elementOf	CodeSystemQualifier , CodeCategoryQualifier	boolean	Validates if the given CodeSystemQualifier is an element of the CodeCategoryQualifier .
isAssigned	CodeSetQualifier , CodeCategoryQualifier	boolean	Validates if the given CodeSetQualifier is assigned to the category CodeCategoryQualifier .
isAssigned	CodeSetQualifier , CodeCategoryQualifier , AssignmentStatus	boolean	Validates if the given CodeSetQualifier within the category CodeCategoryQualifier corresponds to the specified status AssignmentStatus .
isCurrent	CodeSetQualifier	boolean	Validates if the version of the given CodeSetQualifier is the current one.

Table32. CodeValidatorService

16.1.2.6 Further Service Aspects

The Code Services are equipped with some additional functionality:

- **Caching Results** \- Caches the service results
- **Alias Lookup** \- Transforms an alias into the associated code concept
- **Input Validation** \- Inspects the input parameter for validity
- **Journal of Changes** \- Logs changes to the code base
- **Guarding Services** \- Intercepts unauthorized execution of services
- **Transaction Propagation** \- All services support transactions, some require a transaction and will create one if necessary.

These cross-cutting concerns are implemented by decorating the respective functionality around the actual service which is shown in the next picture.

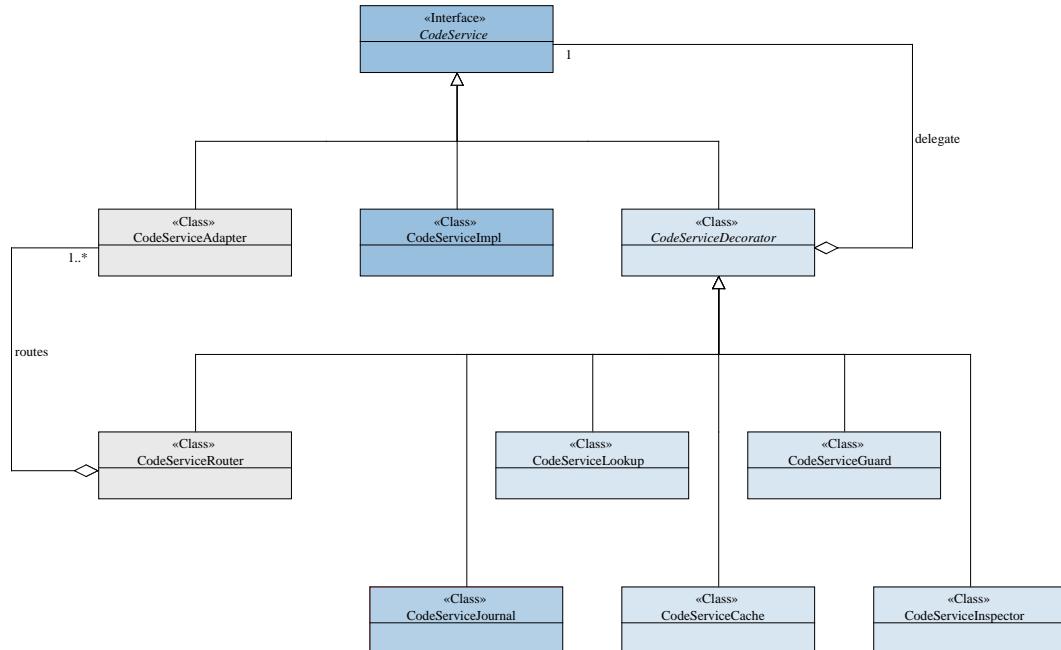


Figure 101: Code Service Decorator

The *ehf-codesystem* makes use of some of these aspects and decorates its implementation services. The following table shows the beans provided by the module context and their current configuration:

Bean	Caching Results	Alias Lookup	Input Validation	Journal of Changes	Guarding Services	Transaction Propagation
codeAliasService	✗	✗	✓	✗	✗	required (some services)
codeAuthoringService	✗	✗	✓	✓	✓	required (all services)
codeFinderService	✓	✓	✓	✗	✗	supports
codeResolverService	✓	✓	✓	✗	✗	supports

Bean	Caching Results	Alias Lookup	Input Validation	Journal of Changes	Guarding Services	Transaction Propagation
codeValidatorService	✓	✓	✓	✗	✗	supports

Table33. Decoration of Code Services

This means for example you may use the CodeValidatorService by injecting the codeValidatorService bean into your program code. According to the previous table, the communication with the service is then memorized by [Caching Results](#) on page 254, an [Input Validation](#) on page 254 of the parameters executed and if necessary an [Alias Lookup](#) on page 254 performed.

16.1.2.6.1 Caching Results:

Currently the results of the CodeValidatorService can be cached with ehcache. This will help to reduce the database hits when validating domain objects, that use coded attributes, against their metadata. In order to enable the cache the existence of the configuration file ("META-INF/ehcache-codesystem.xml") is required in the classpath. The results are stored in a cache named with the full classname of the respective service (for example com.icw.ehf.common.codesystem.CodeValidatorService). The cache is cleared completely when the code base is altered by the CodeAuthoringService .See more in the "[How to enable caching for Code Validation Service](#)" description.

**CAUTION:** Attention

Please make sure that the clearing of the ehcache is properly forwarded to the other nodes within a clustered deployment.

16.1.2.6.2 Alias Lookup:

The alias lookup feature supports the use of a CodeAliasQualifier instead of the actual concept. For example, if an alias is passed to the findCodeSets method of the CodeFinderService the respective CodeFinderServiceLookup decorator intercepts the call and executes the lookup method of the CodeAliasService and forwards the obtained CodeAliasTarget to its delegate. In code this would look like

```
// Use or create an identifier for a concept stored in the db
CodeSystemQualifier codeSystem = CodeSystemQualifier.create("oid");

// Determine your alias name in the default context
CodeAliasQualifier codeAlias = CodeAliasQualifier.create("alias");

// Register your alias for the desired code system
codeAliasService.register(codeAlias, codeSystem);

// Retrieve all code sets defined for your code system by using the registered
// alias
List<CodeSetQualifier> codeSets = codeFinderService.findCodeSets(codeAlias);
```

16.1.2.6.3 Input Validation:

The validation of input parameters verifies any given CodeQualifier argument for null values or missing attributes. If these requirements are violated then a InvalidSpecificationException is thrown. The exception encapsulates the causing exception which would either be a NullPointerException or a PropertyMissingException . See section [CodeQualifier](#) on page 239 for the mandatory properties.

16.1.2.6.4 Transaction Propagation:

All services support their execution within a transaction. The transaction handling must be provided externally by injecting a transactionManager bean which adheres to the org.springframework.transaction.PlatformTransactionManager interface. Whenever the CodeAliasService or the CodeAuthoringService modifies the code base (e.g. registering an alias or adding codes) they require a transaction. If those method calls are not part of the transaction, the transactionManager will create one. Otherwise and all other services engage into the current transaction.

16.1.2.7 Handling defective Requests

The Code Services compensates faulty parameters and other unexpected behaviour or usage with a CodeServiceException . Depending on the cause the following exceptions are indicated

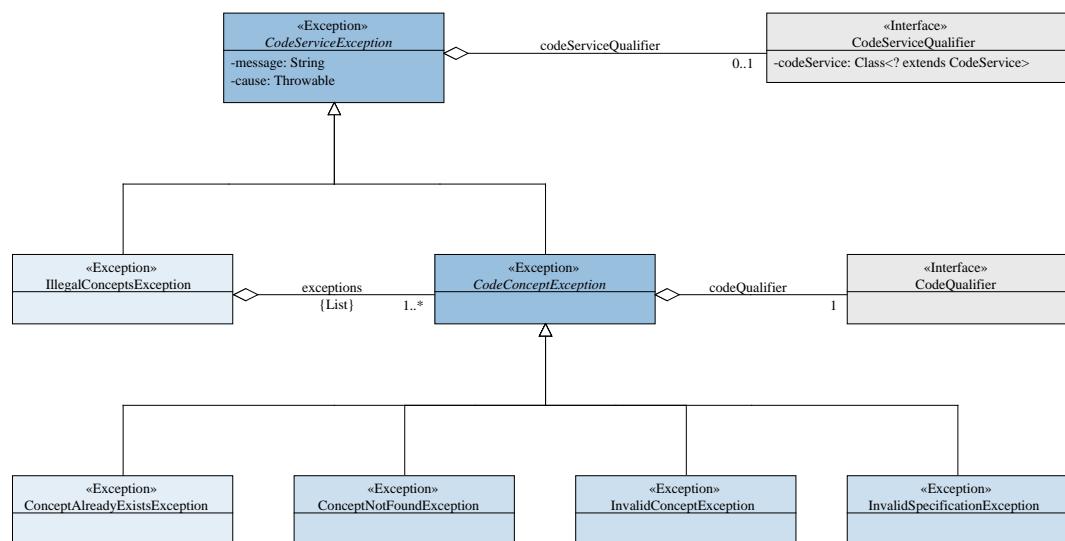


Figure 102: Exception Hierarchy

There is one exception not being shown. An unauthorized access to the CodeAuthoringService triggers a java.security.AccessControlException . See section [Authorization](#) on page 251 for more details.The purpose of the exceptions are listed in the following table:

Exception	Description
CodeServiceException	Defines the interface for all exceptions occurred while processing a CodeService request.
CodeConceptException	Defines the interface for all exceptions occurred while processing a parameter that qualifies a code concept.

Exception	Description
ConceptAlreadyExistsException	Exception indicating the existence of a code concept.
ConceptNotFoundException	Exception indicating the missing of a code concept.
IllegalConceptsException	Exception indicating multiple problems with a collection of code concepts.
InvalidConceptException	Exception indicating the invalidity of a code concept.
InvalidSpecificationException	Exception indicating the invalidity of the code concept attributes.

Table34. Overview of CodeServiceException

See the javadoc to find out, how the thrown exceptions are mapped/related to their services.

16.1.2.8 Examples for using the API

This section provides examples accessing the code content using the previously presented Code Services.

16.1.2.8.1 Database Content:

Suppose the following code values are already persisted in the database.

The code system with the oid '1.2.3' contains 4 codes.

oid	version	key	locale	value
1.2.3	1.0.0	key1	en_US	val1
1.2.3	1.0.0	key2	en_US	val2
1.2.3	1.0.0	key3	en_US	val3
1.2.3	1.0.0	key4	en_US	val4

Table35. Example Code Systems

The code system name 'codeSystem1' is registered as an alias for the oid '1.2.3'.Also two code sets ('codeSetA', 'codeSetB') are defined.

code set name	code set version	oid	version	key
codeSetA	1.0.0	1.2.3	1.0.0	key1
codeSetA	1.0.0	1.2.3	1.0.0	key2
codeSetB	1.0.0	1.2.3	1.0.0	key3

Table36. Example Code Sets

The code with the key 'key4' is not part of any code set.Further the code sets are assigned to the category 'C-CAT'.

16.1.2.8.2 Get a code system from the database:

This example starts by loading the existing code systems within the database. Then it picks one arbitrarily and retrieves the registered aliases using the CodeAliasService . This gives

us the name of the code system since the system will import the name of a code system as an alias.

```
// Get all code systems
List<CodeSystemQualifier> codeSystems = codeFinderService.findCodeSystems();

// There is only one entry (codeSystems.size() == 1)
CodeSystemQualifier codeSystemQualifier = codeSystems.get(0);

// Get all aliases for a particular code system
List<CodeAliasQualifier> codeAliases = codeAliasService.find(codeSystemQualifier);

// There is only the pre-imported (codeAliases.size() == 1)
CodeAliasQualifier codeAliasQualifier = codeAliases.get(0);

// Get the name of the code system (codeAliasQualifier.getContext() is "NAME")
String codeSystemName = codeAliasQualifier.getName(); // = "codeSystem1"
```

Alternatively when the code system name is of interest, you can acquire a map of all code systems and their aliases from the database with a single service call.

```
// Get all code systems and their aliases
Map<CodeAliasQualifier, CodeSystemQualifier> codeSystemMap = codeAliasService.
find(CodeSystemQualifier.class, "NAME");

// There is only one entry (codeSystemMap.size() == 1)
CodeSystemQualifier codeSystemQualifier = codeSystemMap.values().get(0);
CodeAliasQualifier codeAliasQualifier = codeSystemMap.keySet().get(0);

// Get the name of the code system
String codeSystemName = codeAliasQualifier.getName(); // = "codeSystem1"
```

16.1.2.8.3 Get a display value of a code when given a code system:

Once you have a starting point like a code system, you might want to retrieve an element (a CodeKeyQualifier) and its display value (CodedValue).

```
// Get all codes of the code system (use the default criteria)
CodeContainer<CodeKeyQualifier> container = codeFinderService.
find(codeSystemQualifier, null);

// Get the second code of the 4 available (container.numCodes() == 4)
CodeKeyQualifier codeKeyQualifier = container.getCodes().get(1);

// Retrieve the US display value for the code
CodedValue codedValue = codeResolverService.resolve(codeKeyQualifier, Locale.US);

// Get the display value
String displayValue = codedValue.getValue(); // = "val2"
```

When you need more or all display values the following code snippet is a shorter alternative which uses CodeResolverService exclusively.

```
// Get all US display values for the code system
CodeContainer<CodedValue> container = codeResolverService.
resolve(codeSystemQualifier, CodeCriteria.createResolveCriteria(Locale.US));

// Get the second value of the 4 available (container.numCodes() == 4)
CodedValue codedValue = container.getCodes().get(1);

// Get the display value
String displayValue = codedValue.getValue(); // = "val2"
```

16.1.2.8.4 Register an alias for a code category:

Instead of using a concept like code system or code category directly, you may use an alias. In this example we will retrieve all code categories from the database as a starting point. Then we pick one arbitrarily and register an alias in our own context for that particular code category.

```
// Get all code categories
List<CodeCategoryQualifier> categories = codeFinderService.findCodeCategories();

// There is only one entry (categories.size() == 1)
CodeCategoryQualifier codeCategoryQualifier = categories.get(0);

// Define an alias
CodeAliasQualifier codeAliasQualifier = CodeAliasQualifier.create("my_alias",
    "my_context");

// Register the alias for the category
codeAliasService.register(codeAliasQualifier, codeCategoryQualifier);
```

16.1.2.8.5 Get all codes for a given code category alias:

An alias for a code concept is represented by a CodeAliasQualifier . The qualifier does not provide methods to retrieve the code concept it stands for. This must be done manually by calling the lookup of the CodeAliasService . But this isn't a real problem because the qualifier can be used wherever a service takes a parameter of the type CodeKeyCollection . The following example does just that and retrieves all codes of a code category.

```
// Get all codes for a given alias
CodeContainer<CodeKeyQualifier> container = codeFinderService.
find(codeAliasQualifier, null);

// The code category "C-CAT" contains 3 codes
assertTrue(container.getCodes().size() == 3);
assertTrue(container.getCodeKeyCollection() instanceof CodeCategoryQualifier);
```

Of course it is also possible to retrieve the display values using the CodeResolverService .

```
// Get all code values for a given alias
CodeContainer<CodeValue> container = codeResolverService.find(codeAliasQualifier,
    CodeCriteria.createResolveCriteria(Locale.US));

// The code category "C-CAT" contains 3 code values
assertTrue(container.getCodes().size() == 3);
assertTrue(container.getCodeKeyCollection() instanceof CodeCategoryQualifier);
```

16.1.2.8.6 Add a code to a code set:

This example shows how to extend a code set. Starting with a given CodeSystemQualifier , we find all code sets that are based on this code system. We'll pick the code set 'codeSetB' and add the code with the key 'key4' to it by calling the add method of the CodeAuthoringService . When successful, a new relation between the code and a new version of the code set is persisted. The resulting version is returned.

```
// Determine the code set (for example)
List<CodeSetQualifier> codeSets = codeFinderService.
findCodeSets(codeSystemQualifier);
```

```

CodeSetQualifier codeSetQualifier = codeSets.get(1);
String codeSetName = codeSetQualifier.getName(); // "codeSetB"

// Get the last code of the 4 available (container.numCodes() == 4)
CodeContainer<CodeKeyQualifier> container = codeFinderService.
find(codeSystemQualifier, null);
CodeKeyQualifier codeKeyQualifier = container.getCodes().get(3);
String codeKey = codeKeyQualifier.getKey(); // = "key3"

// Add the code to the code set
Set<CodeKeyQualifier> codeKeys = new HashSet<CodeKeyQualifier>();
codeKeys.add(codeKeyQualifier);
CodeSetQualifier newCodeSetQualifier = codeAuthoringService.add(codeSetQualifier,
codeKeys);
String newCodeSetVersion = newCodeSetQualifier.getVersion(); // = "1.1.0"

```

16.1.3 Codesystem Import Processor

Overview

The handling of terminologies is grouped into three aspects. The definition of code systems, code sets and code terms are terminology definitions which can be defined independently of each other. An XML structure along with appropriate parsers are provided to define the concept. Each code concept has its own tag. In order to control the availability and the location of the content a terminology catalog describes these information in XML. By referencing catalogs and listing the required code concepts a terminology configuration determines the terminology content of an application and allows further to glue this content to the domain by defining assignments between code sets and categories.

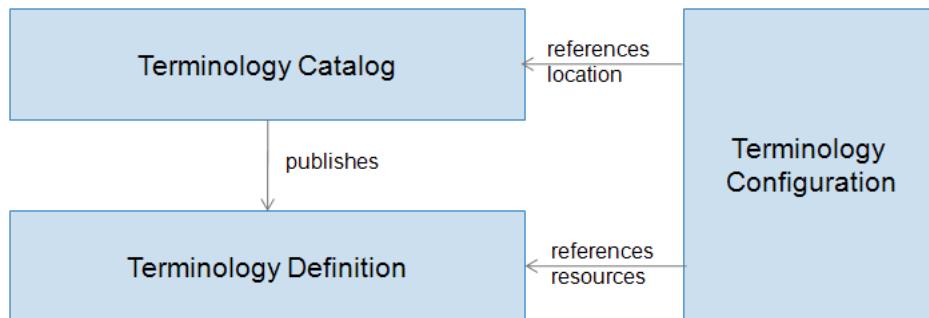


Figure 103: Handling of terminologies in eHF based applications

Terminology Definition

The terminology definition includes the definition of code system, code set and code terms resources.

Often only a portion of the concept code of a code system issued by a standard developing organization or others is required in the context of an application. This fact is taken into account by using code system contributions. A contribution specifies a bundle of code concepts that are all identified by the same object identifier. All contributions that reference the same object identifier form the code system content. A contribution is joined into the code system when its status is final. Drafted or retired contributions are as indicated in a state prior or after the active usage.

A code system contribution consists of

- the object identifier and the code system name as identification of the code system

- meta information like status, description and source information about the curator of this code system and a source tag that represent the revision number of the code system
- the list of codes and their concept values. The concept value describes the concept within a comprehensive term and is normally defined by the curator of a code system. The concept value serves two major purposes. It supports the understanding of the concept represented by the code and it will be used as default value of code terms when none is available.



Note: As the concept value is mandatory multiple contributions may provide such a term. Therefore the codes are joined in such a way that the first contribution that specifies the code determines its concept value. Further contributions do not overwrite this.

```
<?xml version='1.0' encoding='UTF-8'?>

<ehf-terminology-definition version='1.0.0'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns='http://www.intercomponentware.com/schema/ehf-terminology-definition'
  xsi:schemaLocation='http://www.intercomponentware.com/schema/
    ehf-terminology-definition
    http://www.intercomponentware.com/schema/
    ehf-terminology-definition.xsd'>

  <codesystem oid='dummy-oid' name='dummy-codesystem' status='final'>
    <source tag='revision'>Dummy</source>
    <description>Dummy code system description</description>
    <codes>
      <code key='a'>cv_a</code>
      <code key='b'>cv_b</code>
      <code key='c'>cv_c</code>
    </codes>
  </codesystem>
</ehf-terminology-definition>
```

Figure 104: Example - Code system definition

While the concept value of a code serves as a technical term to describe the meaning of the concept behind the code, code terms serve as localized display values because they have a language encoding. Such code terms come in two flavors. One is addressed as a preferred term and only one is allowed for each code in a particular locale while the other are synonymous values which are not restricted in number. Like contributions the code terms support a status draft, final and retired. Only the final code term definitions are imported.

```

<?xml version='1.0' encoding='UTF-8'?>

<ehf-terminology-definition version='1.0.0'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.intercomponentware.com/schema/ehf-terminology-definition'
    xsi:schemaLocation='http://www.intercomponentware.com/schema/
        ehf-terminology-definition
        http://www.intercomponentware.com/schema/
        ehf-terminology-definition.xsd'>

    <codeterm oid='dummy-oid' locale='en' status='final'>
        <code key='a'>en_a</code>
        <code key='c'>en_c</code>
        <code key='d'>en_d</code>
        <code key='e'>en_e</code>
    </codeterm>

    <codeterm oid='dummy-oid' locale='en_US' status='final'>
        <code key='a'>us_a</code>
        <code key='b'>us_b</code>
    </codeterm>

</ehf-terminology-definition>

```

Figure 105: Example - Code term definition

A concept value is different from a preferred code term as it is mandatory. That means for each code concept a term exists. This is not the case for the preferred terms. So sometimes there are codes without a preferred term for a locale. The codesystem services provide a fallback mechanism to fill up those terms in a way that all codes have a code term in a particular locale. More precise, missing preferred terms are substituted by the parent locale or the concept value of a code. This mechanism works similar to the properties locale fallback in Java. For example a missing term for 'en_US' is replaced with the one for 'en' and if none is present the concept value of that code is used. See the next table for a better understanding.

Value Type	Key					
	a	b	c	d	e	f
Terminology Definition						
Concept Value	cv_a	cv_b	cv_c	cv_d	cv_e	cv_f
Code Term locale 'en'	en_a	-	en_c	en_d	en_e	-
Code Term locale 'en_US'	us_a	us_b	-	-	-	-
When queried						
Code Terms locale 'en'	en_a	cv_b	en_c	en_d	en_e	cv_f
code terms locale 'en_US'	us_a	us_b	en_c	en_d	en_e	cv_f

Figure 106: Fallback mechanism of code terms

Since the introduction of the terminology authoring service the control over the code set versioning went over from the author to the system. Applying this aspect to the terminology definition lead us to the denomination of a code set definition as a draft to ensure the data integrity of code sets. The versioning of code sets must follow a set of rules that are enforced by the system. The rules are as follow:

- A newly created code set is given the version 1.0.0.
- If a code is added to a code set, the minor version number is increased by 1 (e.g. from 1.0.0 to 1.1.0).
- If a code is removed from a code set, the major version number is increased by 1 and the minor version is set to 0. (e.g. from 1.5.0 to 2.0.0).

For example a draft that brings more codes than what the current code set version has in the system will lead to a minor increase of the version number with the newer codes added.



Note: The control of the code set versioning can be executed manually. But by doing so it is important not to violate the set of rules. If a customer change these code sets, data integrity issues may occur!

A code set draft consists of

- the code set name and the OID of the referenced code system as an identification of the code set
- meta information like the status of the code set definition and a description on the purpose of the code set
- the list of codes or an regular expression that defines which codes from the referenced code system are included or excluded.

The following examples show two code set drafts. The first defines a code set draft using an explicit list of codes while the second is based on regular expressions.

```
<?xml version='1.0' encoding='UTF-8'?>

<ehf-terminology-definition version='1.0.0'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.intercomponentware.com/schema/ehf-terminology-definition'
    xsi:schemaLocation='http://www.intercomponentware.com/schema/
        ehf-terminology-definition
        http://www.intercomponentware.com/schema/
        ehf-terminology-definition.xsd'>

    <codeset name='dummy-codeset' ref-oid='dummy-oid' status='final'>
        <description>Dummy code set description</description>
        <code key='a' />
        <code key='b' />
    </codeset>
</ehf-terminology-definition>
```

Figure 107: Example - Explicit defined code set

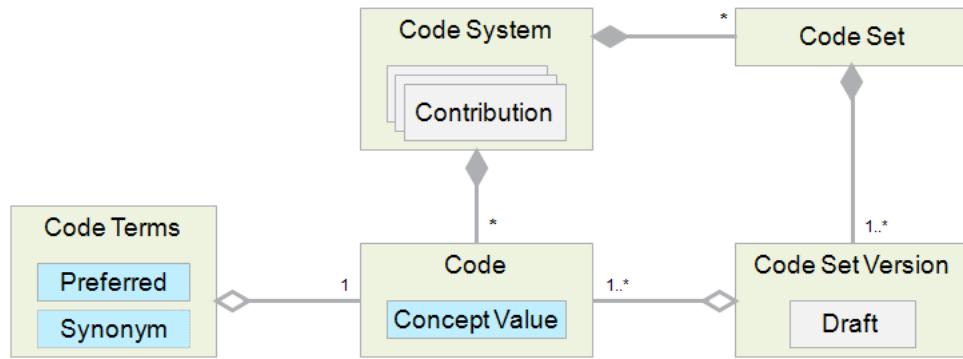
```
<?xml version='1.0' encoding='UTF-8'?>

<ehf-terminology-definition version='1.0.0'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.intercomponentware.com/schema/ehf-terminology-definition'
    xsi:schemaLocation='http://www.intercomponentware.com/schema/
        ehf-terminology-definition
        http://www.intercomponentware.com/schema/
        ehf-terminology-definition.xsd'>

    <codeset name='dummy-codeset' ref-oid='dummy-oid' status='final'>
        <description>Dummy code set description</description>
        <pattern>
            <include>.*</include>
            <exclude>e</exclude>
        </pattern>
    </codeset>
</ehf-terminology-definition>
```

Figure 108: Example - Implicit defined code set

The following figure gives you an overview on the terminology definitions and their related concepts.

**Figure 109:** Concepts of Terminology Definitions

Terminology Catalog

The approved terminology definitions are published by a terminology catalog. This includes a list of the available

- code system contributions,
- code set drafts or code set versions and
- code terms.

Therefore a catalog serves as table of content. It further specifies the location of the resources. As the resource information are hidden from the import configuration it is easy to provide the terminologies from different locations (e.g. classpath, ftp, file).

The import process supports multiple terminology catalogs. So you may define catalogs for special purposes like a catalog for each domain group or per terminology revision increment. The import processor only retrieves the available terminologies of the catalogs that are configured.

The code concepts that are directly or transitively dependent on an object identifier are addressed as a terminology in the catalog file. The following example shows you the terminology definition that are dependent on the object identifier `dummy-oid`.

```

<?xml version="1.0" encoding="UTF-8"?>

<ehf-terminology-catalog version="1.0.0"
    id="dummy-id" location="classpath:META-INF/repository"
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.intercomponentware.com/schema/ehf-terminology-catalog'
    xsi:schemaLocation='http://www.intercomponentware.com/
        schema/ehf-terminology-catalog
        http://www.intercomponentware.com/
        schema/ehf-terminology-catalog.xsd'>

    <terminology oid="dummy-oid">
        <codesystem>
            <contribution>
                <resource path="codesystems"
                    filename="dummy-oid-contribution1.xml" />
                <resource path="codesystems"
                    filename="dummy-oid-contribution2.xml" />
            </contribution>
        </codesystem>
        <codesets>
            <codeset name="dummy-codeset">
                <resource path="codesets" filename="dummy-codeset-draft1.xml" />
                <resource path="codesets" filename="dummy-codeset-draft2.xml" />
                <resource path="codesets" filename="dummy-codeset-draft3.xml" />
            </codeset>
        </codesets>
        <codeterms path="codesystems" filename="dummy-oid-terms.xml">
            <codeterm locale="en" />
            <codeterm locale="en_US" />
        </codeterms>
    </terminology>
</ehf-terminology-catalog>
```

Figure 110: Example - Terminology Catalog

For an incremental import you may want to create a catalog for a terminology revision increment. Such a catalog would list only the updated or newly available definitions.

The relations between the terminology catalog and their terminology definitions will be illustrated in the following figure.

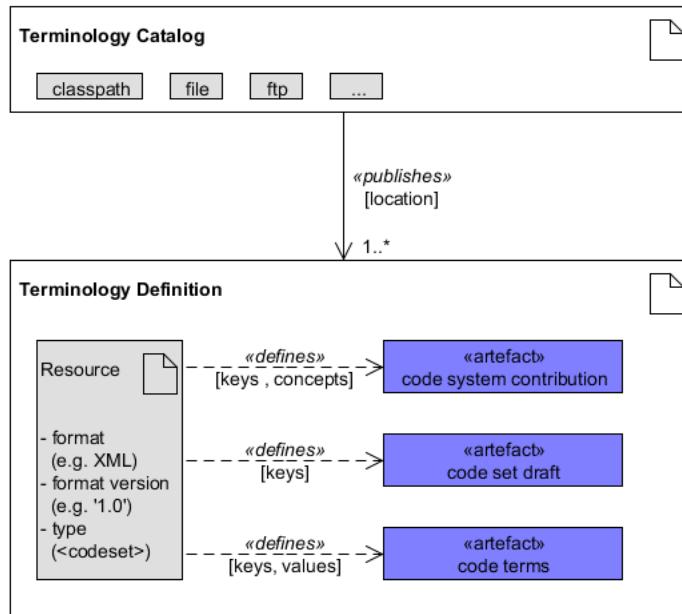


Figure 111: Terminology Definition and Terminology Catalog

Terminology Configuration

The terminology configuration determines the import. Normally you need to configure

- the terminology catalogs to look up the available concepts and their locations,
- the code concepts that are required by the application, this specifies which code system, code sets and localized code terms are needed, and
- the assignment of code sets to those code categories that are bound to the coded attributes of the application.

Like in the old import process, it is possible to assign different code sets to a code category. The status of an code set assignment is either active or inactive. Only one version of a code set can be assigned as active. If you assign a new version of a code set, the previous version that was assigned as active will be set to inactive by the import processor.



Note: At the moment, only the current version of a code set can be assigned as active
(This is subject to change).

