

DHIS 2 Technical Architecture

Lars Helge Overland

Version 2.0.1

1. Overview

This document outlines the technical architecture for the District Health Information Software 2 (DHIS 2). The DHIS 2 is a routine data based health information system which allows for data capture, aggregation, analysis, and reporting of data.

DHIS 2 is written in Java and has a three-layer architecture. The presentation layer is web-based, and the system can be used online as well as stand-alone.

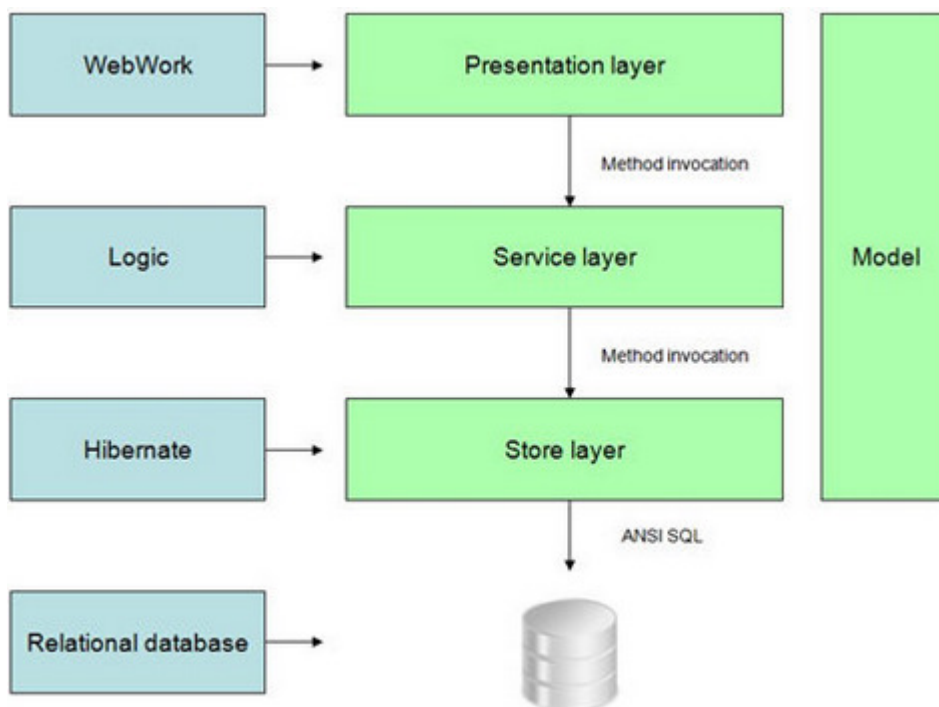


Fig. Overall architecture

2. Technical Requirements

The DHIS 2 is intended to be installed and run on a variety of platforms. Hence the system is designed for industry standards regarding database management systems and application servers. The system should be extensible and modular in order to allow for third-party and peripheral development efforts. Hence a pluggable architecture is needed. The technical requirements are:

- Ability to run on any major database management system
- Ability to run on any J2EE compatible servlet container
- Extensibility and modularity in order to address local functional requirements
- Ability to run online/on the web
- Flexible data model to allow for a variety of data capture requirements

3. Project Structure

DHIS 2 is made up of 42 Maven projects, out of which 18 are web modules. The root POM is located in /dhis-2 and contains project aggregation for all projects excluding the /dhis-2/dhis-web folder. The /dhis-2/dhis-web folder has a web root POM which contains project aggregation for all projects within that folder. The contents of the modules are described later on.

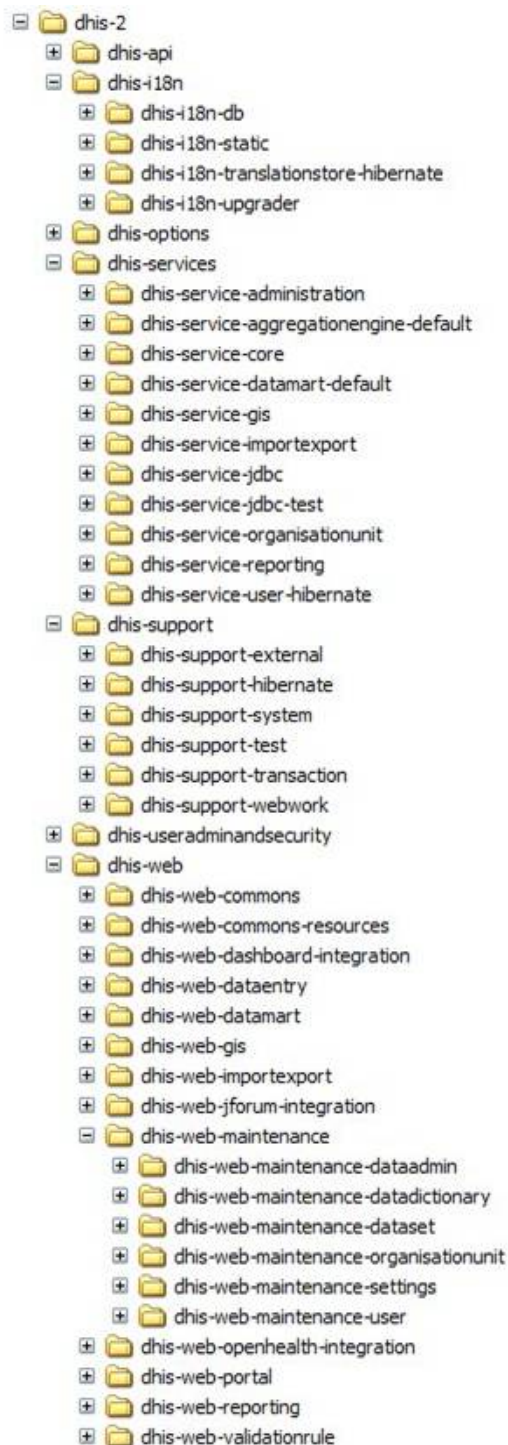


Fig. Project structure

4. The Data Model

The data model is flexible in all dimensions in order to allow for capture of any item of data. The model is based on the notion of a `DataValue`. A `DataValue` can be captured for any `DataElement` (which represents the captured item, occurrence or phenomena), `Period` (which represents the time dimension), and `Source` (which represents the space dimension, i.e. an organisational unit in a hierarchy).