The following example shows a configuration of an exemplary application that requires the code set dummy-codeset, the code system dummy-oid and its code terms for the USA. Further for all configured code systems the english code terms are needed. The code set dummy-codeset is assigned to the code category dummy-category. The resources of the configured terminology definition are taken from the catalog dummy-catalog.xml.

```

<?xml version="1.0" encoding="UTF-8"?>

<ehf-terminology-config version='1.0.0'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns='http://www.intercomponentware.com/schema/ehf-terminology-config'
    xsi:schemaLocation='http://www.intercomponentware.com/
        schema/ehf-terminology-config
        http://www.intercomponentware.com/
        schema/ehf-terminology-config.xsd'>

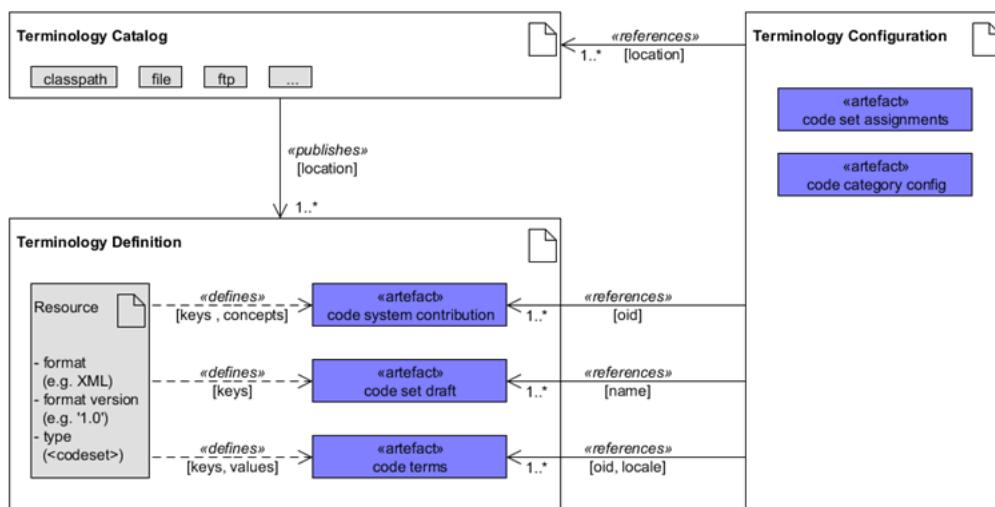
    <catalogs>
        <catalog location='classpath:META-INF/repository/dummy-catalog.xml' />
    </catalogs>

    <codeterms>
        <codeterm locale='en' />
    </codeterms>
    <codesystems>
        <codesystem oid='dummy-oid'>
            <codeterm locale='en_US' />
        </codesystem>
    </codesystems>
    <codesets>
        <codeset name='dummy-codeset' />
    </codesets>
    <categories>
        <category name='dummy-category'>
            <active codeset='dummy-codeset' />
        </category>
    </categories>
</ehf-terminology-config>

```

Figure 112: Example - Terminology Catalog

The relations between the catalog, the terminology definitions and the configuration for the import process is illustrated in the following figure.

**Figure 113:** Terminology Definition, Terminology Catalog and the Configuration

16.2 eHF Configuration

The eHF Configuration module provides applications with a mechanism to define placeholders for property values of Spring beans. The values for these placeholders can be configured at or before deploy-time. They can be exposed as runtime configuration items via JMX. Changes made at runtime are persistent.

16.2.1 Features

- Definition of bean properties can be delayed to later phases by using placeholders
- Transparent isolation of placeholder values for separate eHF Modules
- Placeholder values can be stored in files or the database
- Overriding of file-defined placeholder values by database-defined ones
- Differentiation between runtime and deploy-time configuration values
- Management of runtime configuration values via JMX
- Unified view of all active placeholder values via JMX
- Changes to placeholder values are published as events using the eHF messaging infrastructure

Supported property types are:

- String
- Integer/int
- Long/long
- Boolean/boolean

16.2.2 Restrictions

eHF Configuration only supports the management of properties of JavaBeans which where instantiated by Spring'sbean factory.

Since version 2.9.8 eHF Configuration supports the redistribution of changed deploy-time configuration values among cluster nodes.

Placeholder values defined in the database have always precedence over values defined in property files. There is no support to change this behavior.

16.2.3 Usage

In order to make use of the features eHF Configuration provides both the affected modules and the assembly have to be configured accordingly.

16.2.3.1 Module Configuration

16.2.3.1.1 Dependencies :

Define a dependency in your module's project.xml to the eHF Configuration module and the eHF Messaging module as follows. Replace '\${ehf.version}' with actual used version of artifacts.

```
<!-- eHF CONFIGURATION -->
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-configuration-config</artifactId>
    <version>${ehf.version}</version>
    <properties>
        <ehf-module>ehf-configuration</ehf-module>
    </properties>
    <type>jar</type>
</dependency>
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-configuration</artifactId>
    <version>${ehf.version}</version>
```

```

<properties>
    <war.bundle>true</war.bundle>
</properties>
<type>jar</type>
</dependency>
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-messaging-api</artifactId>
    <version>${ehf.version}</version>
    <type>jar</type>
</dependency>
<!-- eHF CONFIGURATION END -->
```

16.2.3.1.2 Configuration Namespace :

Introducing the Configuration Namespace

1. Define a configuration bean in your Spring context referencing your module's name. This requires declaring the additional configuration namespace and schema in the Spring application context, e.g. in the custom context.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ehf="http://www.intercomponentware.com/schema/icw-util"
       xmlns:configuration="http://www.intercomponentware.com/schema/ehf-
configuration"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.5.xsd
                           http://www.intercomponentware.com/schema/ehf-
                           configuration
                           http://www.intercomponentware.com/schema/ehf-
                           configuration/ehf-configuration.xsd">

    <configuration:configuration module-name="@@module.name@@">
        [...]
    </configuration:configuration>
</beans>
```

The module name should not be entered explicitly; instead the eHF build can resolve it automatically if the "@@module.name@@" placeholder is used (this is an eHF build-level placeholder and completely unrelated to the placeholder mechanism of eHF Configuration described here). Child elements of the configuration bean are described in later sections.

2. Place your property files in the resources directory of your module and list every file to include as an individual location element:

```

<configuration:configuration module-name="@@module.name@@">
    <configuration:location path="configuration.properties" />
    <configuration:location path="other_configuration.properties" />
</configuration:configuration>
```

These property files contain the values for all the placeholders in use, unless they were changed at runtime or directly in the database.

Making an existing bean property deploy-time configurable

The following steps are required to make an existing bean property deploy-time configurable:

1. Replace the actual value in the bean definition by a placeholder with a name that is unique within the module. The placeholder must take the following form "\${placeholdername}" (as defined by Spring's PropertyPlaceholderConfigurer which the module utilizes). The placeholder name should only consist of alphanumeric characters, dots or underscores. Example:

```

<bean id="testBean" class="...">
    <property name="aProperty" value="${test_key}" />
    <property name="anotherProperty" value="${second.test.key}" />
```

```
</bean>
```

The bean property value must not contain anything other than the placeholder (i.e. no prefix or suffix). Nested placeholders and usage of several placeholders in one property are not supported. Placeholders are supported only for string properties, so that wrapper classes should be used for all other property types. To avoid side effects, it is not allowed to use the same placeholder in more than one bean.

2. Add a new entry to one of the configured property files with the chosen placeholder name as the property key and the desired value as the property value. Example:

```
test_key=my value
second.test.key=my other value
```

When the application starts it will read in all placeholder values from the list of property files as well as from the database. In case a placeholder value is defined in both places the value in the database has priority.

Making a property placeholder runtime configurable

Before a bean property can be made runtime configurable it has to be deploy-time configurable (see above). The following steps are required in addition:

1. Configure the JMX name for the bean. The additional element

`configuration:jmx` added to the configuration described above will expose a Spring bean named `testBean` under the MBean name `testMBean`.

```
<configuration:configuration module-name="@@module.name@@">
    <configuration:location path="configuration.properties" />
    <configuration:location path="other_configuration.properties" />

    <configuration:jmx name="testMBean" ref="testBean" />
</configuration:configuration>
```

2. Optionally, developers may explicitly define which methods are exposed. If this step is omitted, the default behavior is to expose only the getter and setter methods of properties that contain place holders. To define methods and properties to expose, use the `configuration:method` child element of the `configuration:jmx` element:

```
<configuration:configuration module-name="@@module.name@@">
    <configuration:location path="configuration.properties" />
    <configuration:location path="other_configuration.properties" />

    <configuration:jmx name="testMBean" ref="testBean">
        <configuration:method name="getAnotherProperty" />
        <configuration:method name="setAnotherProperty" />
        <configuration:method name="findSomething" />
    </configuration:jmx>
</configuration:configuration>
```

The method name is interpreted according to the rules defined by Spring's `MethodNameBasedMBeanInfoAssembler`. While it is possible to expose setter methods of properties without place holders, it is discouraged to avoid confusion for the user of the JMX client, who would then have no way of telling if a change will be persisted. If a setter method of a property is exposed, the implementing class has to define a public getter method for this property.

All beans are exposed under their module name as the root JMX folder. It is currently not supported to override this behavior.

16.2.3.1.3 Bean Dependencies :

To use eHF Configuration, a module needs two platform beans:

- A `ConfigurationService`
- A `MessageSender`

Configure the bean dependencies in the module context file:

```
<beans:beans>
    <module id="yourModule">
```

```
[...]
<import>
    <ref-bean alias="eventMessageSender" interface="com.icw.ehf.messaging.
MessageSender" />
    <ref-bean alias="configurationService" interface="com.icw.ehf.
configuration.ConfigurationService" />
</import>
</module>
</beans:beans>
```

16.2.3.1.4 Testing :

Custom configuration context for quicker testing

It is recommended to package the eHF configuration context setup into a dedicated file to be able to include the eHF configuration (and its dependencies on eHF audit and eHF messaging) for productive use but exclude it during testing. An example of such a dedicated file would be yourModule-configuration-context.xml. To be included for productive use, it has to be imported in the module context configuration as follows:

```
<module id="yourModule">
<configLocations>
    [...]
    <value>classpath:/META-INF/yourModule-configuration-context.xml</value>
</configLocations>
    [...]
</module>
```

Furthermore is it recommended to package internal bean definitions into a dedicated file yourModule-custom-context.xml in folder src/main/config. It can be used in both contexts, module and test. To be included for productive use, it has to be imported in the module context configuration as described before. To exclude it during testing, make sure you do not reference module context in your test definition. But don't forget to import your custom context with needed beans in test context (yourModule-test-context.xml) as follows:

```
<!-- remove module context for tests -->
<!--     <import resource="classpath:/META-INF/yourModule-module-context.xml" />-->
<import resource="classpath:/META-INF/yourModule-custom-context.xml" />
```

Testing can instead use Spring's standard PropertyPlaceholderConfigurer to resolve the same placeholders. To use it during testing, it has to be setup in the test application context (yourModule-system-context.xml). For example:

```
<bean class="org.springframework.beans.factory.config.
PropertyPlaceholderConfigurer">
    [...]
    <property name="properties">
        <props>
            <prop key="aProperty">my test value</prop>
            <prop key="anotherProperty">my other test value</prop>
        </props>
    </property>
</bean>
```

How to determine active placeholder values at runtime

eHF Configuration exposes runtime configuration information via JMX under @@module.name@@/configurationInfo seperately for each module. All active placeholder values are accessible under the attribute ActiveProperties. The column *name* contains the placeholder name and the column *value* the placeholder value. The list displays all placeholder values from all property files, unless the placeholder value has been overwritten by a database stored value (i.e. one changed at runtime).

16.2.3.2 Assembly Configuration

Prepare your assembly configuration

1. Define a dependency in your assembly's project.xml to the eHF Configuration and your modules. Replace '\${ehf.version}' with actual used version of artifacts.

```
<!-- eHF CONFIGURATION -->
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-configuration-config</artifactId>
    <version>${ehf.version}</version>
    <properties>
        <ehf-module>ehf-configuration</ehf-module>
    </properties>
    <type>jar</type>
</dependency>
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-configuration-runtime</artifactId>
    <version>${ehf.version}</version>
    <properties>
        <war.bundle>true</war.bundle>
    </properties>
    <type>jar</type>
</dependency>
<dependency>
    <groupId>ehf</groupId>
    <artifactId>ehf-messaging-runtime</artifactId>
    <version>${ehf.version}</version>
    <type>jar</type>
</dependency>
<!-- eHF CONFIGURATION END -->
<!-- YOUR MODULE DEPENDENCIES -->
...
<!-- YOUR MODULE DEPENDENCIES END -->
```

2. The eHF Configuration module publishes runtime changes as events which can be used for auditing purposes. For each change to a place holder value an event is fired containing a time stamp, place holder name, bean property name, old value and new value. These events are published using the eHF Messaging Infrastructure.

First of all make sure, that you have set up a eHF Messaging infrastructure, detailed information can be found in the *eHF Reference Documentation*. After setting up eHF Messaging, make sure that you declare a spring bean which is named *eventMessageSender* and is of type *com.icw.ehf.messaging.mem.MessageSenderImpl*. The eHF Configuration module is using this bean to publish messages containing configuration change events. Furthermore, an event transformer, which will transform events of type *com.icw.ehf.configuration.event.PlaceholderValueChangeEvent* into eHF Audit events has to be registered with the messaging context. You can easily register this transformer by adding it to an *TransformationBeanBinder* in *ehf-system-messaging-context.xml*:

```
<bean id="jmxEventBinder"
    class="com.icw.ehf.transform.bind.TransformationBeanBinder">
    <constructor-arg index="0"
        value="com.icw.ehf.configuration.event.PlaceholderValueChangeEvent" />
    <property name="transformer">
        <bean class=
            "com.icw.ehf.configuration.event.
PlaceholderValueChangeEventTransformer" />
    </property>
</bean>
```

With this basic setup, eHF Configuration, e.g., can participate in auditing property changes.

3. It is necessary to activate a module eagerly in order to enable the export of JMX Beans. In order to ensure this you can either use the ModuleActivator bean in your ehf-system-context.xml as follows:

```
<!-- Activate all modules which are marked with 'supportActivation' -->
<bean class="com.icw.ehf.commons.spring.context.ModuleActivator">
    <property name="lazyActivation" value="false" />
</bean>
```

The better approach is to define your module as ordered. This can be done in the module context of your module:

```
<ehf:module id="myModule" order="20">
    [...]
</ehf:module>
```

Please note that with the order setting you can influence in which order modules are started. Modules with the ordered attribute are deterministically started.

4. The eHF Configuration internally uses ehCache. Therefore, it expects a cache configuration file called ehcache-placeholder.xml on the classpath during (test) runtime.

16.2.3.3 Replication

The ehf-configuration modules provides a feature to replicate place holder updates among a cluster. That feature relies on ehcache's peer replication facility being set up for all participating nodes/application instances.

1. In your assembly, enable cache replication for the placeholderEhCache ehcache in the appropriate ehcache configuration file.

Note: Please, see the appendix for how to configure caches in ehf modules.



Note: Please, see http://ehcache.org/documentation/distributed_caching.html for more information on how to configure a distributed cache with ehcache. Available technology choices are, among others, synchronous or asynchronous RMI, JMS or Terracotta (www.terracotta.org).

-
2. With step 1 in place, you are already propagating place holder updates in your module to all listening peers. In order to receive updates from peers, too, you need to export some additional beans from your module context into the platform context:

- All beans containing place holders you want to replicate (e.g., the testBean from the examples above)
- The <moduleName>_placeholderPersister bean
- The <moduleName>_placeholderReplicationListener bean

The latter two beans are automatically created by the configuration namespace. See the example below:

```
<export>
    <bean id="testBean"
        interface="com.whatever.TestBean" />
    <bean id="< moduleName>_placeholderPersister"
        interface="com.icw.ehf.configuration.spring.beans.
PlaceholderPersister"/>
    <bean id="< moduleName>_placeholderReplicationListener"
        interface="com.icw.ehf.configuration.event.
PlaceholderEventListener"/>
```

</export>

17 Application Modules

The Application modules allow the maintenance of medical information. The module eHF Document enables storage and retrieval of documents of arbitrary content. It supports a specialized document versioning mechanism. The module eHF Record provides, on the one hand, services for basic medical information such as diagnoses, medications, and observations, and on the other, administrative data that are related to patients and emergency contacts.

The subsequent sections provide a detailed view of the application modules, including their descriptions of the business domain objects and provided services. Please note that client applications must not access the classes of the domain model and the module services directly, but they have to use the transfer objects that are defined in the service adapter layer (see [Service Adapter Layer](#) on page 37). Class names of transfer objects are the same as the domain class names except the extension `Dto`. For instance, the domain object `DocumentContext` of eHF Document translates into the transfer object `DocumentContextDto`.

The second way of accessing and manipulating the domain objects are the web services. A detailed documentation of them can be found in the Web Service documentation of the corresponding modules.

17.1 Record Facade

The module eHF Record provides the basic functionality of an electronic health record, describing the central information in a medical system. It is composed of two sub-modules separating the administrative from the medical data. The sub-modules provide means to store and retrieve information in a structured way. The eHF Record Admin contains basic health record information of administrative nature (*administrative data*, ADM). The sub-module eHF Record Medical supports *basic medical data* (BMD) based on standardized vocabulary and common structures. The general data structures of administrative and basic medical data are based on the [HL7 v3 RIM](#) (Health Level 7 Version 3, Reference Information Model).

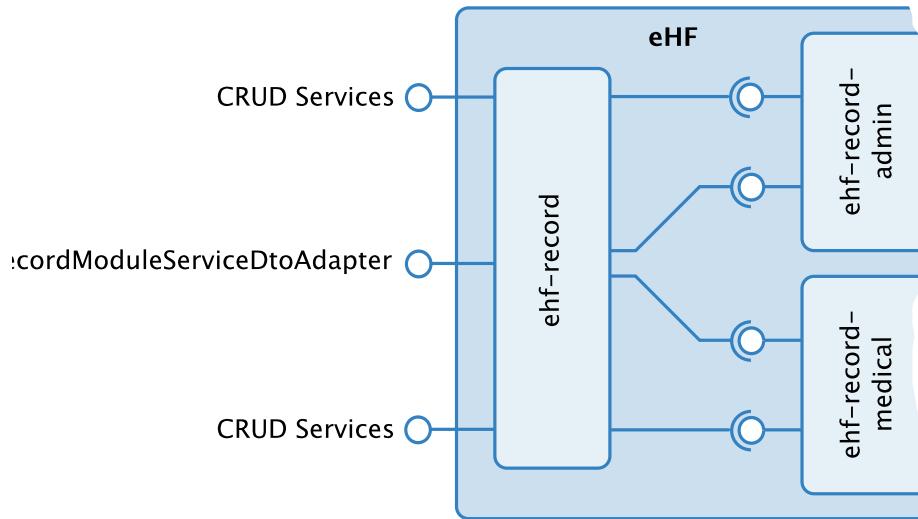


Figure 114: Facade of eHF Record

Organizing information in sub-modules enables to physically separate the storage of persisted information. This approach supports central data protection and pseudonymization considerations.

The two sub-modules are self-contained and can therefore be used as separate modules. eHF Record, however, provides one common interface for both sub-modules. It does not offer any additional functionality. Instead it acts as a so called *facade* as displayed in [Figure 114](#). The facade service `RecordModuleDtoServiceAdapter` provides the service operations of both, `RecordModuleMedicalDtoServiceAdapter` and `RecordModuleAdminDtoServiceAdapter`. In addition, eHF Record exposes the CRUD services (see [CRUD Services](#) on page 38) of the two sub-modules.

The subsequent sections provide a detailed view of both sub-modules, eHF Record Medical and eHF Record Admin, including their descriptions of the business domain objects and services.

17.2 Record Medical

The sub-module eHF Record Medical covers all *basic medical information* relevant within the context of an electronic health record. It contains, for example, diagnoses of physicians, the symptoms that physicians observed, outpatient or inpatient treatments, sick certificates, allergies and health risks, as well as medications and vaccinations.

Specialized medical domain objects such as medical history, risk history, therapy schedules, and so on are not regarded as part of eHF Record Medical. Each of the two target groups mentioned above have different business requirements lying outside the specific domain objects, which could not be supported by a common data set. They have to be implemented as part of separate modules, for example a module to support risk management. However, such specialized modules may have a strong relationship with eHF Record Medical by integrating and reusing basic objects such as health risk or allergies.

In addition to the health record specific domain model, the eHF Record Medical provides a number of services for loading and manipulating the domain objects. Both the domain model and the service operations will be covered in the following.

17.2.1 Domain Model

The domain model of eHF Record Medical is separated into multiple groups of domain classes, which are the subject of this section.

The domain model of eHF Record Medical is based on the [HL7 v3 RIM](#) (Health Level 7 Version 3 Reference Implementation Model). According to this, the abstract business domain object *Act* is the root object of all medical objects such as encounters, observations, and substance administrations as shown in [Figure 115](#).

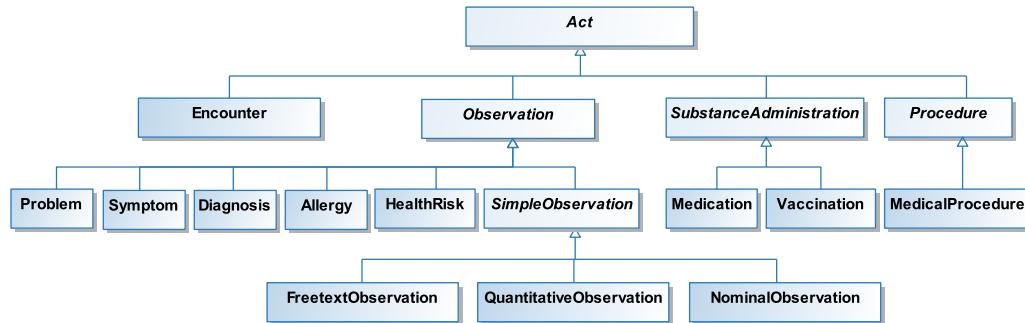


Figure 115: Overview of the eHF Record Medical Domain Model

An observation itself is an abstract domain object for quantitative and nominal observations as well as diagnoses, allergies, and health risks. Substance administrations are either medications or vaccinations.

Act

The central class of eHF Record Medical is the abstract class called *Act*. It provides, for example, attributes like *effective time*, *performing time*, *text*, and *status code*. Each act may have several participants (see [Figure 116](#)).

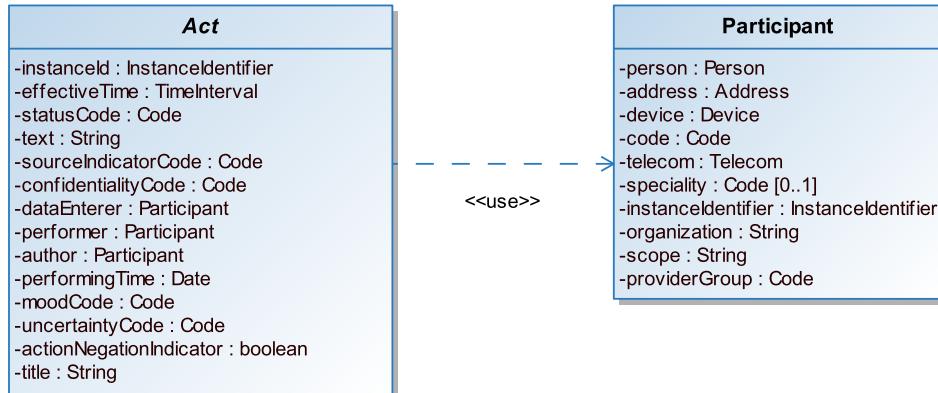


Figure 116: Act and Participants

A participant is a person who plays a certain role during the actual process of the act or during the process of documenting the act within the health record. The following types of participants are supported:

- The *performer* of an act is the person who carries out the particular action. For example, a surgery resident could act as performer by operating under supervision of

- an attending surgeon. If eHF Record Medical is used as the basis for a health record the performer (for example, of a measurement) could be the end consumer as well.
- The *author* of an act represents the people and/or machines that originally created the content. This may be a person or organization such as a healthcare provider (for example, in the case of a diagnosis identified during an inpatient encounter), a related party such as a family member (for example, in the case of an allergy reported by the mother for her child), a machine or computer system (for example, in the case of a laboratory system automatically analyzing and reporting lab values) or the end consumer himself (for example, in the case of reporting his own health risks). The author originates the act and therefore has full responsibility for the correctness of the information given.
 - The *data enterer* is the party who actually enters data associated with the act into the system. Therefore, each data enterer documented as part of an act has to be strongly related to a user known to the system. The data enterer (for example, a typist) of an act may be different from the performer (for example, a surgery resident) as well as from the author of an act (for example, the surgeon). In many cases, the roles of the data enterer and the author will be played by the same person.

All three types of participants have a documentary character. If the corresponding information changes after the creation of the act (for example, because the data enterer or performer changes his name or address) the information that is stored as part of the act still has to reflect the situation as it was at creation time. An automatic update of such information would lead to incorrect documentation.

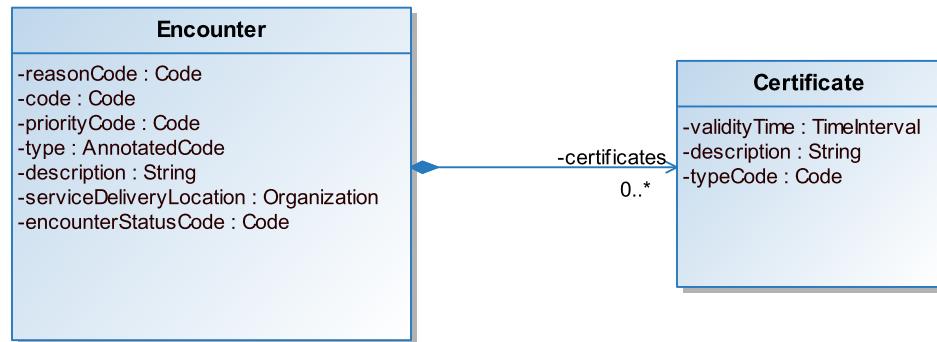
Each act may also be related to other acts and may have a number of specializations. For example, an act has a specialization called observation, which itself has a specialization called diagnosis. Diagnosis inherits the attributes of both act and observation.

Encounter

An encounter is the meeting of two or more different parties, the meeting of a patient and a doctor in the doctor's office or in the hospital, for example. Two different types of encounters are supported:

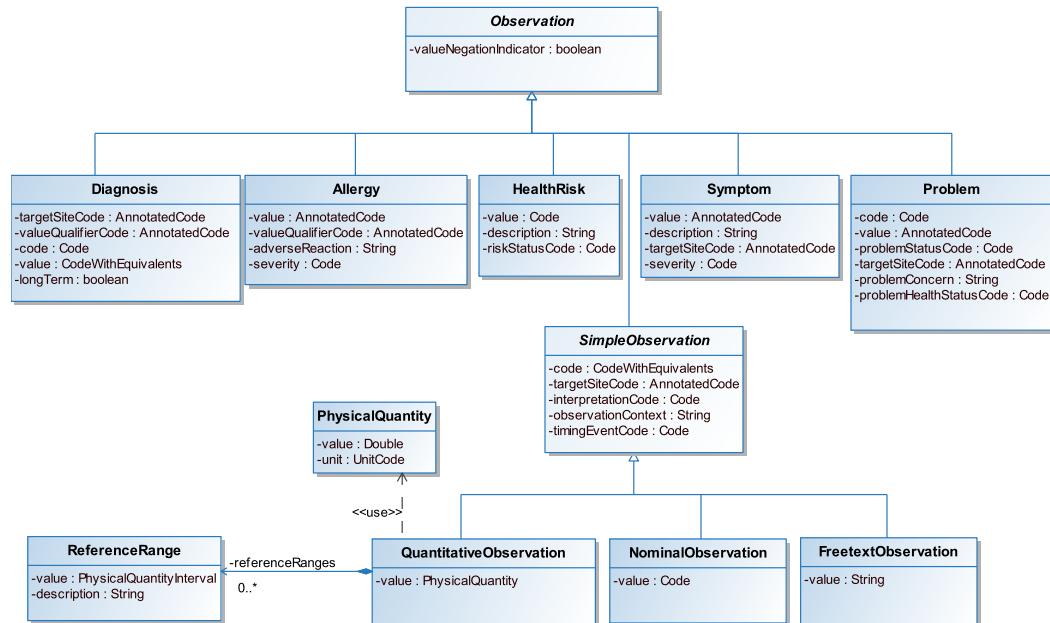
- An *ambulant or outpatient encounter* is the meeting of a patient and a medical professional in an outpatient facility related to a hospital or in a doctor's office. Outpatient means that the patient just visits for diagnosis and/or therapy and then leaves again. By definition, an ambulant encounter never includes an overnight stay in the medical facility.
- An *inpatient encounter* is the meeting of a patient and a medical professional in a hospital where the patient is *admitted* and stays overnight or for several days, weeks or even months.

During an encounter a set of optional certificates (see [Figure 117](#)) could be handed over to the patient. Examples of certificates are sick certificates, certificate of disability, and letter of confinement. All information that was documented in the context of an encounter (for example, the symptoms the patient observed, the observations the physician made, the diagnosis that he derives from the former, documents related to the encounter like lab results and doctor's letter) can be linked to an encounter instance.

**Figure 117:** Encounter and Certificates

Observation

In the medical context, a lot of information - such as symptoms, weight-measurements, temperature, blood group or allergies and health risks - is observed by the patient or physician. Conclusions derived from a set of such observations could also be regarded as observations themselves (for example, a diagnosis is the observation of a specific disease a patient suffers from). Within eHF Record Medical the general concept of an observation is represented by a corresponding abstract class from which several specializations are derived. [Figure 118](#) summarizes the supported medical observations.

**Figure 118:** Observation and its Descendant Domain Objects

Diagnosis

The aim of the diagnostic process is to identify a disease by observing its signals, symptoms and the results of various diagnostic procedures. The conclusion reached through that process is called a *diagnosis*. Medical diagnoses are usually assigned a code derived from a standardized classification (for example, ICD-10 GM used in Germany or ICD-9 WHO, which is used in the USA). In addition, eHF Record Medical also supports diagnoses without a code from a predefined code set (see [Code System](#) on page [279](#)) to allow end consumers to enter a free text description to describe a former diagnosis they want to

document within their health record (for example, that they suffered from a "fracture of the leg" ten years ago without having to know the correct ICD code).

Allergy

An *allergy* is a special kind of diagnosis that represents an immune malfunction whereby a person is hypersensitive to typically non-immunogenic substances. Since in the context of an allergy additional information like adverse reactions and severity are of special interest eHF Record Medical defines allergy as a separate business domain object.

Health Risk

A *health risk* or *risk factor* identifies an increased risk of a future disease. For example, high blood pressure, a high cholesterol value or smoking cigarettes are health risks that increase the likelihood of getting heart disease. Managing health risks is an important part of the prevention process since taking useful countermeasures can decrease the risk and help prevent the occurrence of disease.

Simple Observation

A certain set of observations is covered by the abstract class `SimpleObservation` which represents a measurement. All simple observations have in common that the actual type of measurement is identified by a code derived from a standardized vocabulary (for example, defining whether the height, weight, temperature or the blood group was determined). The different specializations distinguish measurements with different scale levels. Currently, eHF Record Medical supports quantitative observations (containing a numeric value and an optional unit as the result of the measurement), nominal observations (containing a coded value derived from defined code sets as a result) and freetext observations (containing a freetext value for textual results). Observations that belong together such as systolic blood pressure and diastolic blood pressure are grouped by means of the `observationContext` attribute (which can be regarded as a group-id). Therefore, observations with the same observation context belong together.

Quantitative Observation

For *quantitative observations* units are defined as UCUM codes (Unified Code for Units of Measure). UCUM is a code system intended to include all units of measures being currently used in international science, engineering, and business. The purpose is to facilitate unambiguous electronic communication of quantities together with their units. Examples of quantitative observations are:

- laboratory values such as cholesterol, hemoglobin, or blood sugar,
- other medical parameters like heart rate or weight,
- fitness parameters like the total numbers of steps or duration of exercise in minutes,
- nutrition parameters like calories per day or amount of meals per day.

Nominal Observation

For *nominal observations* all valid values have to be derived from a predefined set, represented by a code set (see [Code System](#) on page). Examples for nominal observations are

- blood group (with values "A", "B", "AB" and "0") and rhesus factor (with values "positive" and "negative"),
- pain (with values "extreme", "high", "medium", "low", "nothing"),
- sleeping problems (with values "never", "rarely", "often"),
- training units (with values "daily", "twice per week", "once per week", "less than weekly").

Freetext Observation

For *textual observations* the observed information is stored in form of a freetext value. Examples for freetext observations are

- answers to open questions within a questionnaire (like "How do you feel today?"),
- medical notes of a physician that might be linked to an encounter.

Symptom

In medicine a *symptom* is a sign (for example, headache or chills) that points to a disease. It may be detected by a doctor (medical report), or be experienced by the patient himself (complaint). Significant symptoms are usually noticed by the patients themselves and give them a reason to seek medical advice. Other symptoms, however, are only discovered in the context of the patient's medical history and by physical examination. Symptoms, together with other findings such as laboratory investigations, are the basis for providing a diagnosis. A laboratory investigation links the symptom to the corresponding diagnosis. Alternatively, a symptom can, for example, be linked to a quantitative, nominal or freetext observation.

For symptoms all valid values have to be derived from a predefined set, represented by a code set (see [Code System](#) on page [119](#)). In addition, eHF Record Medical also supports symptoms without such a code to allow the documentation of patient complaints as free text description (for example, that they suffered from a "splitting headache" which is not part of an actual code set).

Substance Administration

Substance administration refers to the distribution of a pharmaceutical substance to the patient. This general concept is represented in eHF Record Medical by an abstract class (see [Figure 119](#)). A manufactured product is in a composition relationship with the substance administration class. Medication and vaccination are supported specializations of a manufactured product. Both of these cover important information needed as a basis for further treatment and could also serve as input for a drug interaction check. Details about the pharmaceutical product (for example, trade name, unique identifier, manufacturer, package size, pharmaceutical form, ingredients) are related to the abstract class `SubstanceAdministration` through the `ManufacturedProduct` class and they are treated the same way for both *medications* and *vaccinations*.

Medication

Medication describes drugs which are either only available on prescription or OTC (over the counter) drugs taken to cure or reduce symptoms of an illness or medical condition. Besides details about the pharmaceutical product dosage instructions and other medication specific information (such as how to administer the medicine [for example, oral, vaginal]) can be documented. In addition, two different types of information can be added to a medication independently of whether the medication was originally created by a medical professional or not:

- Information about why and from when a person stopped taking the medication,
- Information about when the medication was prescribed, received at a pharmacy and additional pharmacy information.

Vaccination

A single *vaccination* can immunize a patient against one disease or a combination of diseases, for example tetanus and diphtheria in one shot. Besides details about the pharmaceutical product, adverse reactions to a vaccination can be documented.

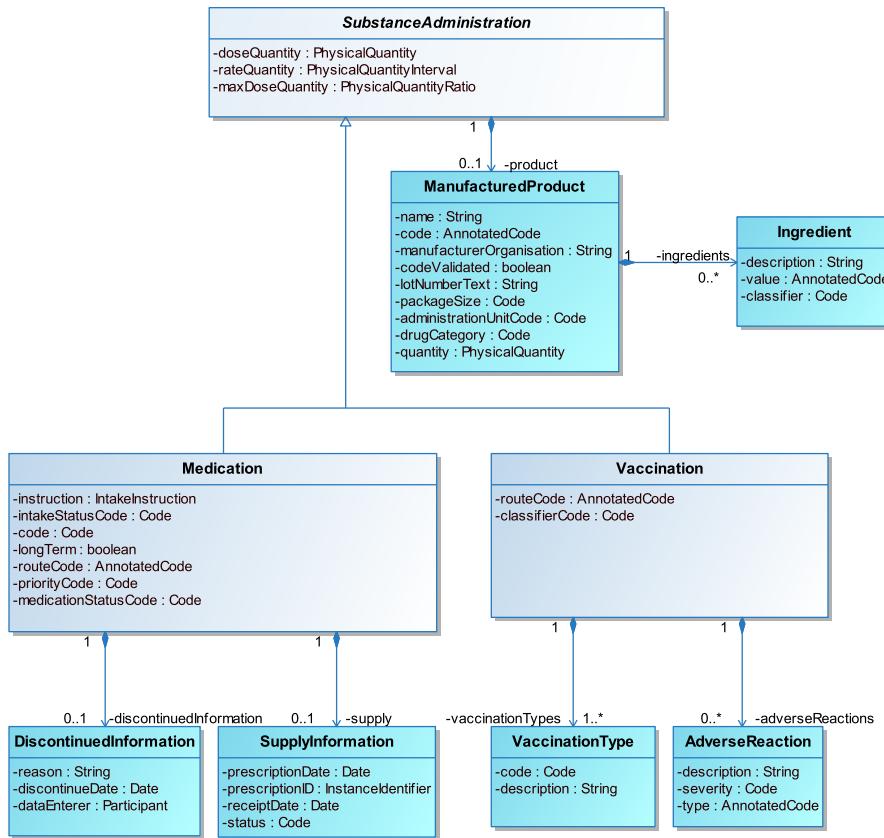
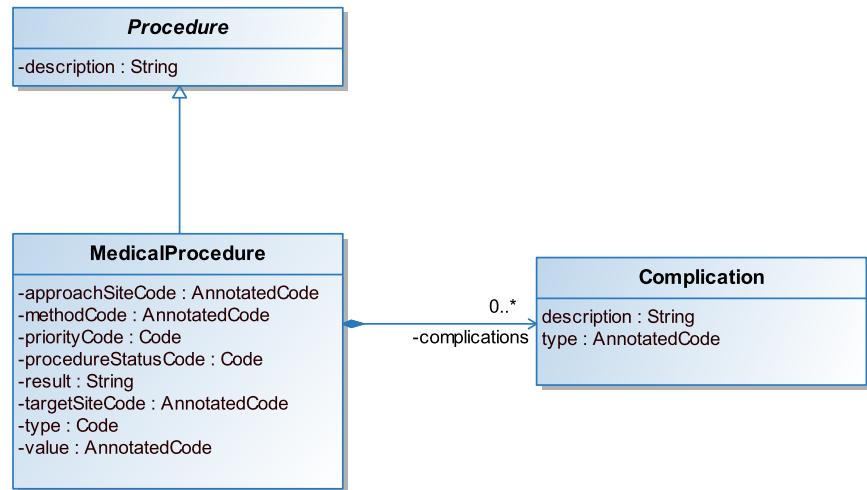


Figure 119: Substance Administration

Medical Procedure

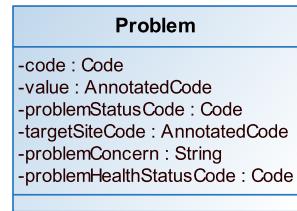
A *medical procedure* is an act whose immediate and primary outcome is the alteration of the physical condition of the subject. For example, medical procedures may involve the disruption of some body surface (for example, an incision in a surgical procedure) conservative procedures such as reduction of a luxated joint, including physiotherapy such as chiropractic treatment, massage, balneotherapy, acupuncture, shiatsu.

**Figure 120:** Procedure

Problem

In general, a *problem* is a situation which is difficult to deal with. A medical problem needs to be classified by a type such as "complaint", "functional limitation" or just "problem". In countries where problem-oriented medical records are common and problem lists are the central entry point to the medical information of a patient a problem might also be used to represent diagnoses and symptoms. In German speaking countries in contrast, it is rather uncommon to find the concept "problem" in medical records. Usually, the specialized concept "diagnosis" as represented within the eHealth Framework by the distinct class "Diagnosis" is used in such health care systems.

Another use case for "Problem" is the documentation of the nursing process which follows several phases like assessment (of patient's needs), diagnosis (of human response needs that nurses can deal with), planning (of patient's care), implementation (of care) and evaluation (of the success of the implemented care). During assessment phase data are collected to identify a patient's nursing problems. For example, these might be of type "diagnosis" (e.g., "Diabetes mellitus"), of type "complaint" (e.g., "Pain in the back after lying in bed the whole day"), or of type "functional limitation" (e.g., "Not able to dress oneself without assistance"). Later within the nursing process, for each problem a measurable goal is set and a nursing care plan with concrete nursing activities is derived.

**Figure 121:** Problem

17.2.2 Dependencies

[Figure 122](#) provides an overview of the dependencies to other eHF modules. The module `ehf-core` provides a set of common data types that are used by business domain objects

of the eHF Record modules, for example `DateDto`. Furthermore, the eHF Record Medical module uses the functionality of `ehf-code-system` and `ehf-commons-expertentry`.

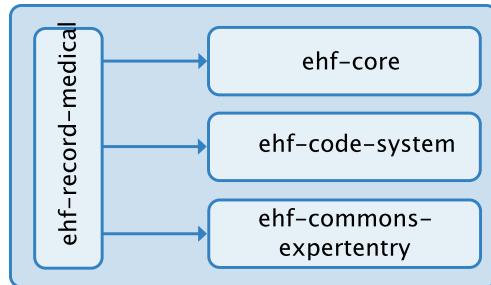


Figure 122: Dependencies of the module eHF Record Medical

17.3 Record Admin

The eHF Record Admin sub-module covers administrative data within the context of a health record. It builds a common basis for both personal health records (aimed at end consumers) and electronic health records (aimed at medical professionals or medical organizations). However, business domain objects or information that is only relevant to one of these two target groups (e.g. relevant billing information) is not regarded as being part of eHF Record Admin.

In addition to the domain model, the module eHF Record Admin provides a number of services for loading and manipulating the domain objects. Both the domain model and the service operations will be covered in the following.

17.3.1 Domain Model

The central class of the domain model of eHF Record Admin is `record`. It contains a `subject`, `health care providers`, `emergency contacts`, and `insurance policies`. [Figure 123](#) illustrates the relationship between the record object and the other administrative domain objects.

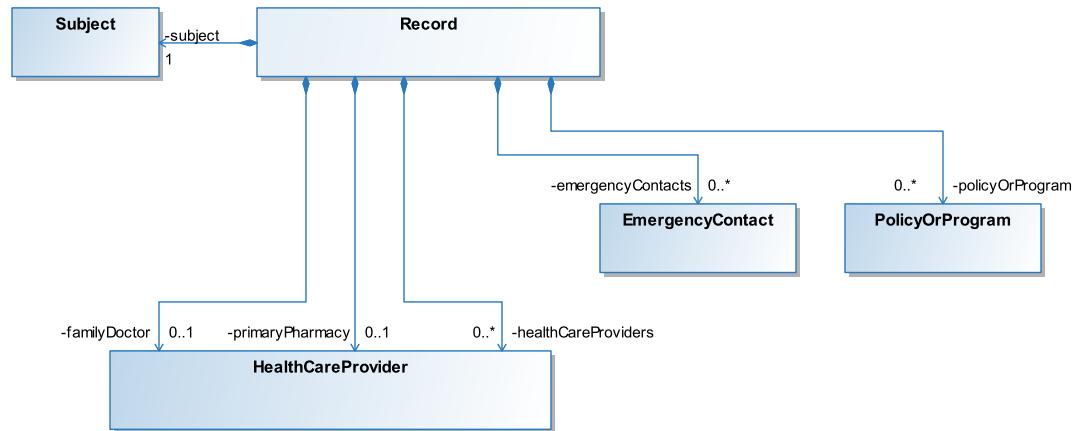


Figure 123: Administrative Business Domain Objects Overview

All of the aggregated parts of `record` are descendants of `contact parties` as shown in [Figure 124](#). A contact party contains several information that is needed to establish contact to a person, for example, by phone, email or fax. For that, a contact party provides

the information *address* and *telecommunication*. A contact party can also belong to an organization that provides *address* and *telecommunication* information accordingly.

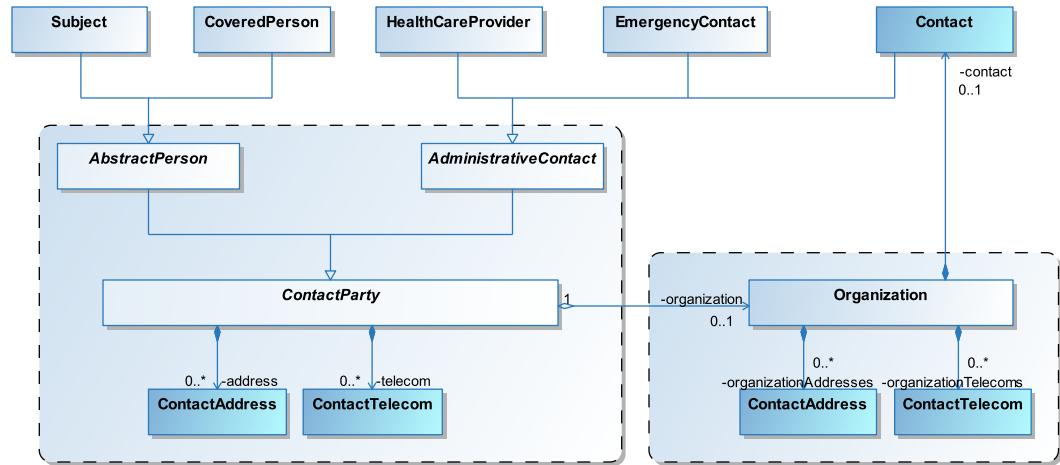


Figure 124: Contact Party Overview

Subject

By definition, each health record contains data about one single person who is regarded as the *subject* of the health record (see [Figure 125](#)). Each piece of information within the record is strongly related to this person, e.g. encounters the subject has had with a doctor, diagnoses about diseases the subject suffers from, information about medications or vaccinations that were carried out with or on the subject.

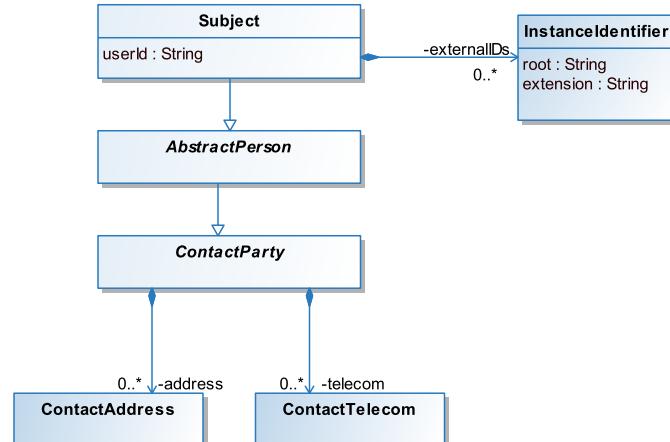


Figure 125: Subject

Even if some information within the health record is about a different person this information is always directly relevant to the medical treatment of the subject. For example, in the case a family member has a disease as there is a greater risk that the subject will also suffer from the same disease (= family risk).

The business domain object offers administrative information relevant for identifying the subject including instance identifiers like insurance or social security number (customizable for each localization), birth date, gender, name and address information as well as different types of contact information like telephone or fax numbers, email addresses and others.

Health Care Provider

Several different types of *health care providers* participate in the medical treatment of a person.

Some of them are only relevant for documentary reasons (e.g. the nurse measuring a blood pressure value) and therefore, are regarded as part of the participant object - the corresponding business domain object from eHF Record Medical. Others are regarded as an important part of the record since they play a major role within the medical treatment process (e.g. the family doctor as the main contact about the health problems of a record's subject and responsible for aftercare following an inpatient encounter in a hospital). The latter are explicitly modeled as independent business domain objects of the administrative data domain.

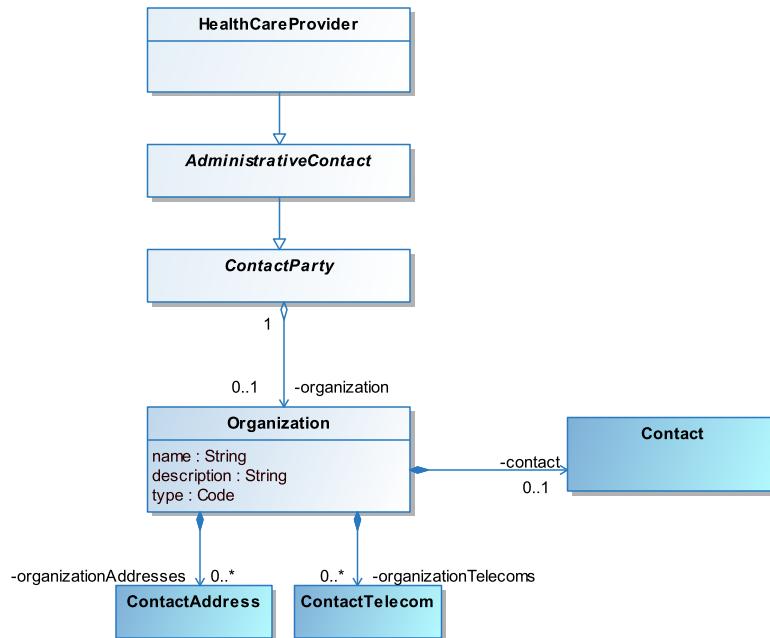


Figure 126: Health Care Provider

It is important to understand that only business domain objects with documentary relevance for the record's content are supported by eHF Record Admin whereas an address book may contain any contact that helps support the user to manage his health himself (or even private entries that have no relationship to his health information at all). Therefore, general address book functionality is not regarded as part of eHF Record Admin and needs to be implemented by a separate module. Currently, eHF Record Admin supports the following subtypes of health care providers: Primary care physician and primary pharmacy.

PrimaryCarePhysician

The *primary care physician* or *family doctor* represents the medical professional who is mainly responsible for the general medical treatment of the record subject. Usually, doctor's letters or other medical information created by a specialist is sent to the family doctor for further processing. Therefore, per record one primary care physician can be created at most.

PrimaryPharmacy

The *primary pharmacy* is regarded as the record subject's main pharmacy. In some scenarios a patient receives benefits if he buys all his pharmaceutical products from one

single predefined pharmacy. Therefore, per record one primary pharmacy can be created at most.

Emergency Contact

An *emergency contact* represents a person who should be immediately informed in case of an emergency. This could be a family member, a friend, a neighbor or any other person related to the record subject (e.g. a caregiver). A record could contain an unlimited set of emergency contacts. Though only one of these has to be defined as the primary one to ensure that in the case of an emergency it is clear who to call first.

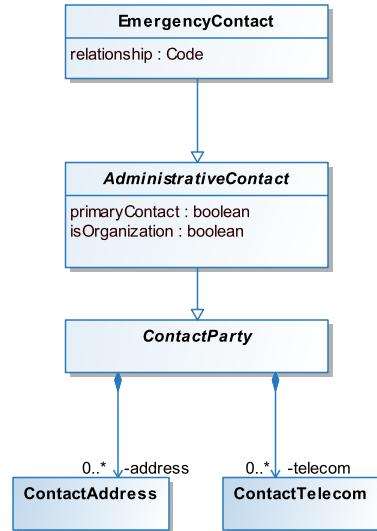


Figure 127: Emergency Contact

Regarding the relationship to address book information the same arguments as discussed within the context of health care provider information apply to emergency contacts as well.

Policy or Program

The **PolicyOrProgram** business domain object covers several types of health insurance and represents an important part of the administrative information about a record subject.

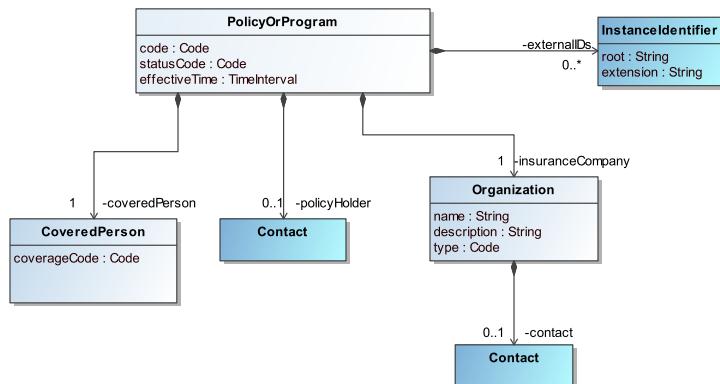


Figure 128: Policy or Program

Besides information about the policy itself administrative information about the covered person, the policy holder and the insurance organization (including information about a contact person) are supported.

The policy holder is the person who owns the insurance policy and pays for it. The covered person is the individual whose health care costs are covered by the insurance policy. Usually, these two roles are associated with the same physical person. In the case of a family relationship the policy holder (e.g. spouse, father) could differ from the covered person (e.g. children).

17.3.2 Module Services

The module eHF Record Admin provides a number of services for loading and manipulating the health record specific domain objects. As with eHF Record Medical, eHF Record Admin does not provide the services itself. Instead, the services are accessible by the public API of the module eHF Record (see [Record Facade](#) on page 275). Most service operations of eHF Record Admin are provided by CRUD services (see also [CRUD Services](#) on page 38). With the help of the CRUD services the domain objects of eHF Record Admin can be created, updated, deleted and read. The complete list of CRUD services of eHF Record Admin is displayed in [Figure 129](#).

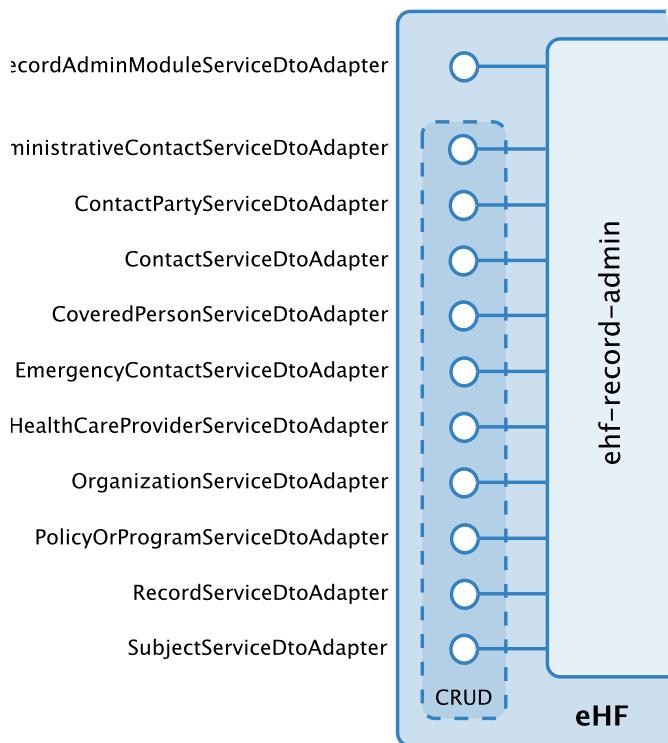


Figure 129: Provided Services of eHF Record Admin

Record Service Operations

The main domain object of eHF Record Admin is `RecordDto`. The following code listing shows the creation of a record by the CRUD service `RecordServiceDtoAdapter`:

```

ContactAddressDto contactAddress = new ContactAddressDto();
contactAddress.setCity("Walldorf");
contactAddress.setPostalCode("69190");
contactAddress.setStreet("Industriestra\u00dfe 41");
contactAddress.setScope(TEST_SCOPE);

SubjectDto participantDto = new SubjectDto();
participantDto.setGivenName("Test_Name");
participantDto.setFamilyName("Test_FamilyName");
participantDto.setUserId(USER_ID);

```

```

participantDto.addAddress(contactAddress);

recordDto = new RecordDto();
recordDto.setScope(TEST_SCOPE);
recordDto.setSubject(participantDto);

recordDto = recordAdminRecordService.create(recordDto);

```

While the CRUD services provide operations that always follow the same generic pattern, the module service adapters provide custom service operations. The following example code demonstrates both possibilities of retrieving record objects by the service `RecordAdminModuleService`: First, the module service operation `findRecordByUserId` requires the user identification as a parameter. Second, the service operation `loadById` of the record CRUD service requires the record identification.

```

RecordDto recordDto = recordAdminModuleService.findRecordByUserId("ervin");
assertNotNull(recordDto);

recordDto = recordAdminRecordService.loadById(recordDto.getId(), true);
assertNotNull(recordDto);
assertNotNull(recordDto.getSubject());
assertEquals("ervin", recordDto.getSubject().getUserId());

```

Both operations above load the record object and all of its aggregated parts such as the subject, health care providers, and emergency contacts. However, loading of a record and all of its dependent objects can be time consuming. Some use cases require a faster loading of records. A health care professional, for example, who wants to get an overview of all records he has access to. In order to realize that, eHF Record Admin provides along with the info object `RecordInfo` a second, more lightweight, view on a record that only contains the most important administrative information as displayed in [Figure 130](#).

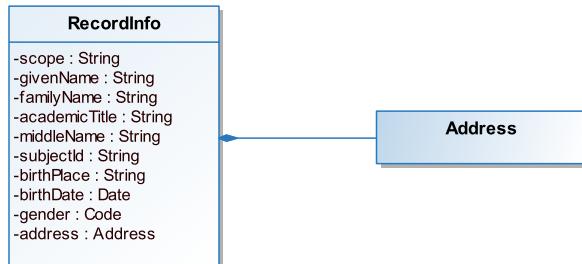


Figure 130: RecordInfo

The corresponding service operation `findAccessibleRecords` lets you access the available record information. The following listing shows a code fragment of how to use this operation. The retrieved records depend on the currently authenticated user, that is the user with the identification "ervin" in the example:

```

Subject subject = new Subject();
subject.getPrincipals().add(new UserPrincipal("ervin"));

PrivilegedAction<List<RecordInfoDto>> action =
    new PrivilegedAction<List<RecordInfoDto>>() {

        public List<RecordInfoDto> run() {
            List<RecordInfoDto> records =
                recordAdminModuleService.findAccessibleRecords();

            assertNotNull(records);
            assertTrue(records.size() > 0);

            RecordInfoDto recordInfo = records.get(0);

```

```

        assertNotNull(recordInfo.getFamilyName());
        assertNotNull(recordInfo.getGivenName());
        assertNotNull(recordInfo.getScope());
        assertNotNull(recordInfo.getSubjectId());
        assertNotNull(recordInfo.getAddress());
        assertNotNull(recordInfo.getAddress().getCity());
        assertNotNull(recordInfo.getAddress().getPostalCode());

        return records;
    }
};

Subject.doAsPrivileged(subject, action, null);

```

Subject Service Operations

The next category of service operations is related to the domain object `SubjectDto`. The operation `findSubjectsByCriteria` finds all subjects that are, on the one side, accessible by the current users, and on the other, match the filter criteria specified in the parameter object `SubjectSearchCriteria`. The `PageQualifier` provides page context information to the services. The order of the result list can be determined by the object `OrderCriterion`.

```

final SubjectSearchCriteria criteria = new SubjectSearchCriteria();
criteria.setFamilyName("ervin");

OrderCriterion order = new OrderCriterion();
order.setProperty("givenName");
order.setAsc(false);
OrderCriterion orderCriteria[] = new OrderCriterion[] { order };

final PageQualifier qualifier = new PageQualifier();
qualifier.setOrderCriteria(orderCriteria);

Subject subject = new Subject();
subject.getPrincipals().add(new UserPrincipal("ervin-1"));

PrivilegedAction<PagedResult<SubjectDto>> action =
    new PrivilegedAction<PagedResult<SubjectDto>>() {

        public PagedResult<SubjectDto> run() {
            PagedResult<SubjectDto> result =
                recordAdminModuleService.findSubjectsByCriteria(
                    criteria, qualifier);
            return result;
        }
    };

PagedResult<SubjectDto> pagedResult =
    Subject.doAsPrivileged(subject, action, null);

pagedResult.getTotalNumberOfObjects();

```

Emergency Contact Service Operations

Finally, the record module service adapter provides a number of service operations that are connected to emergency contacts, namely `addEmergencyContact`, `setPrimaryEmergencyContact`, and `deleteEmergencyContact`. In section (1) of the following listing, two newly created emergency contacts are added to the record. The operation `addEmergencyContact` saves the emergency contact to the persistence store and links it to the given record. Furthermore, if the record still has no emergency contacts, the emergency contact that will be added becomes the primary contact automatically. In section (2), the call to `setPrimaryEmergencyContact` changes the primary contact to the second one. Please note that only one emergency contact can be primary. After the record object is re-loaded with the CRUD service in (3), the example shows in section (4) how to delete an emergency contact again.

```
// (1) add two emergency contacts
recordAdminModuleService.addEmergencyContact(recordDto,
    createEmergencyContact());

EmergencyContactDto emergencyContact =
    recordAdminModuleService.addEmergencyContact(recordDto,
        createEmergencyContact());

// (2) change the primary emergency contact
recordAdminModuleService.setPrimaryEmergencyContact(recordDto, emergencyContact);

// (3) load the emergency contacts of the record
recordDto = recordAdminRecordService.loadById(recordDto.getId(), true);
Set<EmergencyContactDto> emergencyContacts = recordDto.getEmergencyContacts();

// (4) delete the first emergency contact
EmergencyContactDto emergencyContactToDelete = emergencyContacts.iterator().
next();
recordAdminModuleService.deleteEmergencyContact(recordDto,
    emergencyContactToDelete);
```

17.3.3 Dependencies

Figure 131 provides an overview of the dependencies to other eHF modules. The module `ehf-core` provides a set of common data types that are used by business domain objects of the eHF Record modules, for example `DateDto`. Furthermore, the eHF Record Admin module uses the functionality of `ehf-code-system` and `ehf-commons-expertentry`.

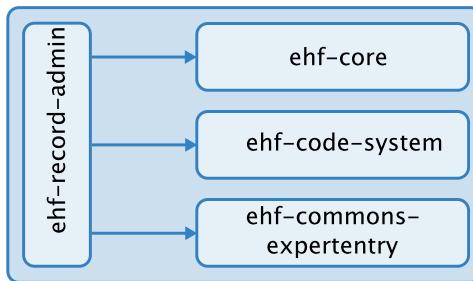


Figure 131: Dependencies of the module eHF Record Admin

17.4 Document

In the health sector more and more medical information is stored in electronic documents. Examples of such documents are discharge letters, medical histories, ambulatory visit reports or X-ray photographs. In order to manage such documents, systems have to support tasks such as storing, retrieving, organizing or finding documents. The application module eHF Document is responsible for document management within the eHealth Framework and should be part of any health care application built on top of eHF. It makes use of [HL7 v3 RIM](#) (Health Level 7 Version 3 Reference Implementation Model) artifacts making it easier to interact with other HL7 based systems.

A document is a file containing any kind of content. The content can be of binary nature (e.g. pictures), textual nature (e.g. plain text), or structured content such as XML documents. The domain model is capable of defining document hierarchies and multiple versions of a document. Additionally, metadata such as MIME type, author, version number, or creation date is stored for documents. Structured documents can be associated with schemas and style sheets making it easy to transform a document using technologies such as [Extensible Stylesheet Language](#) (XSL) for rendering or printing. The eHF Document module provides a set of general service operations for uploading, downloading, and

retrieving documents. Both, the domain model and the service operations, are covered in the following.

17.4.1 Domain Model

The eHF Document domain model consists of two important concepts: context objects for accessing documents and qualifier for identifying and referencing documents.

Context

Documents including their specific metadata and content are managed by context domain objects as illustrated in [Figure 132](#).

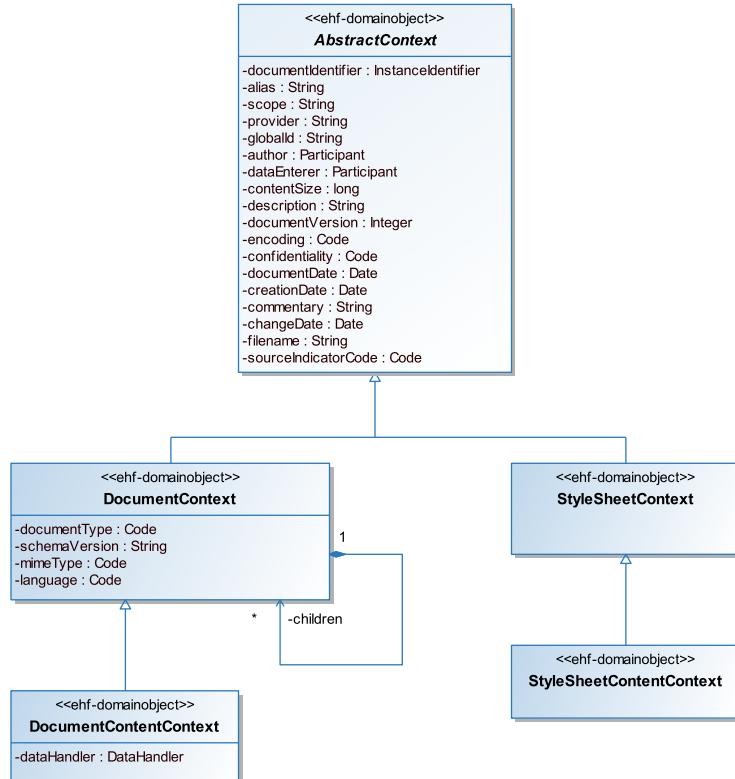


Figure 132: Document Context Hierarchy

With the help of the domain object `DocumentContext`, you can access the metadata of documents. The metadata is retrieved as attributes of the class.

The domain object `DocumentContentContext` extends `DocumentContext` by adding a data handler for managing document content. The access to content is always streamed in order to satisfy the low-memory profile objective. At no stage document content is loaded into memory.

The domain model of eHF Document separates stylesheets from actual documents. The attributes that are common for documents and stylesheets are defined in the abstract class `AbstractContext`. Examples of some attributes are *author*, *creation date*, and *document version*.

As with the domain object `DocumentContext` the `StyleSheetContext` contains the metadata of a stylesheet by implementing the class `AbstractContext`. Please note that it does not add any additional metadata.

The class `StyleSheetContentContext` contains the content of the stylesheet. It is a subclass of `StyleSheetContext` and adds the `data handler` attribute to handle the content.

Qualifier

In order to work with documents it is necessary to be able to reference single instances. The `DocumentQualifier` is an abstract class which is used to uniquely identify a document (see [Figure 133](#)). The document context contains four qualifier for referencing the root document, the stylesheet for rendering and printing and a thumbnail.

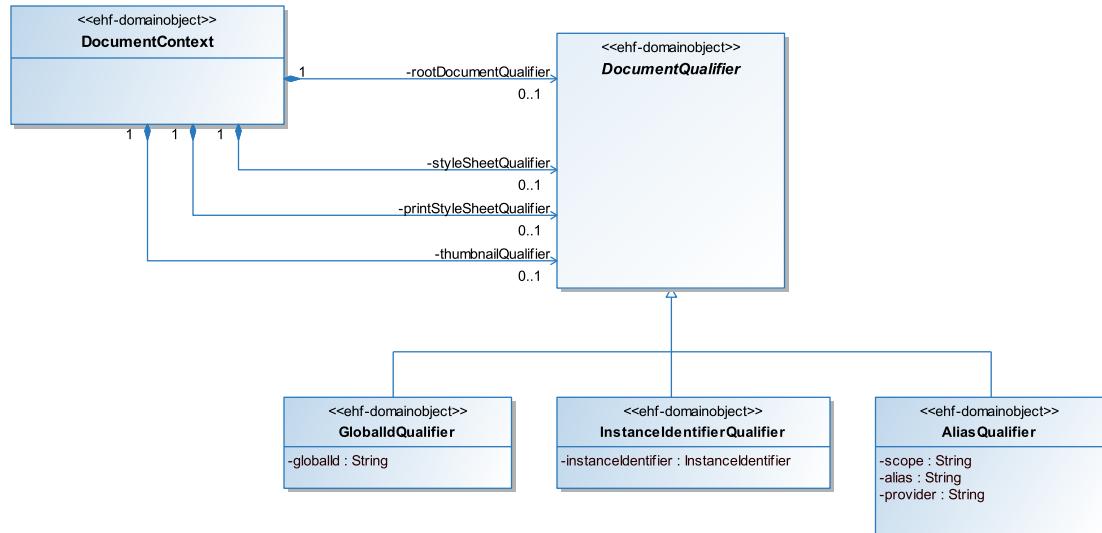


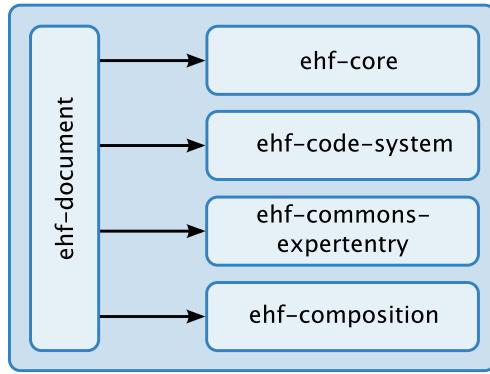
Figure 133: Qualifier Hierarchy

Three different qualifiers are available, which are subclasses of `DocumentQualifier`:

- An *instance identifier qualifier* consists of an object identifier (OID) plus extension. It is not assigned by the eHF Document module but is sent by the uploading client application. The OID is a unique identification assigned from the eHF, while the extension is a unique ID from the client application. Both together, the OID and the extension, build a globally unique identifier for a document.
- A *global identifier qualifier* is a document qualifier that identifies a document by its individual identification. On uploading a document, the eHF assigns an ID to the document that is unique within the eHF based application.
- An *alias qualifier* allows you to specify custom identification criteria. A client application can provide an alias, provider, and scope on creation of a document. Both alias and provider can be specified arbitrarily by the client. However, the scope has to be existent in the persistence. Alias, provider and scope uniquely identify a document.

17.4.2 Dependencies

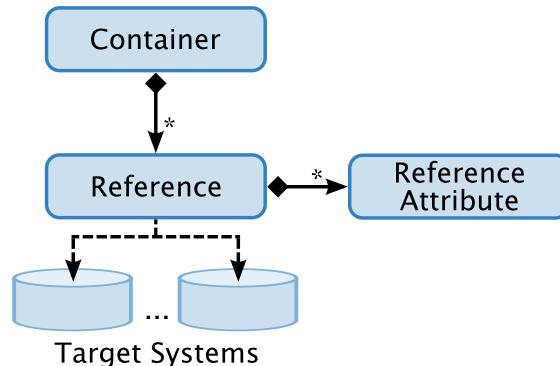
[Figure 134](#) provides an overview of the dependencies to other eHF modules. The module `ehf-core` provides a set of common data types that are used by business domain objects of the eHF Document modules, for example `DateDto`. Furthermore, the eHF Document module uses the functionality of `ehf-code-system`, `ehf-commons-expertentry`, and `ehf-composition`.

**Figure 134:** Dependencies of the module eHF Document

17.5 Composition

eHF Composition provides container-based entity management for eHF applications. The center pieces of eHF Composition are containers which hold references to entities within different target systems. Thereby, it should not be distinguished between application-internal and external entities (held in remote systems). Containers provide an aggregated view on a set of medical entities which are logically grouped within the context of a container. This module provides services to maintain containers. Furthermore, it allows you to link and also unlink references to- and from container instances. Via the API, reference queries can be requested, to get up-to-date representations of all references based in a specific container.

Containers are always of a specific type, which typically represent a use case (e.g. an extended emergency data set container). References uniquely identify the target entity by holding a unique reference ID. Optionally, references can provide the type information of the targeted entity to the clients. References can carry multiple reference attributes which provide additional information to the module users.

**Figure 135:** eHF Composition Overview

References are obtained by Reference Providers. They are in charge of retrieving references for reference queries. Instead of containing the real entities, containers only contain references to them. Reference Resolvers are responsible for resolving references into the real target entities. Both, Reference Providers and Reference Resolvers need to be registered with the composition module to participate in the reference retrieval and resolution process.

17.5.1 Module Services

eHF Composition module service supports the following methods:

- Creating, reading, updating and deleting containers containing references and their reference attributes is available through the generic CRUD interface.
- Finder methods allow you to retrieve existing container instances from eHF Composition.
- Query methods allow you to query for references within a given container. Such a query returns all references to a container, which match the provided query criteria. By formulating a query to eHF Composition, all registered Reference Providers are consulted to retrieve references which match the given query criteria.
- Obtained references could be resolved into the real target entities by providing the references to the eHF Composition service. During reference resolution, eHF Composition delegates requests to registered reference resolver implementations to retrieve the original entity of the references.

17.5.2 Architectural Design

In order to link and unlink arbitrary objects, i.e. to add and remove references to and from a container, eHF Composition provides a set of extension points. These are interfaces that can be implemented to provide the means for retrieval and resolution of references to specific object types.

- *ReferenceProvider*: Creates references to business entities
- *ReferenceResolver*: Resolves entity references and returns the referenced entities
- *ReferenceQuery*: Marker interface which marks an object as a reference query

The *ReferenceProvider* interface defines one single method to retrieve (i.e. create), references to specific objects. The objects, which references are to be retrieved for are specified by a custom *ReferenceQuery*. How this is done depends on the particular implementation. The simplest possible implementation gets the very object for which the reference is to be retrieved wrapped in a *ReferenceQuery*, creates the reference and return it. The references are of type *ReferenceDto*

Once references are obtained they can be resolved to the original objects using a custom *ReferenceResolver*. This interface too, defines only one single method which receives a set of references and returns a set of resolved business entities.

If a *ReferenceProvider* is not able to create a reference for an object it throws a *ReferenceProvisionException*. Accordingly, if a reference cannot be resolved, the *ReferenceResolver* throws a *ReferenceResolutionException*.

eHF Composition already ships with an implementation of *ReferenceProvider*, *ReferenceResolver* and *ReferenceQuery* suitable for eHF based applications. The *EhfDtoReferenceHandler* is an implementation of both the *ReferenceProvider* and *ReferenceResolver* interfaces for eHF *TransferObjects*. The corresponding *ReferenceQuery* called *EhfDtoReferenceQuery* wraps a single *TransferObject* for which a reference is to be retrieved.

17.5.3 Configuration

Assembly Integration

First, your module or application has to define and export a *ReferenceProvider* and a *ReferenceResolver*, say *referenceProvider1* and *referenceResolver1*.

Second, you have to configure your *ReferenceProviders* and *ReferenceResolvers*. This is done by defining for each a bean of the types *CompositeReferenceProvider* and

CompositeReferenceResolver respectively, as shown in listing 1 and 2. Both classes already ship with eHF Composition.

Listing 1

```
<bean id="referenceProvider"
      class="com.icw.ehf.composition.impl.provider.CompositeReferenceProvider">
    <property name="referenceProviderMap">
      <map>
        <entry key="com.mycompany.MyReferenceQuery1">
          <list>
            <ref bean="referenceProvider1" />
          </list>
        </entry>
        <entry key="com.mycompany.MyReferenceQuery2">
          <list>
            <ref bean="referenceProvider2" />
          </list>
        </entry>
      </map>
    </property>
</bean>
```

The `referenceProviderMap` defines the mapping of `ReferenceQuery` types to `ReferenceProviders`.

Listing 2

```
<bean id="referenceResolver"
      class="com.icw.ehf.composition.impl.resolver.CompositeReferenceResolver">
    <property name="referenceResolvers">
      <set>
        <ref bean="referenceResolver1"/>
        <ref bean="referenceResolver1"/>
      </set>
    </property>
</bean>
```

This reference resolver gets injected into the Composition Module Service DTO Adapter.

The specific providers and resolvers (i.e., `referenceProvider1`, etc.) are contributed by the concerned module (see above).

Then, the mapping from container types (codes) to the central reference providers has to be defined and the Map injected into the `CompositionModuleService`

Listing 3

```
<bean id="containerCodeToReferenceProviderMap"
      class="org.springframework.beans.factory.config.MapFactoryBean">
    <property name="sourceMap">
      <map>
        <entry key="EXL/EXAMPLE_ID/1.0.0" value-ref="referenceProvider" />
      </map>
    </property>
</bean>

<ehf:definition-extension target-module-id="compositionModule"
                           target-bean-name="compositionModuleServiceTarget">
  <ehf:property name="containerCodeToReferenceProviderMap"
                ref="containerCodeToReferenceProviderMap" />
</ehf:definition-extension>
```

Moreover, the `CompositionModuleServiceDtoAdapter` has to be propagated as an extension.

Listing 4

```
<ehf:definition-extension target-module-id="compositionModule"
```

```

target-bean-name="compositionModuleService">
  <ehf:property name="referenceResolver"
    ref="referenceResolver" />
</ehf:definition-extension>

```

17.5.4 Authorization

Using the functionality of the eHF Composition module requires the extension of the instance-based access control model that is provided by the eHF Authorization module. eHF Composition functionality provides inter-module links between objects of arbitrary eHF modules. Using this functionality, for example, arbitrary medical objects and documents can be composed in a container object. Further, eHF Composition provides functionality to create and manage containers, to put objects into a container and to retrieve objects from the container. The eHF Composition module provides the *Reference* class and the *Container* class. Via the *Reference* class, a *Container* references objects in other eHF modules. For example instances of BMD objects and *Document* objects can be linked to a *Container* in this way.

To enable a user to access a Container instance and its associated Reference instances, the current instance-based access control functionality provided by eHF Authorization can be used. However, if the same user does not have instance-based permissions on the referenced BMD objects and documents, then this user is not able to access these objects. At this point, a new access control model is introduced to enable a user to access all objects referenced by a specific container without requiring this user to have instance-based permissions on these objects. The concept of this requirement is visualized by the next figure.

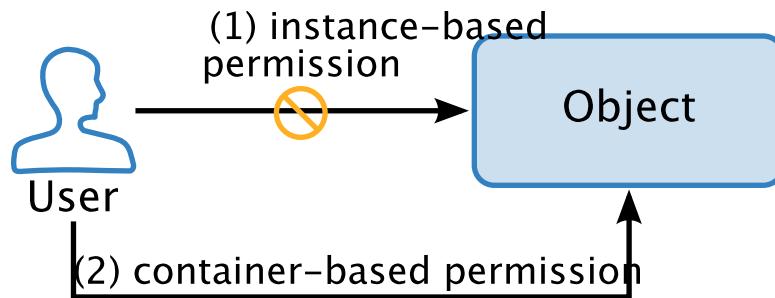


Figure 136: Container-Based Permission Concept

As opposed to the instance-based access control model, this new access control model is called a “container-based access control model”. As the figure above shows, container-based permissions may allow access to an object, even when instance-based permissions would not allow access to this object.

Integration into the eHF Access Control Framework

The next figure shows a class diagram that explains how the container-based access decision logic can be integrated into an eHF-based application using the eHF Access Control Framework.

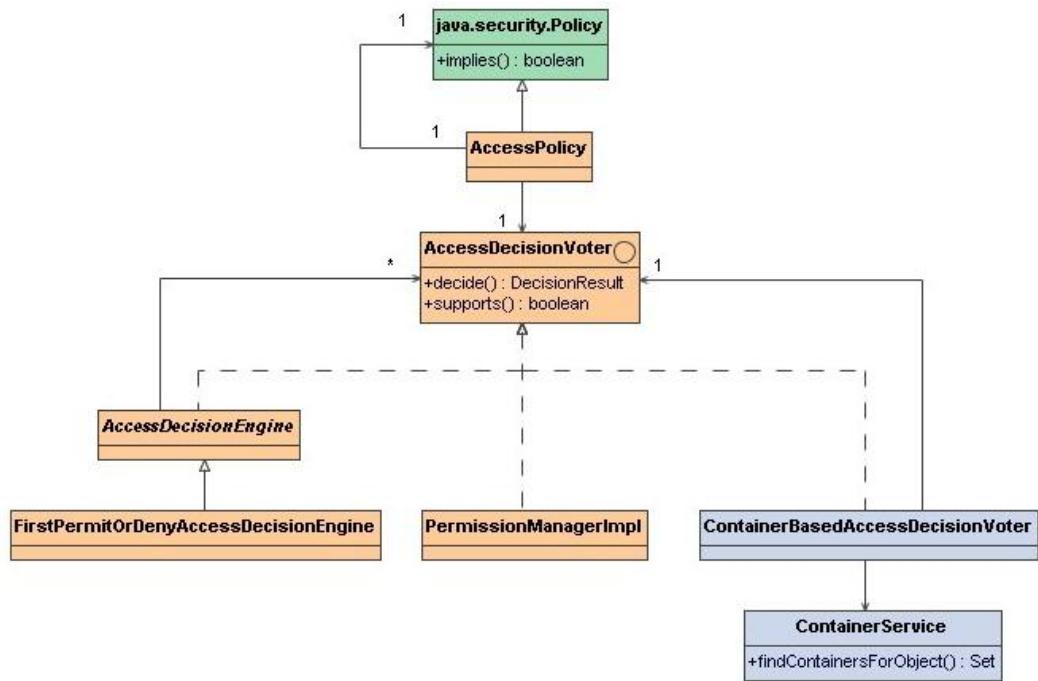


Figure 137: Access Control Class Diagram

The eHF Access Control Framework is build on top of the security infrastructure provided by Java Standard Edition Security (JSE Security). The *Policy* class represents the Policy Decision Point (PDP) of JSE Security. The *implies* method of the *Policy* class is called by the JSE Security infrastructure to initiate the access decision. The *ProtectionDomain* parameter of the *implies* method holds information about the current user. The eHF Authorization module provides the *com.icw.ehf.authorization.policy.AccessPolicy* class as an extension of the JSE *Policy* class. The *AccessPolicy* class delegates access decision requests to an implementation of the *AccessDecisionVoter*. An implementation of the *AccessDecisionVoter* typically represents the access decision logic of a single access control model like the instance-based access control model or the container-based access control model. If an eHF-based application needs more than one access decision voter, then an implementation of the abstract *AccessDecisionEngine* is used, which is a composite access decision voter. The *AccessDecisionEngine* combines the access decision results of all configured access decision voters into a final access decision result.

The eHF Assembly module is configured to use an instance of the *PermissionManagerImpl*, which provides instance-based access decision logic and an instance of the *ContainerBasedAccessDecisionVoter*, which provides container-based access decision logic. The results of both access decision voters are combined by the *FirstPermitOrDenyDecisionEngine*. The combination algorithm of this access decision engine is explained in the section.

Please note that the *ContainerBasedAccessDecisionVoter* calls the *ContainerService* provided by the eHF Composition module to get the attributes relevant to make a container-based access decision. The logic of the container-based access decision voter is explained in the section.

Container Based Access Decision Voter

The container based access decision voter needs several attributes to make a container based access decision. These attributes are:

- Resource attributes:

The unique identifier of the accessed object is needed. It is possible to find all containers that have a link to the object using this unique identifier. This identifier is contained in the *DomainPrivilege* object which represents the requested permission. This permission object is provided by the eHF Authorization module and is an extension of the JSE *Permission* class.

- Performed action:

This information is also contained in the *DomainPrivilege* object.

- User attributes:

We get information about the requesting user, that is the user's *UserPrincipal* and his *RolePrincipals* from the JSE security infrastructure via the *ProtectionDomain* parameter offered by the *implies* method of the JSE *Policy* class.

- Container attributes:

These attributes are not provided by the eHF Authorization infrastructure and must be collected by the container-based access decision voter itself. The details are discussed next.

As described above, a user may be allowed to access an object, even if the user does not have instance-based access rights for this object. To be more specific, this is the case when the accessed object is linked to a container the user has container access rights to. To make a container-based access decision, first all containers the object is linked to are collected. Next, there is a check to see if the requesting user has the required container access right on any of the found containers. If at least one container can be found that the user has the required container access right to, then the access is permitted, otherwise the container-based access decision voter does not permit the access.

Container Service

As stated above, the first task of the container-based access decision voter after it has been invoked is to collect the decision relevant attributes from all containers holding a link to the requested object. To do this, the voter calls the service method *findContainersForObject* of the *ContainerService* provided by the eHF Composition module which returns all containers holding a link to the given object.

Access Control List (ACL)

The implementation of the container based access decision model uses the existing instance-based ACL infrastructure. That is, the existing eHF Authorization XML syntax used to define the instance-based access control policies is also used to define the container-based access control policies. The assignment of a container-based access permission to a user is specified as in the following XML snippet. This assignment allows a user to access the content of a container, that is, to access all objects linked to that container, whether the user has instance-based access permissions to these objects or not. The "ContentOfContainerType:" prefix used in the *<type>* element marks this permission as a container-based permission. The *<identifier>* element could be used to restrict the permission to a single container instance. The *<role>* element and the *<context>* element currently do not have a meaning for container-based permissions and should therefore contain the wildcard value ("*"). Please note that the permission assignment in the XML snippet below does not give the user any access permissions to the container object itself or to reference objects associated with the container object. These permissions must be assigned to the user separately using instance-based access permission assignments.

```
<assignment>
    <domainProfile>
        <profileReferences />
        <profileStale>false</profileStale>
        <profile>
```

```

<permission>
    <target>
        <type>ContentOfContainerType:MY_CONTAINER</type>
        <role>*</role>
        <context>*</context>
        <identifier>*</identifier>
    </target>
    <actions>
        <action>READ</action>
    </actions>
</permission>
</profile>
</domainProfile>
</assignment>

```

Permission Profiles

The eHF Authorization module supports the concept of permission profiles. A permission profiles is a named collection of permissions that can be assigned to a user in a single assignment operation using the name of the profile. Container-based permissions can use the permission profile concept as well. In fact, it is even possible to mix instance-based permissions and container-based permissions in a single profile. This is possible because we identify container-based permissions by a specific prefix used in the `<type>` element. We use the prefix “ContentOfContainerType:” to specify container-based permissions.

Access Decision Logic

Once the container-based access decision voter has obtained a list holding all containers that have a link to the requested object, the decision voter initiates the actual decision logic to find out if the user has the requested permission to access the object. To understand how this decision logic is implemented, let's first look at how the current instance-based decision logic is implemented. As mentioned above, the eHF Authorization module is based on the JSE Security infrastructure. One key element of this infrastructure is the `java.security.Permission` class, which represents an access right to a system resource. eHF Authorization provides an extension of this class, which is `com.icw.ehf.authorization.domain.DomainPrivilege`. This class holds all access decision relevant attributes. Further, this class overwrites the `implies` method of the Java `Permission` class, that checks if the `Permission` given as a parameter to this method is implied by the `Permission` that is called. To find out if a user has the requested access right, the `AccessPolicy` basically iterates over all permissions that are assigned to the requesting user and calls the `implies` method on all these permissions providing the requested `Permission` as the method's parameter. If at least one `implies` method returns `true`, then the access can be permitted (deny logic is equivalent to that). From this implementation point of view, the access decision logic for container-based access control is very similar. In fact, the container-based access decision voter reuses the current `Permission` class implementations of eHF Authorization. The difference to instance-based access control is that the requested `Permission` objects are not provided by the `SecurityService` directly. Instead, these `Permission` objects are created by the access decision voter itself. This is due to the fact that the `SecurityService` does not know all attributes necessary for a container-based access decision. As described above, the missing attributes are collected by the access decision voter itself. Therefore, the access decision voter must create the requested permissions itself. Basically, the container-based access decision voter creates one requested `Permission` object per container that it has collected before. The logic for creating these `Permission` objects is pretty simple. All necessary attributes, namely the type, the domain and the identifier attribute can be found in the `Container` objects. The type is prefixed with the well-known prefix string for container-based access control, in our case “ContentOfContainerType:” (see above). The context and the role attributes are set to wildcards, because they are currently not used for container-based access control. For each `Permission` object created that way, the logic is now equivalent to instance-based access control described above.

ACL Caching

Container-based access decision functionality puts a lot of overhead on each and every object access in an application. In fact, depending on the amount of containers used in an application, the overhead could be significantly higher compared to instance-based access control. The eHF Authorization module provides a session-scoped ACL cache. This cache is implemented by the `com.icw.ehf.authorization.service.PermissionCache` class and is currently used by the instance-based access control functionality. To be more specific, the `com.icw.ehf.authorization.service.PermissionManagerImpl` class uses this cache as it provides the actual implementation of the current instance-based access decision logic. As described above, from an implementation point of view, the logic for instance-based access decisions and container-based access decisions are identical after the point where the container-based access decision voter has created the requesting `Permission` object. Further, as also described above, both, instance-based access control and container-based access control use the same ACL. That is, the container-based access decision voter uses the same instance of the `PermissionManagerImpl` class that is used by the instance-based access control voter. So the container-based access decision voter reuses the already existing ACL cache instance.

ContainerCaching

Finding all the containers holding a reference to a specific object is relatively time-consuming, because typically the database will be accessed during this operation. To avoid accessing the database for every container-based access decision request, the container-based access decision voter uses a container cache. This container cache holds all Containers for a specific object, and respectively for a specific object identifier. The container cache is populated lazily during a container-based access decision request. A cache entry is deleted from the cache whenever the associated object is added to or removed from a container. When a container is deleted, then all cache entries with this container are deleted from the cache.

To reduce the memory footprint, the cache does not store complete `Container` objects. Instead, only the access decision relevant attributes of a container are stored. These attributes are called permission qualifiers and are represented by the `PermissionQualifier` class. That is, a cache entry consists of an object identifier as the key element and a set of permission qualifiers as the value element. Therefore we call the cache a permission qualifier cache.

The permission qualifier cache implements the `PermissionQualifierCache` interface. The `PermissionQualifierCacheEhImpl` class is a EHCAFE-based implementation of the permission qualifier cache. This EHCAFE-based implementation is configured using the `ehcache-permission-quaifier.xml` file that must be in the classpath. The eHF assembly project contains a sample configuration file that demonstrates cache replication in a cluster environment.

As described above, when an object is added to or removed from a container or when a container is deleted, then the respective cache entries must be deleted. This functionality is implemented by the `PermissionQualifierCacheInterceptor` class which is configured as an interceptor of the `ContainerServiceImpl` class.

Policy Administration Point (PAP)

The PAP consists of services to manage the policy data and to provide that data to the PDP. For the instance-based access control model this functionality is implemented by the class `com.icw.ehf.authorization.service.PermissionManagerImpl`. Because container-based access decision functionality reuses the existing instance-based ACL infrastructure, we also reuse this class for the management of container-based access control.

Assembly Integration

The container-based access decision voter is integrated into an application (assembly) via Spring configuration. The Spring bean `containerBasedAccessDecisionVoter` is exposed by the eHF Composition module. This bean must be configured to use the instance based access decision voter (see next listing).

```
<ehf:definition-extension target-module-id="compositionModule"
    target-bean-name="containerBasedAccessDecisionVoter">
    <ehf:property name="instanceBasedAccessDecisionVoter" ref=
    "accessManagerTarget" />
</ehf:definition-extension>
```

Further, the access decision engine must be configured to use the voters. This is shown in the next listing.

```
<bean id="accessDecisionEngine"
    class="com.icw.ehf.authorization.voter.FirstPermitOrDenyDecisionEngine">
    <property name="voters">
        <list>
            <ref bean="accessManagerTarget"/>
            <ref bean="containerBasedAccessDecisionVoter"/>
        </list>
    </property>
</bean>
```

A sample configuration can be found in the eHF Assembly configuration file `ehf-access-decision-context.xml`.

As described above, the container-based access decision voter uses a permission qualifier cache. The current implementation of the cache uses EHCACHE technology and can be configured in the `ehcache-permission-qualifier.xml` file.

Appendix

A UML Profile for eHF Generator

This chapter lists all available stereotypes of the eHF UML profile and details their supported concepts. The description of each stereotype both summarizes all available tagged values plus the native UML concepts the stereotypes support in the scope of the eHF.

A.1 Stereotype `ehf-domainobject`

Stereotype Description

The `ehf-domainobject` stereotype can be used to mark a class as an `ehf-domainobject` class. Moreover, the following tags can be used to further qualify the characteristics of the domain object class instances.

Tag Description

Tag	Description
<code>active</code>	This boolean flag defines the state of the domain object. By default, domain objects are marked as <code>true</code> . By setting this flag to <code>false</code> , the generator will suppress the persistence of the domain object. It can be used to mark domain objects as work in progress.
<code>classification</code>	With this tag the classification of the attributes is provided on the class level. This setting will function as a default that can be overwritten by the individual attributes. For more details see the <code>ehf-attribute</code> stereotype documentation.
<code>classifier</code>	Used to specify the permission classifier for the domain object. The permission classifier is by default <code>\${package}.Object</code> . If the tag is modified the package will be preserved and the annotation added, such as: <code>classifier=Example permission classifier: '\${package}.Example'</code> . The permission classifiers are used to define permission profiles on the model.
<code>contractRelevant</code>	Boolean tag indicating that the <code>ehf-domainobject</code> is considered a contract relevant object class. Objects of this type may neither be updated nor deleted by a user, since they represent contract relevant information.
<code>crud</code>	This boolean flag indicates whether a CRUD service implementation should be provided for the domain object. If this tag is not set (default value='false') no CRUD service is generated.

custom	<p>The <code>custom</code> tag is where you can specify on which layer full control over the generated classes is required. Currently we support the layer <code>domainobject</code>, <code>dto</code>, <code>xto</code>, <code>service</code>, and <code>servicesecurity</code>. In future versions more layers may be supported. If multiple layers are required multiple values can be provided using , as separator.</p>
	<p>Note: DEPRECATED: This tag has been deprecated. Please note that since version 2.0.0 it is possible to simply copy over customizable classes (those which are allowed to be customized) from the <code>src/gen</code> folder to the <code>src/main/java</code> folder. The generator will detect this and not duplicate the class anymore in the <code>src/gen</code> folder. This feature should provide more flexibility when working with the generator.</p>
directive	<p>This tag specifies the default directive for classification handling on attributes. The class-level default applies to all attributes and can be overwritten individually on the attribute level. For more details see the <code>ehf-attribute</code> stereotype.</p>
entity	<p>Domain objects can generally be divided in two groups: entities, which have a database identity and value objects without a database identity. Putting objects into one of these groups is normally a modeling concern. The default of <code>true</code> states that the domain object is an entity. Setting the <code>entity</code> tag to <code>false</code> marks a domain object as a value object and that it therefore does not have a database identity. Value objects can be used by other domain objects as simple attributes.</p> <p>Note: The relationship to value objects could also have been realized as a one-to-one aggregation. In this version of the eHF Generator we have chosen to take the attribute-based approach in order to stress the difference between value objects and entities with database identity. Please also note that the value object (or component) concept in</p>

	<p>Hibernate was not used due to a number of issues. However, a migration to the Hibernate functions should be possible without a major impact on the domain model.</p>
export	<p>The <code>export</code> tag regulates the visibility of domain object services on the adapter layer and can take one of the values:</p> <ul style="list-style-type: none"> • internal & external • internal • external • none (Default) <p>The default of <code>none</code> indicates that the given domain object does not export its service to either the internal or external service adapter layer. Setting <code>export</code> to <code>internal</code> or <code>external</code> forces the eHF generator to produce internal and external service adapters for the given domain object. Those adapters provide service interfaces containing all service operations modeled for the given domain object. If the domain object is marked as <code>crud</code>, the CRUD operations will be exported. Additionally, all modeled and exposed domain object operations are visible at the exported service adapter. So, this tag lets you control where service signatures should be visible. By default (<code>none</code>), no domain object service is exported, which means exporting is an explicit decision made by the module developer.</p>
expose	<p>This tag controls the visibility of modeled domain objects. The <code>expose</code> tag can take one of the values:</p> <ul style="list-style-type: none"> • internal & external • internal • external • none (Default) <p>The default of <code>none</code> indicates that the given domain object is neither visible as an internal nor external transfer object. Setting <code>expose</code> to <code>internal</code> respectively <code>external</code> forces the eHF generator to produce an internal and external representation for the given domain object respectively. This is especially useful if domain object model details should be hidden on the internal and external API. For a more fine-grained, attribute-related decision about the <code>expose</code> state, consult the <code>expose</code> tag in the ehf-attribute section.</p>
forceJoinedInheritance	<p>By default, this boolean flag is disabled and the generator decides on the ORM inheritance mapping strategy. By setting this flag to <code>true</code>, the joined inheritance strategy is explicitly taken</p>

	as the inheritance strategy - this means one table per class. This tag need only be set in the top level super class in the class hierarchy, to take effect.
pseudonymized	General encryption is enabled for a module by setting ehf.generator.encryption.enabled=true in the module.configuration.properties. Encryption can be used for many use cases of protecting data (data integrity, confidentiality and pseudonymization (de-identification)). The generator anticipates a pseudonymization use case as default and encrypts certain attributes (in particular the technical attributes) in a domain object for pseudonymization. While this is a good default for user related data it is not very applicable to shared data or non-user related objects. In order to consciously exclude types from this default behavior the pseudonymized tag was introduced with profile version 1.3 of the eHF Generator. The tag defaults to true and can be overwritten for domain objects, where pseudonymization concerns do not apply to.
secured	The secured boolean flag determines whether an object is security relevant or not. In the context of the eHF this determines if certain technical attributes relevant for security purposes will be generated for an object or not. These include fields for storing the scope, the creator or the last change date of an object. The default value for this flag is true. By setting it to false, the only technical attribute that will then be generated for the object is the ID field.
stub	<p>Marks an object as being a stub. Stub objects define dependencies to other modules. In the eHF context the ehf-core projects provide central data structures that can be used via this mechanism in the local model.</p> <p>Note: Currently there is a problem with stub objects that involve other stub objects (associations/attribute/class hierarchy) concerning the proper generation of web service bean mappings. The current workaround is to include all objects as stubs that are connected to the object of interest. In the future the generator will be able to cover this automatically.</p>

suppressEmf	Indicates that Eclipse Modeling Framework (EMF) artifacts should not be produced by the generator. The EMF classes are used for OCL constraint checks. If it is known that there will be no OCL checks performed at runtime for an ehf-domain-object, the suppressEmf tag can be set to true. The default is false.
-------------	---

Supported native UML class concepts

Concept Name	Application in eHF
isAbstract	This boolean marks the domain object as an abstract class. This means that this object must not be instantiated directly. If not set, false is taken as default.
ownedRule	The ownedRule association in UML defines a set of constraints for a given model element. In eHF, we use this concept to define additional constraints on domain objects. The Object Management Group (OMG) defines the Object Constraint Language (OCL), a language for expressing constraints, as part of UML. Each UML constraint provides a specification (in our case, the OCL expression) and an element-wide unique name. The specification expresses the object constraint that has to evaluate to true in order to be a valid instance. The optional name field is interpreted as an error code key in eHF. If a constraint is violated, it is reported to the user in form of a coded exception. If no error code key is given for an OCL constraint, the default key (VAL-000301 for general validation error) is returned. A code is only complete with a corresponding system ID. Each module has the general validation error system ID (OID = 2.16.840.1.113883.3.37.2.9.5.1) as default system ID. You could switch to your module-specific system ID which contains your module-specific error code keys by setting the property module.error.systemid in a module's module.configuration.properties file.

A.2 Stereotype ehf-attribute

Stereotype Description

The stereotype ehf-attribute is used to extend the standard UML Property. It is both applicable on domain object attributes and association ends. All attributes of an ehf-domainobject are implicitly of the stereotype ehf-attribute. As such, the stereotype application itself does not add information to the model. However, the tags associated with this stereotype may provide detailed information on the attribute level. The same set of ehf-attribute tags is available on association ends as well. UML associations are always between two UML types whereas each association end is

again a UML Property. Due to this fact, the stereotype `ehf-attribute` is applicable on association ends as well, to further define the characteristics of domain object interdependencies.

Tag Description

Tag	Description
classification	<p>The <code>classification</code> tag lets you classify attributes in terms of concepts and confidentiality. The classification is then interpreted by the generator to support application-level encryption of the attribute. The chosen classification is included in the encryption process as a parameter. For details consult the documentation on Application-Level Encryption and Pseudonymization.</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends because the classification is only relevant for primitive types (for example, String, Integer).</p>
conceptValidation	<p>Indicates in which way the coded attribute has to be validated. The <code>conceptValidation</code> tag is backed by an enumeration containing three possible values:</p> <ul style="list-style-type: none"> • <code>none</code> (no validation, this is the default) • <code>code</code> (is this code an element of the specified concept?) • <code>systemId</code> (is the systemId part of the code aligned with the specified concept?) <p>This tag is only applicable for coded attributes.</p> <hr/> <p>Note: This tag replaced the <code>codeValidation</code>, <code>codeSystemValidation</code> and <code>codeSystemCategoryValidation</code> tags.</p>
concept	<p>Tag for specifying the code concept for a coded property. If you model a domain object attribute of type <code>Code</code> or any subclass of it, you will need to assign an appropriate code concept in order to define the attribute's value range. The <code>concept</code> tag can have one of the following values:</p>

	<ul style="list-style-type: none"> • <code>CodeSystem</code> - defines a code system as value range for the given attribute. • <code>CodeSet</code> - defines a code set as value range for the given attribute. • <code>CodeCategory</code> - defines a code category as value range for the given attribute. <p>With this tag, you only decide what type of code concept you want to apply to a coded attribute. Through the tag <code>conceptDescriptor</code>, you have to specify the code system, code set or code category which defines the set of values for the given attribute.</p>
<code>conceptDescriptor</code>	<p>This tag references the code concept (code system, code set or code category) which defines the value range for the given attribute. Note, that the definition depends on the selected code concept of the tag <code>concept</code>. If <code>CodeSystem</code> was selected as concept, the concept descriptor has to reference an existing code system. The pattern for referencing a code system is the following:</p> <pre>{OID} {version}</pre> <p>Since the version is optional, you are allowed to provide the code system OID only:</p> <pre>{OID}</pre> <p>If <code>CodeSet</code> is the selected code concept, the concept descriptor has to reference an existing code set, which is identified by its name and optional version:</p> <pre>{code system name} {version}</pre> <p>Again, the version is optional, so you are allowed to provide the code set name only:</p> <pre>{code system name}</pre> <p>If the attribute is bound to a code category by setting <code>concept</code> to <code>CodeCategory</code>, the <code>conceptDescriptor</code> has to reference a code category by its name:</p> <pre>{code category name}</pre>
<code>contractRelevant</code>	<p>Attributes marked as contract relevant are attributes, which may not be modified by a user without special permissions. The check for a contract relevant attribute is a two level check. An attribute may only be contract relevant if the object itself is contract relevant. The default for this tag is <code>true</code>.</p>

	<p>Note: This tag is only evaluated on domain object attributes and not yet on association ends. To mark an associated property as contract relevant, you have to use the <code>contractRelevant</code> tag of stereotype <code>ehf-association</code>.</p>
custom	<p>Tag marking the <code>custom</code> attribute. Custom attributes are under the control of the developer. The generator will generate them into the main source tree and will not overwrite the class during later runs. Currently only <code>custom='domainobject'</code> is supported. Other levels may be added on-demand in the future.</p> <p>Note: This tag is only evaluated on domain object attributes and not on association ends because customization is controlled in the associated domain objects themselves.</p>
defaultValue	Tag for defining a default value.
directive	<p>The <code>directive</code> tag determines how the <code>classification</code> tag is handled for embedded structures. The classification can either be <code>none</code>, <code>overwrite</code>, <code>exclude</code> or <code>encrypt</code>. In case <code>none</code> is specified, the attribute is treated as though no classification exists. In particular when a class-level classification is set, the attribute can be excluded using the <code>none</code> directive. In case an attribute is tagged with <code>overwrite</code> the classification from the entity, which contains the embedded value object is propagated and overwrites the local classification tag. The <code>exclude</code> tag prohibits an overwrite from the entity. The classification of the attribute stays as annotated. The <code>encrypt</code> tag indicates that an attribute is required to be encrypted on database level. This feature is currently in evaluation and not yet supported.</p>

	<p>Note: This tag is only evaluated on domain object attributes and not on association ends because the directive is only relevant for primitive types (for example, String or Integer).</p>
expand	<p>This tag can be switched to <code>true</code> in order to specify that the attributes of an object should be directly accessible on the DTO and XTO layer via additional getter and setter methods. This tag is meant to provide some flexibility for the modeling while keeping the transfer object layer stable.</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
expose	<p>This tag controls the visibility of modeled domain object attributes. The <code>expose</code> tag can take one of these values as an attribute:</p> <ul style="list-style-type: none"> • internal & external (Default) • internal • external • none <p>The default of <code>internal & external</code> indicates that the given domain object attribute is both visible in the internal and external transfer object. Setting <code>expose</code> to <code>internal</code> respectively <code>external</code> forces the eHF generator to expose the given attribute inside the internal respectively external transfer object only. This is especially useful if an attribute should be hidden on the internal or external API. Setting <code>expose</code> to <code>none</code> hides the given attribute in both internal and external transfer objects.</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not yet on association ends. Support for association ends may be part of future releases.</p>

externalized	<p>This boolean flag describes whether the value control is externalized or not. By default, all attributes can be manipulated from external. If this attribute is set to <code>false</code> manipulation from external is prevented.</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
immutable	<p>Tag for providing meta information for the attribute. If an attribute is immutable its content should not be modified by a client. This meta information is currently not enforced by the generated code.</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
index	<p>For a persistent attribute marked as <code>index</code> the generator will ensure an index is generated in the database.</p> <hr/> <p>Note:</p> <ul style="list-style-type: none"> The setting might be in conflict with a unique constraint. For attributes that are supposed to be unique some databases like oracle create an index by default. In case an attribute is tagged as <code>unique</code> and as <code>index</code> the generator will not explicitly force an index here, since it would result in a error during deployment on an Oracle instance. In the future we might require the Hibernate Tools to cover all databases homogenously (preventing an extra index being generated for Oracle). If a more complex index spanning several attributes is required, you currently have to provide the index yourself. To be able to do so the whole

	<p>generated domain object could be moved into the <code>src/main/java</code> folder and could be enriched with custom extensions like additional indexes as required.</p> <ul style="list-style-type: none"> • This tag is only evaluated on domain object attributes and not on association ends.
<code>lob</code>	<p>This tag lets you specify an attribute as a 'large object'. This information is required on the database level on attributes that should be able to contain larger content. From the Hibernate documentation: <i>[...lob] indicates that the property should be persisted in a Blob or a Clob depending on the property type: <code>java.sql.Clob</code>, <code>Character[]</code>, <code>char[]</code> and <code>java.lang.String</code> will be persisted in a Clob. <code>java.sql.Blob</code>, <code>Byte[]</code>, <code>byte[]</code> and serializable type will be persisted in a Blob.</i></p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
<code>maxLength</code>	<p>Specifies the maximum length for a string attribute. The default was derived from the common database default value, which is 255. The max value should not exceed 1024 (because of database level constraints) without being marked as <code>lob</code>.</p>
<code>minLength</code>	<p>Specifies the minimum length for a string. The value '-1' is the default and means that no minimum length can be provided (empty string or null allowed).</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
<code>maxValue</code>	<p>Tag for providing the maximum value for an int. By default the value is 2147483647.</p>

	<p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
minValue	<p>Tag for providing the minimum value of an int. The default value is the lowest possible integer value '-2147483648'</p>
	<p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
nullable	<p>Tag identifying mandatory attributes. The default is <code>true</code>. Setting it to <code>false</code> marks the attribute as mandatory.</p>
	<p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
referenceId	<p>By setting this flag to <code>true</code>, this attribute is marked as a reference ID. All reference ID attributes throughout the model have the same length because they all store IDs following the same pattern. This length is set according to the generator configuration. The current default configuration defines a maximum length of 64 for all reference IDs. This default configuration can be overridden for any given module by defining the property <code>db.id.length</code> in the module's <code>module.configuration.properties</code> file.</p>
	<p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
roleDescriptor	<p>The role descriptor defines the role of an association. Two domain objects may stand in multiple relations where each of them has to be distinguished uniquely (for example, you can have multiple associations with your co-workers - one is the employment</p>

	<p>relationship while another may be a friendship). By distinguishing the roles, we are able to treat them independently. Access control rights can be expressed separately for each association. In eHF, we express roles in codes. So, the role descriptor has to represent a code. In eHF, we are able to through the association role. A code has to following the given pattern:</p> <pre>{codeSetName} {version} {key}</pre> <p>codeSetName and key are mandatory. version is optional and can be omitted. With this role descriptor, a role code can be uniquely identified.</p> <hr/> <p>Note: This tag is only evaluated on domain object association ends and not on attributes.</p>
transient	<p>By marking an attribute as transient the attribute (in the case of a persistent domain object) will not be stored in the database.</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>
unique	<p>Tag defining an attribute as unique. This tag is normally used only on the database level and normally imposes the constraint directly onto the table the object is stored in.</p> <hr/> <p>Note: This tag is only evaluated on domain object attributes and not on association ends.</p>

A.3 Stereotype ehf-association

Stereotype Description

All associations between ehf-domainobjects are implicitly of stereotype eHF Association. As such the stereotype doesn't add information to the model. However the tags associated with this stereotype provide additional details on an association.

There are certain Ad-hoc rules how the different association types are interpreted by the eHF Generator. The Generator differentiates the following association types:

Association	An association represents a structural relationship between the instances of two domain objects classes, which are on the same conceptual level. An association has no further characteristics implied. That means that the instances of the two classes in the relationships are not coupled in a specific way.
Aggregation	An aggregation is a special association. It is often referred to as part-of relationship (object B is part of object A). In this relationship the parts may be regarded as independent, but the instances may be assembled into an aggregate object.
Composition	A composition represents a special Aggregation with strong ownership and coincident lifetime of the party by the whole. This means that the parts of the relationship have the same lifetime as the whole composite object. In particular the party may not exist without the relationship. The object with all composition defines a composite object.

The following table summarizes the characteristics of the relationships as the generator generates them.

	Association	Aggregation	Composition
Persisting	When storing an object graph that has not persisted objects persistence is cascaded.	When storing an object graph that has not persisted objects persistence is cascaded.	When storing an object graph that has not persisted objects persistence is cascaded.
Removing	No cascaded remove.	No cascaded remove.	Cascading remove. The parts of the composition are deleted with the object.
Loading	Lazy (default).	Eager if not marked as dependency, lazy in the dependency case (default).	Eager if not marked as dependency, lazy in the dependency case (default).

Finally, the following table summarizes the possible tags on an `ehf-association` stereotyped association.

Tag Description

Tag	Description
<code>classification</code>	The tag is interpreted when encryption is activated for the module. It only has an effect on one-to-many entity-to-value-object

	compositions. The classification defines the encryption context for the composition. The classification is propagated according to the directives on the value objects attributes.
codeValidation	Indicates whether the full code has to be validated against the code repository. Note: This tag will be revised in the course of future code set and code system consolidation activities.
codeSystemValidation	Indicates whether the codeSystem has to be validated against the code repository. Note: This tag will be revised in the course of future code set and code system consolidation activities.
contractRelevant	Associations marked as contract relevant imply that a user is not permitted to modify the associated instance's content without special permissions defined in by some contract.
dependency	Marks an association as dependency. This is relevant on the transformation from domain object to DTOs and XTOs. Dependencies may be included on demand.
fetch	Lets you overwrite the default fetch type, which is dependent on the association type (see table). Possible values: 'eager', 'lazy'.
forceJoinTable	Lets you control the built-in strategy to map associations to the database. If left blank, the generator applied the most suitable strategy to represent an association. By setting this tag to true the generator will be forced to realize the association by using of a join table even if the built-in implementation would decide to have a mapping without a join table (for example, if it is a simple 1:1 association).
immutable	Tag for providing meta information for the association. If an associated object is immutable its content should not be modified by a client. This meta information is currently not enforced by the generated code.

link	Alternative way to realize a relationship in the database. When using this tag you have to provide an implementation to link and unlink the associated domain objects. The link is realized via an explicit mechanism that is defined via the modules ModuleLinkDaolmpl. The skeleton for this implementation is generated and must be implemented in detail.
ordered	Using the ordered tag you can mark a one-to-many association as ordered, which means that the database will store an index for the elements and the signatures accessing this association will use <code>java.util.List</code> instead of a <code>java.util.Set</code> .

A.4 Stereotype ehf-operation

Stereotype Description

When defining an operation inside a ehf-domain-object or ehf-service stereotype the operation will not be generated on the domain object itself (since this is way to deep in the architectural layers), but on the DAO, service, and service adapter layer.

Tag Description

Tag	Description
expose	This tag controls the visibility of modeled operation. The expose tag could take one of the values: <ul style="list-style-type: none"> • internal & external • internal • external • none (Default) The default of none indicates that the given operation is neither visible as internal nor external service adapter operation. Setting expose to internal respectively external forces the eHF generator to expose the signature of the given operation at the internal respectively external service adapter. This is especially useful for hiding operations needed inside a module only.
includeDependencies	Boolean tag which defines whether dependencies are included when transforming from internal to exposed DTO representation. By default, this value is set to false.
layer	With this tag the location of a method inside the generated application stack can be determined. The following values are supported: <ul style="list-style-type: none"> • service & dao (Default) • service

	<ul style="list-style-type: none"> • dao • none <p>The default states, that an operation is generated in both service and DAO layer. That means, that the service layer is generated which delegates its calls to the DAO layer, where the implementation for the operation has to be provided.</p> <hr/> <p>Note: The tag values <code>service</code> and <code>dao</code> follow a hierarchy in the order they are stated in this sentence. The operation implementation has to be provided in the lowest defined layer. The default envisions to implement the business logic at DAO level. Omitting the <code>dao</code> value results in a customizable service implementation where the business logic has to be placed.</p>
--	---

Supported native UML class concepts

Concept Name	Application in eHF
<code>upper</code>	<p>This integer defines the upper multiplicity of an operation's type. An operation's type in UML is its return value. The default is <code>1</code> which indicates that it is a single-valued. By setting it to <code>-1</code> (or <code>*</code> in UML notation, the return type is interpreted as a collection. Per default, the eHF generator produces a <code>java.util.Set</code> as result collection. This collection type could be controlled by using the <code>isOrdered</code> concept of the UML, which is as well described in this table.</p> <hr/> <p>Note: The described support of the <code>upper</code> UML concept is as well valid for all parameters of a modeled ehf-operation.</p>
<code>isOrdered</code>	<p>This boolean indicated whether an order is relevant for a given parameter. Setting it to true in conjunction with setting <code>upper</code> to <code>*</code> (see the description of the <code>upper</code> concept) forces the eHF generator to produce a return type of <code>java.util.List</code>. If the</p>

	<p>isOrdered attribute is not set, false is taken as default.</p> <hr/> <p>Note: The described support of the isOrdered UML concept is as well valid for all parameters of a modeled ehf-operation.</p>
--	--

A.5 Stereotype ehf-service

Stereotype Description

From version eHF 2.2 on the generator supports custom services. In the past services were possible only on the module level (module service; coarse grained) or on the domain object level (object service; fine-grained). With this stereotype the user of the eHF generator can define his granularity level of services (domain services). The eHF generator produces appropriate interfaces, classes and spring configuration. The customization and implementation of the service is however in the responsibility of the user.

Tag Description

Tag	Description
abstract	Tag indicating whether the service is abstract. This means it is supposed as base infrastructure for other services and will never be exported and thus will never be accessible for module consumers.
crud	This flag indicates whether the service provides CRUD service methods or not. By setting crud to true, the service will contain methods for creating, reading, updating and deleting objects of the domain model.
export	The export tag regulates the visibility of services on the adapter layer and could take one of the values: <ul style="list-style-type: none"> • internal & external • internal • external • none (Default) The default of none indicates that the given service is not exported neither on the internal nor the external service adapter layer. Setting export to internal respectively external forces the eHF generator to produce internal respectively external service adapters for the given service. Doing so, all modeled service

	operations (marked with <code>expose = internal</code> respectively <code>expose = external</code>) will be exported at the internal respectively external adapter interface. So, this tag allows to control where service signatures should be visible. Per default (<code>none</code>), no service is exported, which makes exporting an explicit decision for the module developer.
--	---

Supported native UML class concepts

Concept Name	Application in eHF
<code>isAbstract</code>	This boolean marks the service as abstract. This means that this service can not be instantiated directly. Abstract services may provide service functionality which can be reused by other services through inheritance. If not set, <code>false</code> is taken as default.

B eHF Profile Versions

In order to be able to both drive the evolution of our model-driven approach and remaining backwards compatible, all eHF profiles carry a version number. Each eHF module can individually use one of the supported eHF profiles in the model. The eHF Generator thereby always supports all released profile versions by internally translating between versions. It is recommended to always use the latest version of the eHF profile in order to benefit from the latest improvements and to use the latest concepts. This section describes the eHF profile evolution path. It informs you about differences between versions and could be seen as a guideline for upgrading eHF profiles to higher versions.

The eHF Generator project contains the latest profile plus all supported legacy profiles. You can find them under `src/main/model/profile` inside the eHF Generator project. Outdated legacy profiles are always stored in subfolders. Please note that the eHF profile comes in different formats:

1. `eHF_Profile.profile.uml` - the eHF profile as an EMF UML2 file. This is a plain EMF model file containing the eHF profile, which you have to use when your working model is a plain EMF file (for example, when using TopCased).
2. `eHF_Profile.xml` - the eHF profile in MagicDraw XMI. This file contains the eHF profile serialized in MagicDraw's XMI dialect. So, if you use MagicDraw for modeling, please use this profile inside your model.

B.1 Updating the eHF Profile

The following steps are necessary to update your model to a new version of the eHF profile. The tasks are described independently of the modeling tool as far as possible.

1. Find out the current version of your eHF profile. Each eHF profile carries its version number in form of an UML constraint.
2. Check for the next higher eHF profile version to switch to.

3. If there is a newer eHF profile version available, get its profile file and replace your old eHF profile file with the newer one (to be on the safe side concerning profile incompatibilities, make a backup of your current model and the corresponding profile).
4. Inform yourself about the new concepts and constructs of the new profile (see the next topics within this section for detailed profile change information). See the implications part for each listed profile change. Depending on the expressed implications, you have to adapt your model in order to preserve its correct semantic.
5. Redo the previous steps if you want to upgrade to the next higher profile version as well.



Note: Be aware that the current procedure has to be performed manually. In future, we want to provide transformations or scripts which automatically transform models from one version to a higher one.

B.2 eHF Profile Version 1.0

This initial version provides all the concepts and constructs known since eHF 2.7. Since there is no predecessor profile, there is no need to describe a migration path towards this 1.0 profile.

B.3 eHF Profile Version 1.1

This version of the eHF profile is available with eHF 2.8 (end of sprint 21) and contains the following improvements:

- Support for coded OCL constraints.

In previous versions, the OCL constraints could only be expressed inside the constraint tag of the `ehf-domainobject` stereotype. Multiple constraints had to be concatenated through `&&` expressions. This approach was error-prone and very inconvenient since returned OCL constraint violation exceptions only carried the violated OCL expression inside. With eHF 1.1, the `constraint` tag was removed from the `ehf-domainobject` stereotype. Since UML already provides the concept of constraints, each OCL constraint now has to be modeled as UML constraint. Optionally, the constraint name can be used to define the constraint's error code key. If a OCL constraint is violated, its corresponding error code key is used to return a coded information back to the user. If no code key is specified, the default validation error (VAL-000301) is used as the default.

Implications:

When migrating to eHF profile version 1.1, all your modeled OCL constraints need to be split into individual constraints and modeled as UML constraints. If you not do so, all modeled constraints will be lost since the tag `constraints` is no longer supported in profile version 1.1.

Additionally, you could provide error code key information for each extracted constraint.

B.4 eHF Profile Version 1.2

This version of the eHF profile is available with eHF 2.8 (sprint 22) and contains the following improvements:

- First production implementation of the `ehf-service` stereotype.

Previous eHF profiles already contained the `ehf-service` stereotype but only on an experimental basis. With version 1.2 we provide an implementation that can be used inside your module. So, instead of modeling operations on your domain object, you are able to define domain-object-independent services in UML.

- Consolidation of `internal`, `external` and `expose` tags.

Those three tags regulated the visibility of domain objects. We now are able to use one tag (`expose`) which defines the necessary enumeration literals we need (`internal & external`, `internal`, `external` and `none`). With it, we are able to control which objects we want to have transfer objects generated for. The same enumeration is reused in the newly introduced tag `export`. With it, you are able to control the visibility of domain object services on the respective adapters. By introducing the `export` tag, we clearly separate the concerns of exposing the data model and exporting services.



Note: Beside domain objects, the stereotypes `ehf-service` and `ehf-attribute` make use of the tags `expose` and `export` as well. In both cases, the semantic is preserved and applicable in the same way.

-
- Revision of the `ehf-operationlayer` tag.

The `layer` tag previously was a comma-separated string only. With profile 1.2, it is replaced by an enumeration type (with literals `service` & `dao`, `service`, `dao` and `none`).

- Introduction of tags for pseudonymization support

Profile 1.2 provides new tags named `classification` and `directive` to control the pseudonymization and encryption behaviour of domain objects and their attributes.

Upgrade to eHF profile 1.2

1. Revise your modeled eHF domain objects: the usage of the tags `internal`, `external` and `expose` need to be transferred into the enumeration-based tags `expose` and `export`. Be aware, that the previous tags summarized both concerns of exposing the data model and exporting services. While upgrading your domain objects you have to decide on a case by case basis whether to use an `expose` or an `export` value for each particular domain object.
2. Revise your modeled eHF attributes: The tags `internal`, `external` and `expose` need to be resolved in one enumeration literal of the new `expose` tag. To not pollute your model be aware of the default (`internal & external`) of the `ehf-attribute`'s `expose` tag.
3. Revise your modeled eHF operations: The tag `expose` need to be resolved in one enumeration literal of the new `expose` tag. Be aware that you now have the new possibility to make a decision even between exposing internally and externally. Also be aware of the changed default: in previous profiles, all operations were exposed by default. With the new profile, we follow a more restrictive way of modeling: Only explicitly marked operations get exposed.

B.5 eHF Profile Version 1.3

This version of the eHF profile is available with eHF 2.9 and contains the following improvements:

- Removal of the `abstract` tag in the stereotypes `ehf-domainobject` and `ehf-service`. Instead of using proprietary tags, we can use UML's built-in mechanism to set a UML class and interface respectively to abstract.
- Removal of the `multiple` tag in `ehf-operation`. This proprietary eHF tag was replaced by UML's upper concept. With it, you can specify an upper boundary for any parameter and return parameter respectively.
- Renamed the tag `embedded` to `entity` for the stereotype `ehf-domainobject`. In future, we want to use the terminology of entities and value objects and deprecate the usage of the term embedded type because it often implies wrapping of types which does not necessarily have to be the case.
- Refactoring of the code concept assignment tags. The tags `codeSystem`, `codeCategory`, `codeVersion` and `codeSubsystem` for stereotype `ehf-attribute` are replaced with tags named `concept` and `conceptDescriptor`.
- Refactoring of the code concept validation assignment tags. The tags `codeValidation`, `codeSystemValidation`, and `codeSystemCategoryValidation` for stereotype `ehf-attribute` are replaced with tag named `conceptValidation`.

Upgrade to eHF profile 1.3

1. Find all usages of the `abstract` tag in your model. Domain objects or services marked as abstract need to be migrated by setting the UML property `isAbstract` to true.
2. Find all usages of the tag `multiple` on `ehf-operation` elements. Each operation which is marked with `multiple = true` needs to be migrated. The upper multiplicity of the UML return parameter has to be set to * and -1 respectively.
3. Find all usages of `ehf-domainobjects` which have the tag `embedded` set to true. For the new profile, this corresponds to setting `entity` to false. Note, that not only the name of the tag changed - the default value toggled from false to true, too.
4. Find all usages of the tag `codeSystem` for `ehf-attribute` elements. If the value defines a code system OID, you have to select the `CodeSystem` as the value for tag `concept`. If the value is a code set name, select `CodeSet` as the `concept` tag value. In a second step, you have to set the OID and code set name respectively as the value of the new tag `conceptDescriptor`. If the current attribute has the tag `codeVersion` set to something other than 1.0.0, you have to append the code version information to the `codeConcept` value as well. Note, that OID respectively code set name and version are separated by |.
5. Find all usages of the tag `codeCategory` for `ehf-attribute` elements. For all found attributes, set the `concept` tag to `CodeCategory` and transfer the category name into a new `conceptDescriptor` tag.
6. Find all usages of the tag `codeSubsystem` for `ehf-attribute` elements. This tag is no longer supported in eHF profile 1.3. So, you either have to provide a code system, code set or code category as the value range for the given attribute.
7. Find all usages of the tags `codeValidation`, `codeSystemValidation`, and `codeSystemCategoryValidation` for `ehf-attribute` elements. These tags are no longer supported in eHF profile 1.3. Instead you have to set the `codeValidation` appropriately. If `codeValidation` had been set use the value `code` instead. If `codeSystemValidation` or `codeSystemCategoryValidation` had been set, then use the value `systemId`. Otherwise none.

C Versioned Template Reference

In order to isolate the evolution of the generator from existing generated modules the concept of versioned templates was introduced. Using versioned templates the generator enables a user of the generator to migrate her/his project using the documented migration steps as required. In this section the version templates are listed and the migration process is described.

C.1 DAO Template 2.0

The DAO Template version 2.0 was the first versioned template introduced. The initial approach to the DAO implementation level was too restrictive and caused major inflexibility. The DAO template version 2.0 resolves this issue and provides a more natural approach.

Migration from the standard version to 2.0 is very easy:

1. Simply add the following line to your `module.configuration.properties` file:
`ehf.generator.dao.version=2.0`
2. Run the build. The generator will fail if you are already using the standard DAOs. If you do not have custom operations the build will terminate successfully and no further action is required (Step 3 can be skipped).
3. Move the code in the `....dao.hibernate` package from the original `...DaoDelegate.java` to the `...Dao.java` classes. This can be done by copying and pasting the methods from the delegate to the DAO. The pasted code has to be adapted as follows:
 - a. In the DAO signature implementations the first parameter (`...Dao dao,...`) must be removed.
 - b. Everywhere the DAO was used in the code (for example, `dao.getHibernateTemplate()`) the 'dao.' must be removed.
4. Check that all build errors are resolved (tests may still include references to the delegates).
5. Rerun the build to check that everything is fine.

C.2 Web Service Template 2.0

The standard web service generation was quite verbose in terms of Java classes and the custom code a developer had to add. It consisted of a `<name>ModuleServiceXtoAdapter` interface and an implementation of `<name>ModuleServiceXtoAdapterImpl` respectively. On top of this, there are one more interface `<name>ModuleWebServiceInterface` and two more implementation classes: `<name>ModuleWebServiceImpl` and `<name>ModuleWebServiceDelegate`.

All the individual classes had their rational and in particular separated concerns.

However, the usability was not satisfactory. In order to expose a module service method you had to make additions to all five classes to expose this method on the web service interface. In addition to that you had to extend the web service configuration file to include this method.

The new web service templates reduce this issue and make the exposure of a method much more natural. The general idea is to derive the web service signatures directly from a Java interface. In order to reduce the number of required classes and locations for adding custom code the number of classes is reduced. No additional configuration is needed.

Ultimately the <name>ModuleServiceXtoAdapter interface was identified as the interface to determine the web service signatures. The interface and classes <name>ModuleWebServiceInterface, <name>ModuleWebService-Impl and <name>ModuleWebServiceDelegate are not required anymore.

This means that exposing a module service method now simply means adapting the XtoAdapter interface and implementation.

The responsibility of the obsolete three classes was moved to a special Axis handler.

ATTENTION: the new template will change the WSDL, which means that the clients for your module have to be regenerated and adapted. This may have an impact in QA and the SDK.

Here the steps to migrate your web service:

1. Add the following line to your `module.configuration.properties` file:
`ehf.generator.webapi.version=2.0`
2. Move the custom signatures from the <name>ModuleWebService-Interface to the <name>ModuleServiceXtoAdapter and make sure they are implemented by the <name>ModuleServiceXtoAdapterImpl.
3. Remove the <name>ModuleWebServiceInterface, <name>ModuleWeb-Servicelmpl and <name>ModuleWebServiceDelegate.
4. Ensure the XtoAdapter interface extends ModuleServiceXtoAdapter instead of ModuleServiceAdapter. And the implementation extends AbstractModuleServiceXtoAdapter instead of AbstractModuleService-Adapter
5. Check all build errors are resolved (tests may still include references to the delegates).
6. Check your `method.wsdd.fragment`. Normally you can keep it as it is. In order to gain more freedom for later additions you may want to a '*' as wildcard for any method.
7. Run the build. The generator generates a new WSDD file with additional metadata for the handler (interface, implementation, beanId). The build is not supposed to fail.
8. Run an assembly to check your web service looks and works as anticipated.

NOTE: The content of the XtoAdapterInterface is completely under your control. When the generator generates them they inherit standard signatures from a standard interface. If you don't want this, simply remove the extends ModuleServiceXtoAdapter.

NOTE: The limitation that methods can not be overloaded remains. When you expose overloaded methods your web service will not be functional.

C.3 Web Service Template 2.1

The web service template 2.1 has some minor changes that mainly focus on cleaning up the WSDD and not exposing internal concerns. Especially the Xto extension, which was exposed to the outside is removed from the object's namespaces. This makes the API much cleaner.

The migration steps from 2.0 to 2.1 consist only of:

1. Adding the following line to your `module.configuration.properties` file:
`ehf.generator.webapi.version=2.1`
2. Running the build. The generator creates a modified WSDD file.

3. Please note that in terms of backward compatibility a migration has to be considered with care. Changing the web service of a module, which is already exposed in the field violates backward compatibility and has to be resolved by expensive transformations. The BAS team is currently evaluating a strategy to provide a simple migration path for these cases. (that is, exposing two services; one in 2.0 and one in 2.1 fashion. However, the WSDL for the 2.0 fashion is to be hidden).

C.4 Auditable Service Template 2.0

One general design fault was using Hibernate interceptors to implement technical and business concerns. One of them also exposes the most prominent naming ambiguity of the system. The AuditInterceptor manages creator and changer details on domain objects. It is controlled via the @Auditable annotation.

In the future we want to replace the Hibernate interceptors by other means of our architecture or with Hibernate listeners. In eHF we propose the first option, since it allows for more control.

The new template does not create any @Auditable annotations anymore. Moreover, the supporting Spring configuration is generated for new modules. Existing eHF modules have to register one create and one update listener in the Spring custom context in order to replace the @Auditable functionality:

- com.icw.ehf.commons.principal.listener.PrincipalCreateObjectEventListener
- com.icw.ehf.commons.principal.listener.PrincipalUpdateObjectEventListener

C.5 Persistence Aggregate Template 2.0

As part of the cascading update implementation it was required to reorganize the location of foreign keys on the border from composition to aggregations / associations. When switching to the latest template the generator ensure the foreign keys are never stored with the shared objects.

The following steps are required to move to the 2.0 version:

1. Add the following line to your `module.configuration.properties` file:
`ehf.generator.persistence.aggregate.version=2.0`
2. Run the build. The generator creates new annotations on domain object level
3. In case the build procedure reports wrong indices please revise your custom domain object classes and adjust the index to map the new Join Table name.
4. Provide upgrade steps in case the switch has had an effect on the schema.

C.6 Security Annotations Template 2.7

Permission checks compose a classical cross-cutting concern, which was widely spread throughout the whole architecture. The authorization was enforced through AspectJ, Spring AOP and Meta Data Enhancement in an invasive and not an isolated way. With the new versioned template, the Security Annotations Framework is used to clearly isolate business from authorization concerns. The Security Annotations Framework provides an annotation-driven method to enforce permission checks. The new version produces artifacts carrying the necessary security annotations. As well, Spring beans are generated into the Spring context to evaluate the annotations. Having this in place makes the always generated security aspect weaved on domain objects obsolete.

The following steps are required to move to the 2.7 version:

1. Add the following line to your `module.configuration.properties` file:
`ehf.generator.security.annotations.version=2.7`
2. If not yet there, add a dependency to the Security Annotations Framework in your `project.xml`:

```
<dependency>
    <groupId>safr</groupId>
    <artifactId>safr-core</artifactId>
    <version>0.8.2</version>
    <properties>
        <aspectj.weaveWith>true</aspectj.weaveWith>
    </properties>
</dependency>
```

The `<aspectj.weaveWith>` property is needed to weave the Security Annotation Framework aspect into all annotated classes.

3. The new versioned template puts `@SecureObject` annotations on all generated domain object classes. In case you already have customized domain objects in the domain package inside the `src/main/java` folder, make sure that all customized domain objects carry the `@SecureObject` annotation in order to be detected by the Security Annotations Framework.
4. The class `com.icw.ehf.authorization.aop.DomainServiceSecurity` is deprecated and replaced by the security annotations. So, classes extending this deprecated class without any implementation can be removed from the `security` package inside `src/main/java` if they exist.
5. Non-empty implementations which extend
`com.icw.ehf.authorization.aop.DomainServiceSecurity` typically implement custom security checks which are not covered by standard eHF. eHF provides several mechanisms to hook in your custom logic (for example, additional checks) before and after service invocations. If your service security implementation only makes additional READ checks on finder methods, those implementations could be replaced by their SAFR counterparts. To do so, first delete those service security implementations and move the domain object service implementation they previously protected to `src/main/java`. At the finder service method you want to protect, just add SAFR's `@Filter` annotation to make SAFR filter the result of those methods. For additional ways of intercepting services, please have a look at `com.icw.ehf.commons.aop.ServiceStrategy` and `com.icw.ehf.commons.service.interceptor.ServiceInterceptor` for more information about how to introduce your custom logic around services.
6. The bean names for service implementation changed from `<module name><service name>SecureService` to `<module name><service name>ServiceImpl`. So, if you reference those beans directly (for example, in unit tests), please adjust the bean names accordingly.
7. Run the build. The generator creates new `@SecureObject` annotations on domain objects (see domain package in `src/main/gen`) and the service security artefacts under the `security` package are omitted. The generated Spring context in this versioned template comes without any Spring proxies around the service beans.

C.7 Input Validation Template 2.9

With version 2.9, eHF provides support for input validation on all service methods. The new feature is isolated in a separate versioned template so that it can be enabled in each module individually. Once this versioned template is set to version 2.9 all service method input is validated implicitly. Currently only the String attributes of domain objects are validated according to their length meta data (other meta data aspects such as mandatory

are not checked during input validation) and individual Strings (not part of a domain object) and those that are part of either a String array or a String parameterized Collection (i.e. List<String>) with a length greater than 255 are rejected.

The following steps are required to move to the 2.9 version:

1. Add/update the following line of your module.configuration.properties file:
ehf.generator.input.validation.version=2.9
2. Run the build again.
3. Unit tests which violate the input validation constraints will fail. Unit tests which expect a meta data validation exception will now get input validation exceptions (com.icw.ehf.commons.exception.validation.ValidationException). Each conflict has to be solved individually either through adaptation of the unit tests or through relaxing the validation rules for service methods. Out-of-the box, the generator only produces the default input validation, which could be customized on method and even parameter level. Keep in mind, that the input validation provides an additional level of security. Conflicts have to be solved with respect to balancing both API usability and security concerns.
4. If you have an external test client (for example, a web service test client) you will probably have to adjust your expectations. The service methods are more restrictive with input they accept.

C.8 Persistence Association Template 2.10

Under certain modeled associations the generator placed the Hibernate annotations on classes on the wrong side of the association. This then resulted in incorrect SQL code (self referencing foreign keys). The effected associations were bi-directional aggregations, and any many-to-many associations. Examples of these are shown in the following image:

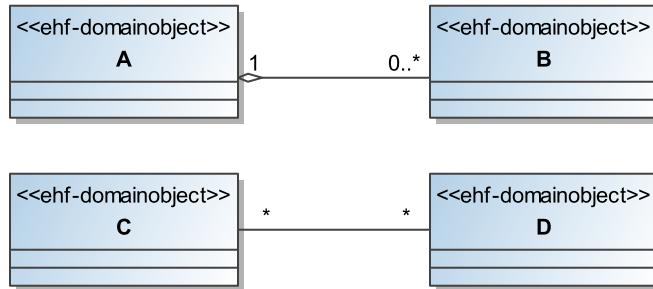


Figure 138: Association Issues

By fixing BAS-1838, the correct Hibernate annotations are now placed on the correct sides of the associations. Additionally some general foreign key and index annotations where cleaned up.



CAUTION: When switching to this template, even if you don't have the described scenarios modeled, you may still find there are some small database schema changes that result due to some optimization of the foreign keys and indexes that has also been carried out as part of the work done for this template.

The following steps are required to move to the 2.10 version:



Note: If your module is not yet in production, there is no production database schema available which needs to be migrated. So, you only have to perform step 2 below to move to version 2.10 of this template.

1. Build your module and store the generated `schema-create.sql` files which are available in `src/main/gen/META-INF/db`. These represent your old schema.
2. Add the following line to your `module.configuration.properties` file:
`ehf.generator.persistence.association.version=2.10`
3. Run the build again.
4. Now compare the newly generated `schema-create.sql` file with the stored one. The differences represent the changes in your underlying database schema for which you have to provide appropriate upgrade scripts. We hereby cannot provide a boilerplate solution because the upgrade scripts are highly dependent on the modeled domain objects.

C.9 Persistence Inheritance Annotations Template 2.9

The eHF Generator has a built-in decision implementation to decide how generalizations between domain objects in the model are mapped to the underlying database. If your domain object has domain object relationships which follow the pattern shown in [Figure 139](#), the wrong inheritance strategy was chosen.

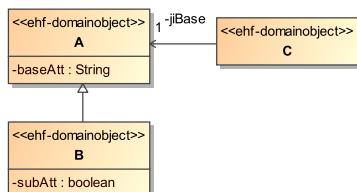


Figure 139: Inheritance Pattern

The old version uses the single table inheritance strategy whereas the joined inheritance strategy is more suitable here. By fixing BAS-932 and BAS-1233, the correct inheritance strategy is chosen for the above mentioned scenario. So, the eHF Generator now produces a deviant database schema if your domain model has such a described domain object combination modeled. So, if your module is already deployed in a previous version, you have to provide appropriate database upgrade scripts in order to be fully backward compatible.

The following steps are required to move to the 2.9 version:



Note: If your module is not yet in production, there is no production database schema available which needs to be migrated. So, you only have to perform step 2 below to move to version 2.9 of this template.

1. Build your module and store the generated `schema-create.sql` files which are available in `src/main/gen/META-INF/db`. These represent your old schema.
2. Add the following line to your `module.configuration.properties` file:
`ehf.generator.persistence.inheritance.annotations.version=2.9`
3. Run the build again.
4. Now compare the newly generated `schema-create.sql` file with the stored one. The differences represent the changes in your underlying database schema for which you have to provide appropriate upgrade scripts. We hereby cannot provide a boilerplate solution because the upgrade scripts are highly dependent on the modeled domain objects. Note that not only columns move to other tables but with them, data moves as well. Beside scripts to update the columns you as well have to provide scripts to copy already existing attribute data to their new location.

C.10 Persistence Join Table Template 2.9

The original template creates ambiguous Join Table names that can cause conflicts in case multiple associations are modeled from a class.

The new template version creates unambiguous Join Table names and therefore prevents a conflict.

The following steps are required to migrate from version 1.0 to 2.9:

1. Add the following line to your `module.configuration.properties` file:
`ehf.generator.persistence.jointable.version=2.9`
2. Run the build. The generator creates new annotations on domain object level.
3. In case the build procedure reports wrong indices please revise your custom domain object classes and adjust the index to map the new Join Table name.
4. Provide upgrade steps in case of changes. Please note that the template switch may not have an effect on the schema level at all, in case your model does not define a case, where the template is applicable.

C.11 Persistence Holder Template 1.1

Starting with version 2.9 eHF has enhanced the support for modeling one-to-many entity-to-value-object compositions. The holder template in version 1.1 overcomes certain hibernate limitation that have been monitored in the past. The template introduces additional `not-null` constraints on the value object columns in the database.

The following steps are required to move to the 1.1 version:

1. Add the following line to your `module.configuration.properties` file:
`ehf.generator.persistence.holder.version=1.1`
2. Run the build again. Currently there are no known issues. Your module should build as usual without any impact.

3. However, upgrade steps are required to add the `not-null` constraints to an existing schema. The existing content needs to be upgraded by replacing `null` entries with a blank (' '). The implementation will handle the blank properly. The schema creates scripts that are created by the Hibernate tools and provide the SQL statements that need to be copied into the upgrade scripts.

D OCL Reference

E Technologies

This chapter describes some of the most important technologies used by the eHealth Framework. They are crucial in the implementation of the concepts and the architecture presented in the previous chapters. As a developer, you will have to deal with these core technologies in various places.

E.1 Spring Framework

The Spring Framework (or Spring for short; see also <http://www.springframework.org/> ↗) is an open source application framework for the Java Platform, created to simplify the development of complex enterprise applications.

The complexity of developing Java applications lies in the multitude of Java APIs and their different underlying concepts. Spring solves this problem by providing a simplified and unified API for many Java APIs and other popular open source frameworks. It also provides a solution for the management of dependencies between objects. Furthermore, Spring supports aspect oriented programming in order to centralize cross-cutting concerns such as transactions, tracing or security (see [Aspect-Oriented Programming](#) on page 8). In addition, Spring emphasizes a POJO-based programming model. For more information about POJOs please refer to [Plain Old Java Objects](#) on page 8.

Spring plays a central role in the infrastructure of eHF Modules. Deploying eHF Modules is possible in any environment (application server, web container, standalone) but Spring provides easy access to the necessary hooks and benefits of the runtime environment (e.g. JMS, transaction management, JNDI defined objects).

Another advantage of using Spring is that it is largely non-invasive. Spring itself consists of several smaller frameworks for particular domains such as AOP or Web. Using the components of Spring independently is possible. Thus, you have the choice to use only those parts of the Spring Framework you really need. The eHF employs several Spring Frameworks: Spring Core, Spring AOP, Spring ORM, Spring DAO and Web Services. The eHF currently makes use of the Spring features described in the following sections. Most of the classes and configuration files are generated by the eHF Generator (see [Generator](#) on page 88). It is able to customize them through eHF's injection, the extension techniques.

Inversion of Control Container

Spring provides a lightweight Inversion of Control (IoC) (see [Inversion of Control](#) on page 9) container to manage the dependencies between Java objects, which enables a loosely coupled design. It is used for configuration of application components and life cycle management of the Java objects. The container uses Dependency Injection to establish the dependent relationships of the objects. A component simply uses other dependent components instead of instantiating or obtaining them actively by itself. This makes it possible to construct complex dependency structures in a consistent way.

The Spring BeanFactory interface is an implementation of the Factory Pattern and the root interface for accessing Spring's bean container. The eHF uses the declarative way and not the annotations driven way for the Spring context. Thus, this context is managed in XML files. The eHF uses the Spring interface ApplicationContext as entrance to the Spring Framework for this. This is a sub-interface of the BeanFactory. The ApplicationContext also provides the supporting infrastructure to enable numerous enterprise-specific features such as transactions and AOP. The eHF uses the concrete Spring implementation ClassPathXmlApplicationContext of this interface. It is a standalone XML application context taking the context definition files from the class path.

Spring supports hierarchies of contexts. The eHF extends this context with the class ModuleContext, a special implementation of the ClassPathXmlApplicationContext, found in the commons module (see [Commons](#) on page). For the particular behavior of the ModuleContext and its public API please refer to the section [Public API in Spring](#) on page 335. Spring also supports the enrichment of XML configurations with custom namespaces. The eHF provides an own namespace for declaring the module context contract.

Spring AOP

Using the IoC container, so the BeanFactory and the ApplicationContext, without the Spring AOP Framework is certainly possible. But the Spring container is greatly enhanced when AOP is added. You are also able to apply additionally AOP techniques such as AspectJ. For further information about AOP please refer to [Aspect-Oriented Programming](#) on page 8.

Spring AOP is used to apply aspects to Spring-managed components. The eHF uses AOP techniques to enable clear encapsulation of cross-cutting concerns such as permission checks, transaction management, error handling or monitoring. Spring AOP is leveraged by the eHF to apply transaction management, access control, object filtering ([Authorization](#) on page 162) .

Hibernate DAO Support

Spring provides DAO support for several Object/Relational (O/R) Mappers, including Hibernate (see section [Hibernate](#) on page 340) which is used by the eHealth Framework. Spring provides generic support for different O/R Mapper implementations which are extensible. The Spring DAO support provides a set of abstract DAO classes with methods for providing the data source and also for providing any other Hibernate-specific configuration settings. Extending these abstract DAO classes is possible. Thus, Spring enables an easy handling of O/R Mappers such as Hibernate in a consistent way. The eHF uses the `HibernateDaoSupport` as superclass for all its Hibernate data access objects for this.

Spring's `HibernateDaoSupport` requires a `Hibernate SessionFactory` for the resource management to be set. Spring allows this resource to be defined as a bean in the Spring container. Thus, the eHF does not need to implement hard-coded resource lookups.

Transaction Management

Spring provides a consistent abstraction for management of database transactions considered as a logical bracket around several actions such as method calls. The advantages of using transaction management with Spring are, amongst others, a simple API for programmatic transaction handling and the possibility of declarative transactions. Thus, it is a non-invasive and transparent way of transaction management.

The eHF mainly uses the declarative transaction management of Spring. But for import processes, when large amounts of data are transferred into the database, also the programmatic transaction management is used by the eHF. The implementation is provided in the eHF commons module. The interface `ImportProcessor` defines methods for importing data of any kind into the database. The abstract class

`AbstractImportProcessor` implements this interface. Each module which wants to use this import process transaction has to extend this abstract class. Within these implemented classes for convenience the interface `PlatformTransactionManager` of Spring is used. This interface defines a transaction strategy whose notion is the key to the Spring transaction abstraction. For further information about the Spring Transaction Management please refer to <http://static.springframework.org/spring/docs/2.5.x/reference/transaction.html>.

For transaction management with the Spring Framework's declarative transaction implementation the XML snippet below shows the Spring configuration:

```
<bean class="org.springframework.aop.framework.autoproxy
    .DefaultAdvisorAutoProxyCreator"
/>

<bean class="org.springframework.transaction.interceptor
    .TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="txInterceptor" />
</bean>

<bean id="txInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager" />
    <property name="transactionAttributeSource">
        <bean class="org.springframework.transaction.annotation
            .AnnotationTransactionAttributeSource"/>
    </property>
</bean>
```

The `DefaultAdvisorAutoProxyCreator` creates AOP proxies for all Spring beans. The usage of AOP proxies enables you to define your transactional advices as metadata in XML or as Java 6 annotations around method invocation in Java classes. The `TransactionAttributeSourceAdvisor` is an advisor driven by `AnnotationTransactionAttributeSource` which is used for transaction metadata as Java 5 annotations. The `TransactionInterceptor` is a method interceptor for declarative transaction management. The code snippet below shows an example of defining a transactional advice with the `@Transactional` annotation. Here the method `loadCurrentAllergiesByScope` is executed in a single read-only transaction. The transaction is created before the method is invoked and closed after the method finished.

```
@Transactional(readOnly = true)
public AllergyXto[] loadCurrentAllergiesByScope(String scope) {
    ...
}
```



Note: Please note that you can use the Spring transactional advices only on Spring beans, i.e. you must define your Java class as a Spring bean. Otherwise the Spring container does not recognize this Java class and therefore you are not able to use the Spring transaction mechanism.

Spring Templates

Template-based infrastructure provided by Spring simplifies the eHF application. A template uses a callback approach to free application code from having to do deal with basic code for a particular issue such as exception handling. Spring provides templates

for an integration of several technologies. These take advantage of the Template Method Pattern. The eHF uses the `HibernateTemplate` and `TransactionTemplate`.

The `HibernateTemplate` is a helper class that reduces custom Hibernate data access code. It encapsulates the access to the Hibernate API and provides Hibernate Session Handling. It also provides several convenience methods such as `find`, `load` or `delete` for typical single step actions.

The eHealth Framework uses the `TransactionTemplate`, for example, in case of import processes. As well as in case of other processes which do not use declarative transaction management with annotations.

Furthermore Spring templates translate technology-specific exceptions. In the case of Hibernate, this means that checked exceptions such as `SQLException` will be wrapped in objects within the Spring's exception hierarchy with the root class being the unchecked `DataAccessException`. This allows for the possibility of effortlessly changing the underlying persistence technology. Additionally you no longer have to deal with technology-specific exceptions. For further information about eHF exception handling please refer to [Exception and Error Handling](#) on page 127.

Spring Web Services

Generally the eHF uses AXIS 1.4 for Web Services. However the Security Token Service (see section [Token Service](#) on page 127) implement the WS-Trust specification which is not supported by AXIS 1.4. In this particular case Spring Web Services (see <http://static.springframework.org/spring/docs/2.5.x/reference/remoting.html#remoting-web-services>) are used.

E.1.1 Public API in Spring

Defining the public API for Java is only one side of the coin. In order to manifest solid module boundaries, we have to define which beans inside our module-internal Spring contexts should be accessible from outside. Exposing all bean definitions violates the concept of implementation hiding. A service consumer can reference and inject internal beans directly in a way that is not intended by the service provider. For that reason, a service module provider always should define a set of publicly available Spring beans. This set should be a subset of all defined module beans and contains only those which are of interest for the consumers. Keeping this set as small as possible lets both parties benefit. The consumer only has to focus on the set of public beans instead of combing through all Spring context files provided by a module. On the other side, the module could be developed further. All internal beans could be refactored without risking any incompatibilities with existing service consumers.

The central place for the definition of the Spring public API is the module's module context. Its default location is the source folder `src/main/java`. It is a Spring context file which declaratively states the ingoing and outgoing bean from the module's perspective:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ehf="http://www.intercomponentware.com/schema/ehf-commons"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.intercomponentware.com/schema/ehf-commons
           http://www.intercomponentware.com/schema/ehf-commons/ehf-commons.xsd">
    <ehf:module id="moduleId">
        <ehf:configLocations>
            [...]
        </ehf:configLocations>

        <ehf:export>
            [...]
        </ehf:export>
    </ehf:module>
</beans>
```

```

</ehf:export>

<ehf:import>
    [...]
</ehf:import>
</ehf:module>
</beans>

```

You see, that a custom XML namespace is used to specify the module's public Spring context. In order to use the namespace, make sure that the XML namespace is always declared in the `<beans>` tag. The `ehf:module` element starts the definition of a module context. All module contexts have a unique id (typically the module id) to be able to distinguish them. Each module context definition always has the following three recurring sections:

1. `ehf:configLocations` - Here, the Spring context files are referenced which provide the full set of beans for the given module. For generated modules, it is sufficient to list the runtime context only.
2. `ehf:export` - The export element lists all Spring beans within the config locations which are exposed by this module. All not mentioned beans are explicitly hidden and will not be visible and accessible for consumer modules.
3. `ehf:import` - Defines which beans are imported from the platform into the module. It declaratively describes the dependencies to Spring beans defined in other modules or the platform, respectively

Config Locations

The Spring module context is able to reference multiple Spring context files which hold all the beans of a module. The config location specification resembles the known Spring import declaration.

```

<ehf:configLocations>
    <ehf:value>classpath:/META-INF/ehf-module-runtime-context.xml</ehf:value>
</ehf:configLocations>

```

You can use all referencing schemes known by Spring (e.g. the classpath notation seen above) to reference Spring context files. The module context is an own Spring application context which internally imports all beans contained in the declared context files.

Export Section

The export section of the Spring module context defines the beans which are contributed to the platform context for the given module. By exporting beans they are available for other modules. Beans are always exported by referencing the bean id and the interfaces the bean implements. As Spring highly encourages the use of interfaces for better isolation, we want to apply this principle for module contexts as well.

```

<ehf:export>
    <ehf:bean id="myBean" interface="java.util.List" />
</ehf:export>

```

If your bean implements multiple interfaces, you could switch to the alternative export declaration where you can state multiple interfaces of a bean.

```

<ehf:export>
    <ehf:bean id="myBean">
        <ehf:interface name="java.util.List"/>
        <ehf:interface name="java.lang.Iterable"/>
    </ehf:bean>
</ehf:export>

```

All beans which are part of the referenced Spring context files and which are not declared as exported are explicitly declared as private and hence are not visible for module consumers. The Spring module context hereby acts like a filter which only

returns the beans defined in the export section. Referencing another, private bean from the Spring module context will result in a `org.springframework.beans.factory.NoSuchBeanDefinitionException`.

Import Section

The import section of a module context file declares the module's dependencies to external beans. These beans either come from other modules (e.g., some service) or are defined directly in the platform context (e.g., the PlatformTransactionManager). There are two ways to import beans into the module:

1. By Bean Name
2. By Implementing Type

To import a bean by name the `ref-bean` tag is used:

```
<ehf:import>
    <ehf:ref-bean alias="someBean" interface="com.icw.ehf.foo.SomeType" />
</ehf:import>
```

Here, the Spring bean with the name specified by the `alias` attribute is imported. Given the bean could be found it is available in the module context under this very name. Using the `id` attribute, an alternative, module internal name can be specified.

To import a bean by type the `ref-type` tag is used:

```
<ehf:import>
    <ehf:ref-type alias="someListeners" interface="com.icw.ehf.foo.MyListener" />
</ehf:import>
```

Here, *all* beans implementing the specified interface are imported from the platform. By default a bean of type `java.util.List` is registered under the name specified by the `alias` attribute. The type can be changed to `java.util.Set` by setting the optional `type` attribute to `set`. The default value for the `type` attribute is `list`.

E.1.2 Contribution of the Module Context

An assembly has information about all containing modules and needs do get the Spring module context file from each relevant module. This is done by reusing the existing eHF contribution mechanism. Assembly Spring configuration files contain exactly one token for the Spring module context files from all contained modules.

```
<beans>
    [...]
    @@@module.context.import.fragment@@@
    [...]
</beans>
```

Each module which wants to contribute its module context to an assembly has to provide a file called `module.context.import.fragment` located in `src/main/config/merge/assembly`. The token in the assembly is resolved with the contents of the various contribution files. For modules, the contribution file `module.context.import.fragment` contains the import statement for the module context.

```
<import resource="classpath:/META-INF/my-modules-module-context.xml" />
```

This implies that the global platform context of the assembly having a reference to the module's module context. The module's module contexts will only expose the as exported declared beans and prohibit direct access to internal bean definitions.

E.1.3 Assembly contributions into module contexts

As described above, module contexts could contribute the module's beans to a platform context. As described so far, the module context contributions are highly static. Assemblies

are not able to customize the contributed Spring module contexts. There exists various reasonable customization scenarios. For example, think of an implementation of the observer pattern inside a module. To keep the module decoupled, it should not make any assumptions about any registered listeners. Only in the platform context, the listeners need to be injected. In this particular case, the contributed module context has to be extended by injecting the listeners into the correct bean. In eHF, module context contributions could be customized by defining so called extensions in the platform context inside an assembly. Currently, we distinguish between three types of extensions for different purposes:

Bean Extension

By defining a bean extension an additional bean is registered in a target module context.

```
<ehf:extension target-module-id="myModule" class="com.icw.ehf.myModule.MyBean">
    <property name="name" value="myBeanName" />
    <property name="comments">
        <list>
            <value>This is an extension bean.</value>
        </list>
    </property>
</ehf:extension>
```

By placing the above snippet of Spring configuration inside an assemblies platform context, a new bean will be registered in the module context of the module named `myModule`. The attribute `target-module-id` refers to the unique module id for a module which is assigned in the Spring module context. In the example above, an anonymous bean of type `com.icw.ehf.myModule.MyBean` is registered inside the target module context and its property values are injected. Note that you can use the full Spring configuration expressiveness inside the `ehf:extension` element.

Bean Definition Extension

A plain bean extension only adds a single bean into a module context. With it, no wiring could be performed. In most cases, influencing the bean wiring of a module context is intended to express customizations of a module. eHF modules and their module contexts follow the Open/Closed principle. They are open for extensions but closed for modification. Such subsequent extensions could be expressed by an assembly by using bean definition extensions. For example, we have defined a bean with name "myBean" of type `com.icw.ehf.myModule.MyBean` in the module context.

```
[...]
<bean name="myBean" class="com.icw.ehf.myModule.MyBean" />
[...]
```

As we have seen in the previous section, the `MyBean` class has a String property `name` and a property `comments` storing a list of Strings. With bean definition extensions we are able to extend the bean definition from above.

```
[...]
<ehf:definition-extension target-module-id="myModule" target-bean-name="myBean">
    <ehf:property name="name" value="myBeanName" />
    <ehf:property name="comments" ref="commentList" />
</ehf:definition-extension>

<bean id="commentList" class="org.springframework.beans.factory.config.ListFactoryBean">
    <property name="sourceList">
        <list>
            <value>This is an extension bean.</value>
        </list>
    </property>
</bean>
[...]
```

By defining the above snipped inside an assembly's platform context, the module context of the module with ID `myModule` is extended. More precisely, the property definition of the

bean named `myBean` is extended with values from the platform context. In our example, this bean's properties `name` and `comments` are overwritten with a String and a list value respectively. Note, that both String and list are defined in the platform context and extend the module's context. The inner `<ehf:property>` both allow direct value specification and referencing other bean definitions. A referenced bean definition may be specified in the platform context (as in the example above) or could be a module contribution. In the latter case, the bean has to be declared as exported bean to be visible in the platform context. Bean definition extensions rather work on the bean definition than on bean instances. That means that such extensions are applied before any of the beans of an Spring application context is instantiated.

Bean Instance Extension

Due to Spring's life cycle, once a bean is instantiated, it can not be customized by a bean definition extension. However, there exist many cases where the concrete bean instance needs to be customized. This could be achieved by defining a bean instance extension. With this most flexible type of an extension you could code your extension behavior in Java. In the following example, we again want to extend a bean of type `com.icw.ehf.myModule.MyBean`. The extension behavior has to be implemented as a `BeanInstanceExtension` of package `com.icw.ehf.commons.spring.context.extension`. For convenience, the behavior could be implemented by extending the default `DefaultBeanInstanceExtension` implementation:

```
package com.icw.ehf.extension;

public class MyBeanExtension extends DefaultBeanInstanceExtension<MyBean> {

    private String platformComment;

    public String getPlatformComment() {
        return platformComment;
    }

    public void setPlatformComment(String platformComment) {
        this.platformComment = platformComment;
    }

    @Override
    public MyBean extendAfterInitialization(MyBean bean)
        throws ExtensionNotPossibleException {
        if (bean.getComments() == null) {
            bean.setComments(new ArrayList<String>());
        }
        bean.getComments().add(platformComment);
        return bean;
    }
}
```

In `extendAfterInitialization` the bean to be extended is given as argument. After performing the bean extension, the changed bean has to be returned. A bean instance extension could have properties like `platformComment` in the example above. Depending on these property values, the bean could be customized. Because the given bean is already completely initialized, complete information of its object graph could be obtained and influence the bean extension. The bean instance extension is applied directly after the initialization of the bean has been finished. To apply a bean instance extension, it has to be defined in the platform context:

```
[...]
<ehf:instance-extension target-module-id="myModule"
    target-bean-name="myBean" class="com.icw.ehf.extension.MyBeanExtension">
    <ehf:extension-property name="platformComment"
        value="This is a platform comment"/>
</ehf:instance-extension>
```

[. . .]

The `<ehf:instance-extension>` element specifies the target Spring module context (`target-module-id`) and the bean to be extended (`myBean`) plus the type of the bean instance extension. To populate the bean instance extension with properties itself, inner elements `<ehf:extension-property>` are used. Same as for bean definition extensions both direct values and bean references are allowed.

E.2 Hibernate

The eHF uses Hibernate (<http://www.hibernate.org> ➤) as an O/R mapper on the persistence layer. Hibernate is an open source persistence framework for Java. It allows you to store an object's state in a relational database and in turn recreates objects from the appropriate records without you having to code the database request explicitly in SQL. Hibernate provides an API which allows the programmer to perform basic CRUD (create, read, update, delete) operations, retrieve objects based on search conditions, or react to events provided by Hibernate. This section introduces the Hibernate features used in the eHF.

Hibernate Annotations

Hibernate requires additional information to be able to transform Java objects to their relational representation. This information is called object/relational mapping metadata. The metadata specifies the mapping between classes and tables, attributes and columns, associations and foreign keys, Java types and SQL types.

The eHF uses Java 6 and the EJB3/Hibernate annotations (<http://www.hibernate.org/397.html> ➤), instead of XML files, to provide the mapping metadata. With these annotations it is possible to insert the metadata Hibernate requires for object-relational mapping directly into the source code without the intermediate step of creating XML mapping metadata files. This allows you to maintain the object model and the mapping information in one place. It is therefore easier to keep the mapping metadata up to date with the object model.

The eHF Generator (see [Generator](#) on page 88) creates the annotations for a domain object. The annotations that are created by the generator depend on the stereotypes (see [Stereotype ehf-domainobject](#) on page 303) and tags within the domain model.

Database Schema Exporter

The Hibernate3 Tools (<http://www.hibernate.org/255.html> ➤) include some useful features for Eclipse and Ant.

The eHF uses Hibernate's tools to generate the database schema. They provide an Ant task for running the database schema generation. This Ant task `hbm2ddl` runs the `SchemaExport`, a tool to export table schema specific to the target database. It generates the SQL that will be stored in a file (`SCHEMA-CREATE.SQL`). This is carried out as part of the build process. The eHF uses the `ehf-build-maven-plugin` (see [Build Plug-in](#) on page 70) to generate the database schema automatically. Within this plug-in an Ant task `hibernatetool`, is defined, which executes the `hbm2ddl` task.

Currently the eHF supports two databases: HSQL, and Oracle. For each of these databases the Maven plug-in creates one SQL file `schema-create.sql` with the table schema. It also creates this file for MySQL and PostgreSQL, although they are as yet unsupported. This provides the possibility to use the eHealth Framework with more than these two databases.

The domain objects which have to be persisted are defined within the Hibernate XML configuration file `<modulename>-hibernate-cfg.xml` (an example is shown below).

The Hibernate XML configuration file is then used to publish them to Spring (see [Spring Framework](#) on page 332). This file is also created by the eHF Generator:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- Domain Object Mapping -->
    <mapping class="com.mycompany.mymodule.domain.DomainObjectName" />
  </session-factory>
</hibernate-configuration>
```

Hibernate Interceptors and Listeners

Hibernate interceptors enable you to extend the existing functionality with extensible or add-on features. The interceptor interface of Hibernate provides callbacks from the session to the application. These allow you to intercept the methods for saving, updating, deleting and loading of objects. You are able to inspect and manipulate properties of a persistent object before executing these methods. Thus, the main source code is able to mesh actively with the persistence.

The eHF implements some interceptors and listeners but these are legacy and will be replaced in the future. The implementation will be carried out in the object manager because there you also have the context information about objects.

Hibernate Query Language (HQL) and Criteria

The eHF uses both the Hibernate Query Language (HQL) and the criteria query API for retrieving objects from the database, although the latter is generally recommended. This is because it is more robust when it comes to changes in eHF and the model. With HQL you have to be careful not to accidentally expose yourself to a SQL injection.

Criteria, for example, dissolves the complexity of a query for you. Implementing against the criteria query API enables the building of nested and structured query expressions in Java with compile-time syntax checking. This is not possible with a query language like HQL or SQL. By means of Criteria QBE (Query By Example) as advanced Criteria, for example, it is even possible to create queries with given object structures.

E.3 Java Authentication And Authorization Service

The Java Authentication and Authorization Service (JAAS) is the standard security infrastructure provided as part of the Java platform. The eHealth Framework's (eHF) security mechanisms are built on top of JAAS. It is thus a standards-compliant approach that can be integrated in various environments.

The two fundamental concepts of JAAS are **authentication** and **authorization**, covered by eHF's Authentication and Authorization modules, respectively (see [Authentication](#) on page 153 and [Authorization](#) on page 162). The former deals with determining who is currently executing Java code. The latter is responsible for ensuring that only permitted actions can be performed.

Details on JAAS can be found under <http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html>.

F Modularization Addendum

Modularization has been described in detail in the above documentation. In this addendum further details concerning modularization in the scope of the eHF are collected to complement the aforementioned information.

F.1 Further Advantages of Modularization

Further advantages when working in modules are that:

- The probability of conflicts is reduced. As a product is modularized it is easy to also distribute work according to those modules.
- Modules can be reused in other products. Especially when their API is well-defined and the problem domain they address is not too specific the probability for reuse is very high.
- Modular architectures are easier to maintain and to evolve. Modularization is an instrument to decompose an architecture into smaller manageable units.
- Public APIs enable you to define backwards compatibility countermeasures, while reducing overhead to the absolute minimum. In particular when developing a module that is deployed with several products backward compatibility (on all exposed interfaces) is a central non-functional requirement.

F.2 Related Topics and Concepts

- Modularization is in principle a domain and code partitioning strategy. Domain partitioning is especially useful when you have to deal with a complex domain. Trying to define and standardize the full domain is a task of significant complexity. Code partitioning is very useful when you anticipate that your code will evolve. There may be a controlled and slow evolution in some parts, while other parts evolve overnight and change frequently. For some domains this is quite natural, as the knowledge about the domain increases and new specifics are introduced into the model and implementation. In order to keep the rapidly evolving parts isolated, anti-corruption layers can be introduced. In eHF the public API is regarded as an anti-corruption layer for consumers of a module.
- As a consequence of the rules stated above, a modularized system may not be optimized for performance. This general statement means that you do not know the performance of a system and where to optimize until the system is measured in a particular real-life usage scenario. As a result of such an analysis the API of a module can either be enhanced to better support and optimize the usage scenario or - especially in project specific cases - to violate the public API. The latter should only be done with care while balancing the benefits, introduced issues, and the associated efforts. Such optimization is assumed to be very localized and narrow. The overall modularization concept and its benefit is not considered to be at risk.
- Modularization lends itself to a module life cycle. Modules are split, and responsibilities are moved and divided. Some modules will probably also face retirement after a certain period of time. This evolution is a natural extrapolation of the evolving domain of your module towards the product level.

G Build Plug-in Configuration

The easiest, yet most powerful way to influence the eHealth Framework's build process is to override the properties it uses for its configuration. Most of these properties have reasonable default values. Those that do not are typically set by eHF's project templates. The following is a comprehensive list of properties that you can override in your project's `project.properties` file. The property's default values are shown in square brackets.

General Properties

The single most important property for the overall build process is `maven.ehf.build.type`. It determines which kind of build is performed. It is set to an appropriate value by eHF's project templates. All other properties in this section define some basic paths. There is usually no need to change them.

maven.ehf.build.type [BUILD_TYPE_NOT_SPECIFIED]

The type of build to perform. Must be either `module` or `assembly`.

maven.ehf.build.type.ext [jar]

The type of output artifact being produced. Either `jar` or `war`. Set automatically by the appropriate goals.

maven.ehf.build.release [false]

Setting this property to `true` prevents property replacement and fragment cleanup operations. This is used for release builds.

maven.ehf.build.work.{..}.dir [\${maven.build.dir}/tmp{..}]

Several temporary working directories.

maven.ehf.build.resource.dir [\${basedir}/src/main/resources]

The source location of resource files (that is, non-Java) that are required for the module/assembly build. This location is included in the project's build and runtime classpath.

maven.ehf.build.test.resource.dir [\${basedir}/src/test/resources]

The source location of resource files (that is, non-Java) that are required for the test build. This location is included in the project's test build and runtime classpath.

maven.ehf.gen.dir [\${basedir}/src/main/gen]

The output location of generated files and merged configuration files.

maven.ehf.gen.meta.dir [\${maven.ehf.gen.dir}/META-INF]

The output location of generated files and merged configuration files.

Configuration Properties

This group of properties determine files and directories that contain configuration data.

maven.ehf.configuration.dir [\${basedir}/src/main/config]

The source location of configuration files (that is, non-Java). Files in this location are typically not included in the project's build classpath, although some may be accessible at runtime.

maven.ehf.configuration.module.dir [\${maven.ehf.configuration.dir}/module]

maven.ehf.configuration.filename [configuration.properties]

A property file whose values are used to replace property tokens (enclosed within @@ markup) in other configuration files with their values defined in this file. Moreover, this file is made available to the system context at runtime. Typically contains database configuration information.

maven.ehf.configuration.module.file [\${basedir}/module.configuration.properties]

A property file whose values are used to replace property tokens (enclosed within @@ markup) in other configuration files with their values defined in this file. Used for module builds. When building a configuration JAR in a multiartifact build, the values given in

this file are replaced in the configuration JARs content, while all other configuration files are not applied.

maven.ehf.configuration.assembly.file [\${basedir}/assembly.configuration.properties]

A property file whose values are used to replace property tokens (enclosed within @@ markup) in other configuration files with their values defined in this file. Used for assembly builds.

maven.ehf.configuration.users.file [\${basedir}/users.properties]

Used in assembly builds to inject passwords into schema creation scripts.

maven.ehf.configuration.deploy.file [\${basedir}/\${maven.ehf.configuration.filename}]

A property file whose values are used to replace property tokens (enclosed within @@ markup) in other configuration files with their values defined in this file. Not used for release builds.

Merging Properties

This set of properties is used for merging configuration files from various file fragments.

maven.ehf.merge.source.{module|assembly}.dir [\${maven.ehf.configuration.dir}/merge/{module|assembly}]

The source directory of fragment files that are to be merged with other files, for either module or assembly builds.

maven.ehf.merge.source.binary.assembly.dir [\${maven.ehf.configuration.dir}/merge/assembly]

?

maven.ehf.merge.fragment.excludes []

?

maven.ehf.merge.source.ext [fragment]

The file extension of fragment files.

maven.ehf.merge.source.exclude [index.jsp,webgui/index.xhtml]

Files that are explicitly excluded from merging when building WAR files.

Generator Properties

This set of properties controls the generation process.

maven.ehf.generate.force [false]

Setting this property to true will force the generator to run during the build process. Otherwise the generator is not invoked if the generator's output artifacts are newer than its input model file.

maven.ehf.generate.support [true]

Setting this property to false completely disables the generator during the build process. Useful for library modules.

maven.ehf.generate.work.dir [\${maven.ehf.build.work.dir}/generator]

A temporary working directory for the generator.

maven.ehf.generate.file [\${basedir}/src/main/oaw/generated.jar]

The generated artifacts are collected and jarred in this file.

maven.ehf.generate.target.dir [\${maven.ehf.gen.dir}]

The target location of generated artifacts.

maven.ehf.generate.onetime.dir [\${basedir}/src/main/java]

The location of artifacts that are generated only once. If a target file already exists in this location, it is not touched by the generator.

maven.ehf.generate.oaw.config [\${basedir}/src/main/oaw/workflow.oaw]

The oAW workflow file to use for the generation process.

maven.ehf.generate.model.dir [\${basedir}/src/main/model]

The source location where the model file must be located. Note that the actual model file being used is specified in the oAW workflow file (see `maven.ehf.generate.oaw.config` property). Both locations must be changed if necessary.

Database Properties

This list of properties controls database-related goals.

maven.ehf.db.support [true]

unused

maven.ehf.db.schema.generate.history.tables [false]

Determines whether history tables will be generated in the database schema.

maven.ehf.db.schema.path [META-INF/db]

?

maven.ehf.db.schema.binary.path [\${maven.ehf.jar.meta.config.dir}/db]

?

maven.ehf.db.schema.output.dir [\${maven.ehf.gen.dir}/ \${maven.ehf.db.schema.path}]

Determines the output path of generated schema SQL files.

maven.ehf.db.schema.dir []

The directory where schema SQL files are read from. If unset, this defaults to `${maven.ehf.db.schema.output.dir}`, that is the output path of the generated schema files.

maven.ehf.db.schema.create.file [init.sql]

The name of the SQL file that drops and creates a schema (not the tables within a schema).

maven.ehf.db.tables.create.file [schema-create.sql]

The name of the SQL file that drops and creates tables within a schema.

maven.ehf.db.tables.custom.file [schema-custom.sql]

The name of an additional, typically manually created SQL file that is executed after a schema's tables have been created.

maven.ehf.db.schema.generate.tde [false]

unused

maven.ehf.db.initialize [false]

Flag to turn turn on database initialization (that is, data import) for module builds.

maven.ehf.db.initialize.{bootstrap|import} [false]

Flag to turn individually turn on or off database initialization (that is, data import) for bootstrap and import data in module and assembly builds. Note that for assembly builds, data import is usually vital and should explicitly turned on.

maven.ehf.db.initialize.content [false]

unused

maven.ehf.db.initialize.{bootstrap|import}.pattern [classpath:/META-INF/ehf-assembly-{bootstrap|import}.xml]

The location of a Spring XML application context that controls the bootstrap and import database initialization process. The given file must be accessible via the classpath and must therefore be prepended with the classpath: protocol.

maven.ehf.db.initialize.import.content.pattern [classpath:/META-INF/ehf-assembly-content.xml]

The location of a Spring XML application context that controls the content import process triggered exclusively by the ehfdb:content-import goal. The given file must be accessible via the classpath and must therefore be prepended with the classpath: protocol.

maven.ehf.security.policy.file [\${basedir}/ehf-assembly/bootstrap/bootstrap.policy]

The JAAS policy file that is used for importing data into the database.

maven.ehf.db.dialects [see below]

The Hibernate dialects which schema SQL files are generated for. Defaults to org.hibernate.dialect.Oracle9Dialect, org.hibernate.dialect.HSQLDialect, com.icw.ehf.commons.hibernate.dialect.CustomMySQL5Dialect, org.hibernate.dialect.PostgreSQLDialect

maven.ehf.db.dialects.default [hsqI]

The default dialect to be used if an unknown dialect is given in the maven.ehf.db.dialects list.

maven.ehf.db.started [false]

Flag that determines whether a HSQL database instance is running. If false, the build process attempts to start a temporary instance.

maven.ehf.db.skip.prepare [false]

If true, the database is not started and stopped automatically during the build and test process. Moreover, no database bootstrap and data import steps are performed.

maven.ehf.db.fullname [target/db/testdb]

The name of the temporary database instance that is started and stopped during the build process.

Application Initialization Properties

This list of properties controls the application initialization phase. This is in particular used for setting up key stores or other information relevant for application level encryption and pseudonymization. The parameters work in alignment with the database bootstrap properties.

configuration.initialize.pattern [empty]

Defines a pattern for a Spring application context. The Spring application context will be started. Normally the context hosts beans that automatically initialize and trigger certain activities. This is an example taken from the eHF reference assembly:

```
configure.initialize.pattern=classpath:/META-INF/ehf-assembly-initialize.xml
```

configure.initialize.policy [./ehf-assembly/bootstrap/bootstrap.policy]

Points to a policy file that is used during the application initialization.

configuration.processor.java.xms [768m]

JVM setting for the initialization phase.

configuration.processor.java.xmx [768m]

JVM setting for the initialization phase.

configuration.processor.java.xss [256k]

JVM setting for the initialization phase.

Multiartifact Properties

The eHF Build Plug-in is capable of producing several output JARs per module. You can control this process with these properties.

maven.ehf.multiartifacts [false]

Deprecated old flag for multiartifact configuration.

maven.ehf.multiartifacts.runtime [\${maven.ehf.multiartifacts}]

If true, a runtime JAR artifact is created during the build process.

maven.ehf.multiartifacts.runtime.dir [\${maven.build.dest}]

The directory where the runtime JAR's contents are collected from.

maven.ehf.multiartifacts.runtime.targetdir [.]

The directory where the runtime JAR's contents will temporarily be copied to prior to being JARred. This property should never be changed.

maven.ehf.multiartifacts.runtime.include.pattern [/*]**

The files to be included in the runtime JAR.

maven.ehf.multiartifacts.runtime.exclude.pattern [see below]

The files to be excluded from the runtime JAR. The lengthy default setting basically purges all but the spring configuration files from the JAR:

```
**/META-INF/configuration.properties,**/META-INF/config/**/*,**/META-INF/config,**/META-INF/module/**/*,**/META-INF/assembly/**/*,**/META-INF/db/**/*,**/META-INF/db,META-INF/${pom.artifactId}/**/*,META-INF/${pom.artifactId},**/hibernate.properties,**/system-context.xml,**/*.html,**/*.java,**/META-INF/*.properties,**/META-INF/merge/**/*,**/META-INF/merge
```

maven.ehf.multiartifacts.api [\${maven.ehf.multiartifacts.runtime}]

If true, an API JAR artifact is created during the build process. By default it is created whenever a runtime JAR is created.

maven.ehf.multiartifacts.api.dir [\${maven.build.dest}]

The directory where the API JAR's contents are collected from.

maven.ehf.multiartifacts.api.targetdir [.]

The directory where the API JAR's contents will temporarily be copied to prior to being JARred. This property should never be changed.

maven.ehf.multiartifacts.api.include.pattern [see below]

The files to include in the API JAR. Defaults to `**/transfer/*Dto.class, **/transfer/*Xto.class, **/adapter/*DtoAdapter.class, **/service/adapter/*DtoAdapter.class`

maven.ehf.multiartifacts.api.exclude.pattern []

The files to exclude from the API JAR.

maven.ehf.multiartifacts.config [false]

If true, a JAR containing configuration data is created during the build process. Only property substitutions from the `maven.ehf.configuration.module.file` configuration file are applied.

maven.ehf.multiartifacts.config.dir [\${maven.ehf.configuration.dir}]

The directory where the configuration JAR's contents are collected from.

maven.ehf.multiartifacts.config.targetdir [META-INF/config]

The directory where the configuration JAR's contents will temporarily be copied to prior to being JARred. This path will effectively be prepended to the JARred files.

maven.ehf.multiartifacts.config.include.pattern [/db/**/*, **/merge/assembly/**/*]**

The files to be included in the configuration JAR.

maven.ehf.multiartifacts.config.exclude.pattern []

The files to be excluded from the configuration JAR.

maven.ehf.multiartifacts.bootstrap [false]

If true, a JAR containing database bootstrap data is created during the build process.

maven.ehf.multiartifacts.bootstrap.dir [\${basedir}/src/main/resources/META-INF]

The directory where the bootstrap JAR's contents are collected from.

maven.ehf.multiartifacts.bootstrap.targetdir [META-INF]

The directory where the bootstrap JAR's contents will temporarily be copied to prior to being JARred. This path will effectively be prepended to the JARred files.

maven.ehf.multiartifacts.bootstrap.include.pattern [\${pom.artifactId}//*]**

The files to be included in the bootstrap JAR.

maven.ehf.multiartifacts.bootstrap.exclude.pattern []

The files to be excluded from the bootstrap JAR.

maven.ehf.multiartifacts.webapp [false]

If true, a JAR containing web application data is created during the build process.

maven.ehf.multiartifacts.webapp.dir [\${basedir}/src/main/webapp]

The directory where the bootstrap JAR's contents are collected from.

maven.ehf.multiartifacts.webapp.targetdir [META-INF/webapp]

The directory where the bootstrap JAR's contents will temporarily be copied to prior to being JARred. This path will effectively be prepended to the JARred files.

maven.ehf.multiartifacts.webapp.include.pattern [/*]**

The files to be included in the webapp JAR.

maven.ehf.multiartifacts.webapp.exclude.pattern []

The files to be excluded from the webapp JAR.

maven.ehf.upload.file [\${maven.release.name.bin}]

In case of non-multiartifact setup, this property specifies the single file that will be uploaded into the Maven repository.

maven.ehf.upload.file.type [SNAPSHOT]

In case of non-multiartifact setup, this property specifies whether the single artifact will be uploaded into the Maven repository as a snapshot or not.

Release Properties

These properties are used during the creation of a release artifact for an assembly.

maven.ehf.release.configuration.include.pattern [/*]**

The files from the assembly's configuration folder to be included in the release artifact's configuration folder.

maven.ehf.release.configuration.exclude.pattern []

The files from the assembly's configuration folder that should **not** be included in the release artifact's configuration folder.

maven.ehf.release.config.artifacts.pattern [(.+)-config]

Determines which "*config*" dependencies, listed in the pom, will be matched for the purposes of copying the contents of their configuration folders to the release artifact's configuration folder.

maven.ehf.release.extension [-bin]

The extension to be used on the end of the filename of the created release artifact. With the default of *-bin*, an eHF based assembly with an artifactId (defined in the pom) of "ehf", and a currentVersion (also defined in the pom) of "SNAPSHOT", the resultant release artifact would be "ehf-SNAPSHOT-bin.zip".

Miscellaneous Properties

These properties don't match any of the previous categories.

maven.ehf.build.axis.deploy.context.file [/META-INF/axis-deploy-context.xml]

unused

maven.ehf.jar.webapp.dir [META-INF/webapp]

The directory that contains web application meta information. Relevant files are copied there, and are merged from/to this this directory when assembling a WAR file.

maven.ehf.jar.meta.config.dir [META-INF/config]

Several configuration files are collected in this directory.

maven.ehf.web.support.copy [false]

?

maven.ehf.webapi.support.local [true]

unused

maven.ehf.webcontext.wait [3]

unused

maven.ehf.filter.classpath [true]

If true, a module can only access other modules' API. This is accomplished by modifying the compilation classpath to include the modules' API JARs only.

maven.ehf.jar.excludes [see below]

A lengthy list of files to exclude from packaging into a JAR file. This property itself is unused. It is, however, used as the default value for maven's standard maven.jar.excludes property. It defaults to `**/package.html, **/*.*.java, **/basic-server-config.wsdd, **/system-context.xml, **/META-INF/merge, **/META-INF/merge/**/*`.

H Installation Script Properties

The Ant installation scripts for eHF-based applications use specific internal properties that are not passed on the command line. eHF provides sensible defaults (when using the eHF assembly project templates), but usually your product will overwrite these properties in order to control the installation procedure. The overwrite of the properties is normally done in your assembly, where you specify the properties you want to adapt in the configuration.properties or any included property file.

The following list shows the properties you may overwrite

```
# standard properties for installing ehf-based distributable artifacts

# configuration file for the deployment
configuration.file=${basedir}/configuration.properties

# standard database attributes
database.schema.path=META-INF/db
database.schema.init.file=init.sql
database.schema.create.file=schema-create.sql
database.schema.custom.file=schema-custom.sql

# bootstrap property defaults
database.bootstrap.pattern=classpath:/META-INF/ehf-assembly-bootstrap-context.xml
database.bootstrap.policy=assembly/deploy.policy

# test import property defaults
database.importtestdata.pattern=classpath:/META-INF/ehf-assembly-import-context.xml
database.importtestdata.policy=assembly/deploy.policy

# content import property defaults
database.import.content.pattern=
database.import.content.policy=assembly/deploy.policy

# properties controlling the memory settings for importing
database.processor.java.xms=64m
database.processor.java.xmx=768m
database.processor.java.xss=256k
database.processor.java.xx.mps=128m

# properties controlling the memory settings for migration process task
migration.processor.java.xms=64m
migration.processor.java.xmx=768m
migration.processor.java.xss=256k
migration.processor.java.xx.mps=128m

# properties supporting the general 'database:import' goal
database.import.policy=assembly/deploy.policy

# username and password for executing the make grants script
main.database.username=ehfuser
```

```

main.database.password=ehfuser

# in default mode remote actions are logged to the console only. Use 'remote' to
# activate.
database.task.execution.mode=default

# details about the host of the database for connecting via ssh
database.host=<DATABASE_HOST_NAME>
database.host.username=<DATABASE_HOST_USERNAME>
database.host.password=<DATABASE_HOST_PASSWORD>
database.host.tmp.path=<DATABASE_HOST_TMP_DIR>/ehf

# properties supporting the general 'configure:initialize' goal
configure.processor.java.xms=64m
configure.processor.java.xmx=768m
configure.processor.java.xss=256k
configure.processor.java.xx.mps=128m

# controlling the initialize processor
configure.initialize.pattern=
configure.initialize.policy=${basedir}/ehf-assembly/bootstrap/bootstrap.policy

```

I AOP Explained

Imagine an architect planning a house with a building's blueprint on a transparent slide and its projection on a screen. The blueprint shows the building's structural elements, like walls, doors, and windows. But it doesn't show the building's plumbing and electrical wiring. These are other aspects. So you take another transparent slide that shows the plumbing, and a third for electrical wiring. While each aspect (structure, plumbing, wiring) is separated on its own slide, stacking them on the projector provides a merged view of the building, including all aspects.

Aspect oriented programming (AOP) works in the exact same way. The idea is to define each concern separately and later merge the concepts into one comprehensive system. The process of merging is called **weaving** in AOP terms.

But AOP can do more. Consider an architectural rule that says that a light switch must be installed to the left of each door. With slides on a projector, you would have to draw several light switches at the correct locations. With AOP, you only need to define one light switch, together with an exact specification of where to put it. The exact specification in this case is *to the left of each door*, which in AOP terms is called a **pointcut**. A pointcut basically selects a set of so-called **joinpoints**, which are elements an aspect can potentially be attached to. For our building example, these would comprise structural elements like walls and doors. The things an aspect wants to weave at these pointcuts are called **advice**. An advice is applied at all joinpoints selected by its pointcut.

With these simple concepts of joinpoints, pointcuts, and advice weaving, AOP can achieve higher levels of modularization than traditional approaches. The actual potential and applicability of different AOP implementations are defined by their specific facilities for joinpoints, pointcuts, and weaving. One AOP implementation may, for example, be able to detect motion inside a building, so using a "*whenever a person enters a room*" pointcut, a "*play the person's favorite music*" advice can be woven. An AOP implementation that can only deal with static structural elements cannot do this.

J MDSD Explained

The underlying concepts of model-driven software development can be explained with an analogy in the field of civil engineering: When creating a new building, an architect employs an appropriate (graphical) language to produce a blueprint (i.e. a model) of the building.

He arranges the major features like walls and doors on a high level of abstraction and can ignore mundane details like bricks and mortar. He is, of course, constrained by the natural laws of physics and statics.

Similarly, a software architect can express software as a model. He chooses (or even creates) an appropriate, high-level language to do this. This language can either be graphical or textual, or even both. The words and grammar of the language, together with a set of constraints define what can be created with this language, and what cannot. In software, however—and there the analogy ends—, these models can be transformed into running systems automatically. No manual labor to stack bricks and mortar is required. Since it is typically neither feasible nor desirable to express the whole software system as an abstract model, you may have to provide additional code for various implementation details—which is equivalent to decorating your living room. The walls around you, however, are already there.

The concepts that can be expressed by such a modeling language, called *abstract syntax*, are defined in a so-called **metamodel**. For civil engineering, the metamodel comprises walls, doors, and windows, for example. The Java metamodel has concepts like classes, interfaces, and methods. Using a **textual or graphical notation**, called *concrete syntax*, you can create **models**, which are effectively instances of metamodels. A blueprint of the Empire State Building is a model in a graphical language. Java files on the filesystem are a kind of textual model consisting of Java artifacts. Both abstract and concrete syntax together form a language. Languages targeted towards a specific purpose are called **domain specific languages (DSLs)**, as opposed to general-purpose languages. In computer science, **transformation, generation and interpretation** techniques are used to produce running systems out of language artifacts. The most typical transformation is the well-known compiler.

From the explanation of the concepts above, it becomes fairly obvious that the software industry has been going along this path for several decades now. Low-level, general purpose languages are superseded by languages that provide a higher level of abstraction. While computer pioneers had to deal with computer instruction code and processor registers, today's software engineers work with logical control structures and virtual memory allocation—while at the same time giving up direct control of the hardware they are working on.

Model-driven software development takes this evolution one step further. It provides tools and frameworks that facilitate the creation and tailoring of languages that operate on the most appropriate level of abstraction for whichever problem is at hand. It also provides tools and frameworks that facilitate the creation and tailoring of editors, transformers, and generators for such languages. These integrate themselves into the regular software development life cycle, and in many ways, working with an MDSD toolchain is not so much different from working with a "classical" toolchain. From a developer's point of view, it is just another compiler, taking source artifacts and producing output artifacts. It's just the artifacts that are different.



Figure 140: MDSD is similar to a compiler

K Fragment Tokens

eHF provides a mechanism to merge fragments into template configuration files. This mechanism is used in various contexts. In especially it is used to merge fragments at

module build time (generated configuration, custom configuration files) and to merge fragments coming from the modules into the assembly configuration at assembly time.

Fragment tokens are represented in template configuration files as @@@token@@@.

K.1 Module-Level Fragment Tokens

On module level fragment tokens support to merge in configuration into templates and into generated configuration artifacts.

beanmapping.wsdd.fragment

The beanmapping fragment can be merged into the WSDD configuration. It contains extra classes and namespace mappings if they cannot be directly derived from the domain model.

The following shows an example mapping.

```
<beanMapping
    xmlns:ns="@@webapi.namespace.prefix@@/record@@webapi.version@@service"
    qname="ns:ObservationViewXto"
    languageSpecificType="java:com.[...].webapi.transfer.ObservationViewXto">
</beanMapping>
```

method.wsdd.fragment

With the method fragment the list of methods exposed on the web service can be defined.

The information has to be provided on a single line and uses blanks as separators.

The example shows a default set of methods that can be exposed.

```
loadMetaData create delete update loadById loadByScope
```

parameter.wsdd.fragment

With the parameter fragment additional classes can be identified to be included in the possible parameter types of the web service. This is in particular required if the the web service signatures are generic and operate on an interface or abstract class level.

For fully generated modules this fragment is not required as all types are represented in the module.

The example shows how a fully qualified class name of an external transfer object being added to the parameter types.

```
com.icw.ehf.record.medical.webapi.transfer.MedicationXto
```

webapi.handler.fragment

With this fragment additional web service handlers can be introduced on module level.

<object-name>.metadata.invariants.fragment

This fragment is only available for generated domain objects. It can be utilized to provide further OCL invariants that are not specified in the model.

It is however recommended to specify all constraints in the model.

```
<invariant>
    <expression>context EmfPet inv: self.price > 0.00</expression>
    <code>
        <key>VAL-000101</key>
        <oid>2.16.840.1.113883.1.2.3.4.5.6</oid>
    </code>
```

```
</invariant>
```

<object-name>.metadata.attributes.fragment

This fragment is only available for generated domain objects. It can be utilized to provide attribute meta data configuration.

```
<string>
<attribute>name</attribute>
<maxLength>128</maxLength>
<minLength>-1</minLength>
<mandatory>true</mandatory>
<exposed>true</exposed>
<contractRelevant>false</contractRelevant>
<modifiable>true</modifiable>
<unique>false</unique>
<persistent>true</persistent>
</string>
```

hibernate.sessionfactory.xml.fragment

This fragment can be used to add further mappings of persistent classes to the Hibernate session factory configuration. For generated modules this should not be required.

```
<mapping class="com.icw.ehf.reference.domain.Pet" />
<mapping class="com.icw.ehf.reference.domain.PetShop" />
```

hibernate.configuration.xml.fragment

This is a fragment placed in the Hibernate configuration. You can use it to define additional configuration. Properties should not be defined on this level.

Generated modules usually do not require any particular configuration here.

```
<class
  name="com.icw.ehf.reference.domain.PetShop"
  table="T_CUSTOM">
```

module.context.config.location.fragment

Using this fragment the generated module context can be extended with further config locations to support the internal module context.

```
<value>classpath:/META-INF/my-custom-context.xml</value>
```

module.context.export.fragment

Using this fragment the generated module context can be extended to export further beans defined in the fragment.

```
<bean id="referencePetService"
  interface="com.icw.ehf.reference.transfer.adapter.PetServiceDtoAdapter" />
```

module.context.import.fragment

Using this fragment the generated module context can be extended with further imports from the platform context. Either the recommended ref-bean approach can be used or the pattern for matching bean names. The latter is deprecated and may be removed soon.

```
<ref-bean alias="securityService"
  interface="com.icw.ehf.authorization.service.SecurityService" />
<pattern>sessionFactory</pattern>
```

K.2 Assembly-Level Fragment Tokens

Every module can contribute configuration to an assembly using fragments. The contributions happen on several levels. There are static contributions and dynamic contributions.

Static contributions are controlled using fragment tokens. In this case a module specifies a fragment file in the `src/main/config/merge/assembly` folder, which is included during the build in the config artifact under the `META-INF/config/merge/assembly` folder.

Please note that assembly level fragments may be contributed by the modules as well as the assembly itself. The latter helps to work on well-defined and generic configuration templates.

The module level assembly contributions are contained in the folder `src/main/config/merge/assembly`.

axis.deploy.context.fragment

This token enables to contribute a Axis WSDD to the assembly. The fragment may look as follows.

```
<value>/META-INF/reference-project-server-config.wsdd</value>
```

axis.compatibility.fragment

Using this token the compatibility configuration can be contributed to the assembly.

```
<service>
    <servicePath>/@@webapi.version@@/ReferenceWebService</servicePath>
    <transformations>
        <transformation>
            <servicePathPattern>/v2\-\1\-\9/ReferenceWebService</servicePathPattern>
        </transformation>
        [...]
    </transformations>
</service>
```

error.resolver.fragment

Using this token error resolver configuration can be contributed to the assembly.

```
<ref bean="usermgmtErrorDetailResolver"/>
```

hibernate.sessionFactory.cfg.fragment

Using this token Hibernate session factory configuration can be contributed to the assembly.

```
<value>classpath:/META-INF/reference-project-hibernate-cfg.xml</value>
```

module.context.import.fragment

Using this fragment the module context configuration can be contributed to the assembly.

```
<import resource="classpath:/META-INF/reference-project-module-context.xml" />
```

make_grants.fragment

For Oracle database support a specialized file is generated on assembly level, which contains contributions from all persistent modules. This token supports the contribution to an appropriate template.

Generated modules normally do not have to care about the provision of the file. It is generated as part of the generation process.

```
prompt ----- Access REFERENCE-PROJECT Scheme (generated) -----;
```

```
define schemaName=EHF_REFERENCE;
```

```
define schemaPass=@@connection.reference-project.pass@@;

connect &schemaName/&schemaPass@@&SID;

@@func.create.sql
execute grantTo('&schemaName', '&usrEHFName');
@@func.drop.sql
```

web-xml.context.params.fragment

Fragment token supporting the contributions of contexts parameters to the web.xml template of an assembly.

```
<context-param>
    <param-name>com.icw.ehf.commons.codesystem.CodeResolver.NAME</param-name>
    <param-value>cachedCodeResolver</param-value>
</context-param>
```

web-xml.filters.fragment

Fragment supporting the contribution of filters to the assembly. It is not recommended to use this contribution from modules, as the filters must be in a particular order. Therefore the filters are normally explicitly configured on assembly level.

```
<filter>
    <filter-name>targetUriSupportingFormLoginFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>

<filter>
    <filter-name>certLoginFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>

<filter>
    <filter-name>PinLoginFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
```

web-xml.filters.mapping.fragment

Fragment for contributing filter mappings to the assembly. As filter mapping normally are complementary to filter definitions the definition as part of a fragment is not recommended from modules.

```
<filter-mapping>
    <filter-name>csrfGuardFilter</filter-name>
    <url-pattern>/webgui/*</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>logoutFilter</filter-name>
    <url-pattern>/webgui/bye</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>PinLoginFilter</filter-name>
    <url-pattern>/webgui/pin/*</url-pattern>
</filter-mapping>
```

[...]

web-xml.listeners.fragment

Fragement token for contributing context listeners to the web.xml template.

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>

<listener>
    <listener-class>
        com.icw.ehf.authorization.web.ContextListener
    </listener-class>
</listener>
[...]
```

web-xml.servlets.fragment

Fragement token enabling contributions of servlet definitions.

```
<servlet>
    <display-name>FileDownload Servlet</display-name>
    <servlet-name>FileDownloadServlet</servlet-name>
    <servlet-class>
        com.icw.ehf.document.servlets.FileDownloadServlet
    </servlet-class>
    <init-param>
        <param-name>default-error-picture</param-name>
        <param-value>@@default-error-picture@@</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

web-xml.servlets.mapping.fragment

This fragment token enables to contribute servlet mappings to the web.xml template.

```
<servlet-mapping>
    <servlet-name>FileDownloadServlet</servlet-name>
    <url-pattern>/filedownload</url-pattern>
</servlet-mapping>
```

web-xml.security.fragment

Security fragment token for contributing security contraints to the web.xml template.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>ehf-resources</web-resource-name>
        <!-- document up- and download -->
        <url-pattern>/fileupload/*</url-pattern>
        <url-pattern>/filedownload/*</url-pattern>

    [...]

    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
    <http-method>DELETE</http-method>
    <http-method>HEAD</http-method>
    <http-method>OPTIONS</http-method>
    <http-method>TRACE</http-method>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
```

</security-constraint>

L Database Privileges Explained

Explaining Users and Privileges

The database administrator creates users that can be used to log into a database. These users are restricted by what they can do in the database by applying privileges. Privileges can either be granted system wide or on a specific schema object in the database.

There are two types of database *users* used by eHF:

1. System user: Contain system level database access that can change overall table structures
2. Data user : Contain data level database access that allow users to add, modify, delete schema object entries

The eHF System user is required during installation or upgrade of a database to change database structures so greater privileges are required.

Every eHF module with a database representation creates a schema containing a collection of tables objects. In the case of Oracle, a schema is a set of objects owned by a *User*, so this is a special scenario. In this situation, the *data user* is actually split into two users:

1. Module users
2. General data user ("eHF User")

The module user names have the same name as the module, so, for example, ehf-codesystem has a database user called EHF_CODESYSTEM. In addition to eHF module users, eHF uses a general data user which accesses all eHF data in the database. This "eHF User" requires privileges to be able to access the eHF module user information.

Multiple Database Sources

eHF supports multiple database sources e.g for audit, encryption, and data, so the System User, eHF User and privileges should be created on all database sources.

Configuring eHF for Users and Privileges

The Oracle DBA is required to create the *System User* on every database source as a prerequisite to installing eHF. This System user is then used to create the *Module users* and the *eHF user*. The system user is configured as follows in the connection.password.properties file:

```
# connection (oracle)
connection.system.username=system
connection.system.password=manager

# audit connection (oracle)
audit.connection.system.username=system
audit.connection.system.password=manager

# encryption connection (oracle)
encryption.connection.system.username=system
encryption.connection.system.password=manager
```

The following shows an example of how to configure an eHF database user. This is done also in the connection.password.properties file:

```
main.database.username=ehfuser
```

```
main.database.password=ehfusrl
```

The following ANT target is required to configure the scripts: `ant -f install/install.xml configure:all`

L.1 Creating Database Privileges using the Ant Installation Process

Creating the System and Module User

The Oracle DBA is required to create the *System User* as a prerequisite to installing eHF.

This System user is then used by the ANT `database:prepare` target to create the Module users.

Creating the eHF User and applying Privileges

The ANT target `database:privileges` is used to create the eHF User and apply privileges. This should be executed at the end of the installation process.

First the eHF database user is created with the following Oracle privileges:

- create procedure
- create session
- create synonym
- create type
- create any table
- drop any table
- create any index
- drop any index
- alter any table
- alter any index
- select any table
- analyze any

Next all the Oracle object tables from the modules are searched using a `grantTo` procedure. This is defined in the `configuration/database/scripts/[oracledb]/create-functions.sql` and created in the database as part of the `database:privileges` ANT target call. All table, sequences and views used by eHF modules are granted `select, insert, update, delete` privileges for the eHF user.

L.2 Creating Database Privileges using SQL Scripts

Creating the System User

The Oracle DBA is required to create the *System User* on all database sources as a prerequisite to installing eHF.

Creating the eHF User

Next, the Oracle DBA can create the eHF User using the script located at `.install/tasks/database/create-datasource-user.sql`. Before executing the script the following values must be configured by the administrator:

Property	Description
<code>@@datasource.username@@</code>	The database username
<code>@@datasource.password@@</code>	The password for the database user
<code>@@datasource.temp.tablespace</code>	The temporary tablespace

Note: The script must be configured and executed for every database source.

The eHF database user is created with the following Oracle privileges:

- create procedure
- create session
- create synonym
- create type
- create any table
- drop any table
- create any index
- drop any index
- alter any table
- alter any index
- select any table
- analyze any

Creating the Module Users

The created scripts are located at `.install/modules` in each module directory:

Script	Description
init.sql	Create the module database user
schema-create.sql	Creates the module schema objects
schema-custom.sql	Customizes the schema e.g. creates indexes

The module user names have the same name as the module, so ehf-codesystem has a user called EHF_CODESYSTEM.

Create Privileges for eHF Database User

The following scripts are provided at `.install/tasks/database`:

Script	Description
create-functions.sql	Create the procedure to add eHF schema privileges to the eHF User
drop-functions.sql	Drops the grant privileges procedure
grant-schema-permissions.sql	Creates the Privileges for the eHF User.

To execute the grants manually using these scripts the administrator should configure the following in the `grant-schema-permissions.sql` script:

```
define adminName=<System User>;
define adminPass=<System User Password>;
define SID=<Oracle Database SID>

define logFile=<Log File>;
define datasourceUserName=<eHF User>;
define datasourceUserPass=<eHF User Password>;
define tsTemporaryData=<Tempoprary Tablespace>;
```

Next the administrator should copy all scripts to the same directory on each database server before executing the `grant-schema-permissions.sql` script.

M Create ATNA Compliant Audit Messages

This provides an example of how to create ATNA compliant audit event messages in an eHF based application.

Overview

The creation of ATNA compliant audit messages in eHF is based on the Event processing facility described in chapter [Event processing](#) on page 40. The ehf-integration-atna module provides a set of transmogrifiers to be used to transform trigger events emitted by the application (e.g., user login) into ATNA compliant audit messages.

In order to set up the creation of ATNA audit messages based on Camel and in pure Java, several steps have to be performed. First configure the transformation necessary to create the audit messages:

1. Declare which trigger event types you would like to transform. In the following example we have picked two types arbitrarily. For a comprehensive list of available transmogrifiers please refer to the ehf-integration-atna documentation.

```
<util:map id="atnaTransmogrifiers">
    <entry key="com.icw.ehf.commons.event.ApplicationLifecycleEvent">
        <bean class="com.icw.ehf.integration.atna.transmogrify.
ApplicationLifecycleEventTransmogrifier" />
    </entry>
    <entry>
        ...
    </entry>
    <entry key="com.icw.ehf.usermgnt.event.PasswordAuthenticationEvent">
        <bean class="com.icw.ehf.integration.atna.transmogrify.
AuthenticationEventTransmogrifier" />
    </entry>
</util:map>
```

2. Declare a composite transmogrifier which will then be used within the Camel route. Besides the transmogrifier map defined previously, some additional transmogrifiers can be added here as "post processors" to augment all ATNA audit messages with additional information.

```
<bean id="atnaTransmogrifier" class="com.icw.ehf.integration.atna.
transmogrify.CompositeTransmogrifier" >
    <property name="transmogrifiers" ref="atnaTransmogrifiers" />
    <property name="postProcessors">
        <list>
            <bean class="com.icw.ehf.integration.atna.transmogrify.
NetworkAccessPointIDTransmogrifier"/>
            <bean class="com.icw.ehf.integration.atna.transmogrify.
SourceIdTransmogrifier">
                <property name="productInstanceIdentifier" ref=
"productInstanceIdentifier"/>
            </bean>
        </list>
    </property>
</bean>
```

The NetworkAccessPointIDTransmogrifier adds the network access point ID, while the SourceIdTransmogrifier sets the audit source ID, using the available ProductInstanceIdentifier.

3. In order to use transmogrifiers in Camel routes without IPF's Groovy extensions a special Camel processor is necessary. The TransmogrifierAdapterProcessor acts as a wrapper for a transmogrifier. Define the following TransmogrifierAdapterProcessor bean wrapping the atnaTransmogrifier composite transmogrifier defined in the previous step.

```
<bean id="atnaTransmogrifierAdapterProcessor" class="com.icw.ehf.integration.
atna.transmogrify.TransmogrifierAdapterProcessor">
    <property name="transmogrifier" ref="atnaTransmogrifier"/>
</bean>
```

This processor can subsequently be employed within a Camel route.

Once the transformations are in place, the next step is to connect to the messaging infrastructure.

1. The destination for all ATNA audit messages is an IHEAuditor. Configure a Spring bean representing your ATNA repository:

```
<bean id="iheAuditor" class="org.openhealthtools.ihe.atna.auditor.IHEAuditor" factory-method="getAuditor">
<property name="config" ref="atnaConfig" />
</bean>

<bean id="atnaConfig" class="org.openhealthtools.ihe.atna.auditor.context.AuditorModuleConfig">
<property name="auditRepositoryHost"
          value="@@atna.audit.repository.host@@" />
<property name="auditRepositoryPort"
          value="@@atna.audit.repository.port@@" />
<property name="auditSourceId"
          value="@@product.atna.audit.source.id@@" />
<property name="auditEnterpriseSiteId"
          value="@@product.atna.audit.enterprise.site.id@@" />
</bean>
```

The place holders marked with @@ are, during the build process, replaced with actual values from an arbitrary properties file residing on the classpath.

2. This IHEAuditor bean is then plugged into a Camel processor:

```
<bean id="atnaAuditEventMessageProcessor" class="com.icw.ehf.integration.atna.AtnaAuditEventMessageProcessor" >
<property name="iheAuditor" ref="iheAuditor" />
</bean>
```

This AtnaAuditEventMessageProcessor simply invokes the IHEAuditor's audit method for all ATNA audit messages crossing his way.

By now, all building blocks of the Camel route are in place. What is left is the actual route definition. For doing this in Java, create a Camel RouteBuilder and overwrite the configure method. This could look like the following:

```
public void configure() throws Exception {
    from(DIRECT_ATNA_TRANSFORM)
        .processRef("atnaTransmogrifierAdapterProcessor")
        .choice()
        .when(body().isNotNull()).to(DIRECT_ATNA)
        .when(body().isNull()).stop();

    from(DIRECT_ATNA)
        .processRef("atnaAuditEventMessageProcessor");
}
```

Additional configuration:

1. The NetworkAccessPointIDTransmogrifier mentioned above, uses the IP address of the current HTTP request. In order to obtain this IP address, an additional ServletFilter has to be configured:

```
<filter>
<filter-name>ipExtractionFilter</filter-name>
<filter-class>com.icw.ehf.commons.web.filter.IpExtractionFilter</filter-class>
</filter>

<filter-mapping>
<filter-name>ipExtractionFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

The IpExtractionFilter itself relies on the StateHolder having been initialized beforehand. The most convenient way to do this is by additionally configuring the RequestContextFilter:

```
<filter>
  <filter-name>requestContextFilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
</filter>

<filter-mapping>
  <filter-name>requestContextFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```



Note: The `requestContextFilter` has to precede the `ipExtractionFilter` in the `web.xml`.

-
2. Similarly, the `SourceIdTransmogrifier` depends on a `ProductInstanceIdentifier` bean representing the application instance's identification.

As an alternative, it is possible to carry out the above setup using IPF and Groovy. However, this option is not covered in this documentation.

Glossary

Code Coverage

Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing. Those tests follow the program graph and are part of the structure-oriented test methods. They are focused on line and branch coverage.

eHealth

In the informatics domain e-health is a philosophy which empowers electronic-health care consumers by bringing information, products and services online (McGraw-Hill Concise Dictionary of Modern Medicine).

Framework

In software development a *framework* can be described as a collection of classes — or rather like a class library — offering abstract methods which then can be implemented by a developer. As a whole a *framework* can be seen as a skeleton of an application which can be customized. It is also an extensible structure for describing a set of concepts, methods and technologies.

eHealth Framework (eHF)

The eHealth Framework (eHF) is a holistic approach for developing electronic health care solutions. It provides tools, interfaces, software modules, and documentation assisting you throughout the entire software development and product life cycle.

References

A

Apache Commons Logging

<http://commons.apache.org/logging/> ▾

Apache Camel

<http://camel.apache.org/> ▾

Apache Maven 2 Conventions

<http://maven.apache.org/maven-conventions.html> ▾

Apache Maven - Artifact Naming Conventions

<http://maven.apache.org/guides/mini/guide-naming-conventions.html> ▾

Apache Maven - Structuring Modules

<http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html> ▾

Apache Portable Runtime

<http://apr.apache.org/versioning.html> ▾

AspectJ

[AspectJ](#) ▾

C

Continuous Integration

<http://martinfowler.com/articles/continuousIntegration.html> ▾

ClamAV

<http://www.clamav.net/> ▾

D

Domain Driven Design

<http://www.domaindrivendesign.org> ▾

Evans, E.

Domain-driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley (2003)

DIMDI

<http://www.dimdi.de> ▾

E

Eicar (Testfile)

[Eicar test signatures](#) ▾

H

Health Level 7

<http://www.hl7.org> ▾

Hibernate

<http://www.hibernate.org> ▾

Hibernate Tools for Eclipse and Ant

<http://www.hibernate.org/255.html> ▾

Java Persistence with Hibernate

<http://www.hibernate.org/397.html> ▾

Hudson

<https://hudson.dev.java.net/> ▾

I

IHE

<http://www.ihe.net> ▾

Open eHealth Foundation Integration Platform (IPF)

<http://www.openehealth.org/projects/ipf> ▾

ISO

<http://www.iso.org> ▾

ISEC-Partners - Cross Site Reference Forgery

http://www.isecpartners.com/documents/XSRF_Paper.pdf ▾

J

Java Authentication and Authorization Service

<http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html>



JAAS Login Configuration File

[here](#) ▾

Java Conventions

<http://java.sun.com/docs/codeconv/> ▾

Java Module System

<http://jcp.org/en/jsr/detail?id=277> ▾

Java Servlet API specification

<http://java.sun.com/products/servlet/reference/api/index.html> ▾

O

OID

<http://www.oid-info.com> ▾

Open eHealth

<http://www.openehealth.org> ▾

S

Security Annotation Framework (SAF)

<http://safr.sourceforge.net> ▾

Spring Framework - Transaction Management

<http://static.springframework.org/spring/docs/2.5.x/reference/transaction.html> ▾

W

W3C - WSDL 1.1 Specification

<http://www.w3.org/TR/wsdl> ▾

W3C - Extensible Style Sheet Language

<http://www.w3.org/TR/xslt20/> ▾

WHO

<http://www.who.int> ▾

WHO - International Classification of Diseases (ICD)

<http://www.who.int/classifications/icd/en/> ▾

List of Abbreviations

Abbreviation	Definition
AJDT	AspectJ Development Tools
AJP	Apache JServ Protocol
API	Application Programming Interface
CVS	Concurrent Version System
DAO	Data Access Object
DTO	Data Transfer Object
EDA	Event Driven Architecture
EHC	Electronic Health Card
eHF	eHealth Framework
GUI	Graphical User Interface
HPC	Health Professional Card
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
ICW	InterComponentWare
IDE	Integrated Development Environment
IPF	Integration Platform
ESB	Enterprise Service Bus
HL7	Health Level 7
IHE	Integrating the Health Care Enterprise
WHO	World Health Organization
ISO	International Standards Organization
ICD	International Classification of Diseases
OID	Object Identifier