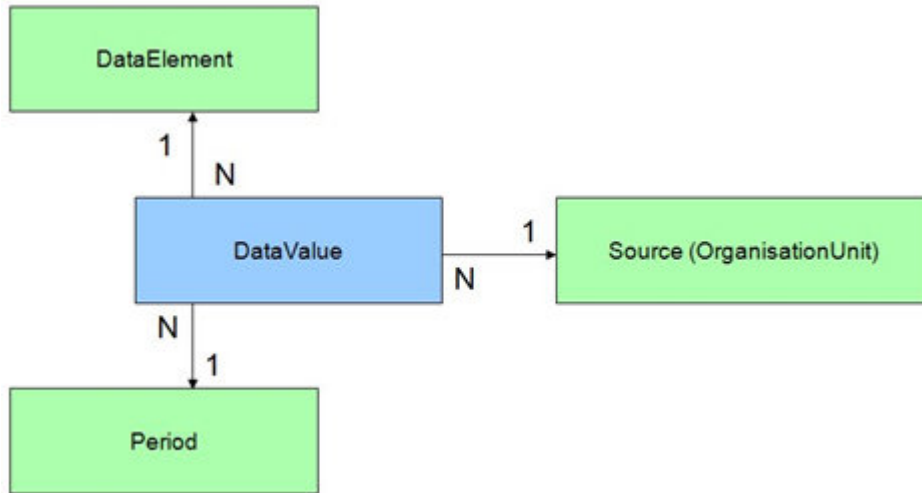


Fig. DataValue diagram

A central concept for data capture is the DataSet. The DataSet is a collection of DataElements for which there is entered data presented as a list, a grid and a custom designed form. A DataSet is associated with a PeriodType, which represents the frequency of data capture.

A central concept for data analysis and reporting is the Indicator. An Indicator is basically a mathematical formula consisting of DataElements and numbers. An Indicator is associated with an IndicatorType, which indicates the factor of which the output should be multiplied with. A typical IndicatorType is percentage, which means the output should be multiplied by 100. The formula is split into a numerator and denominator.

Most objects have corresponding group objects, which are intended to improve and enhance data analysis. The data model source code can be found in the API project and could be explored in entirety there. A selection of the most important objects can be view in the diagram below.

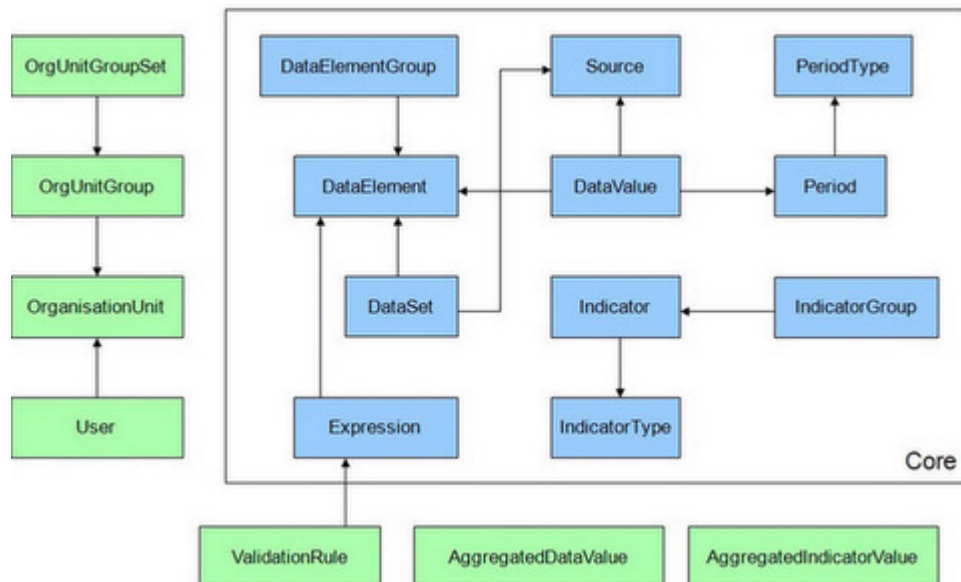


Fig. Core diagram

5. The Persistence Layer

The persistence layer is based on Hibernate in order to achieve the ability to run on any major DBMS. Hibernate abstracts the underlying DBMS away and let you define the database connection properties in a file called hibernate.properties.

DHIS 2 uses Spring-Hibernate integration, and retrieves a SessionFactory through Spring’s LocalSessionFactoryBean. This LocalSessionFactoryBean is injected with a custom HibernateConfigurationProvider instance which fetches Hibernate

mapping files from all modules currently on the classpath. All store implementations get injected with a `SessionFactory` and use this to perform persistence operations.

Most important objects have their corresponding Hibernate store implementation. A store provides methods for CRUD operations and queries for that object, e.g. `HibernateDataElementStore` which offers methods such as `addDataElement(DataElement)`, `deleteDataElement(DataElement)`, `getDataElementByName(String)`, etc.

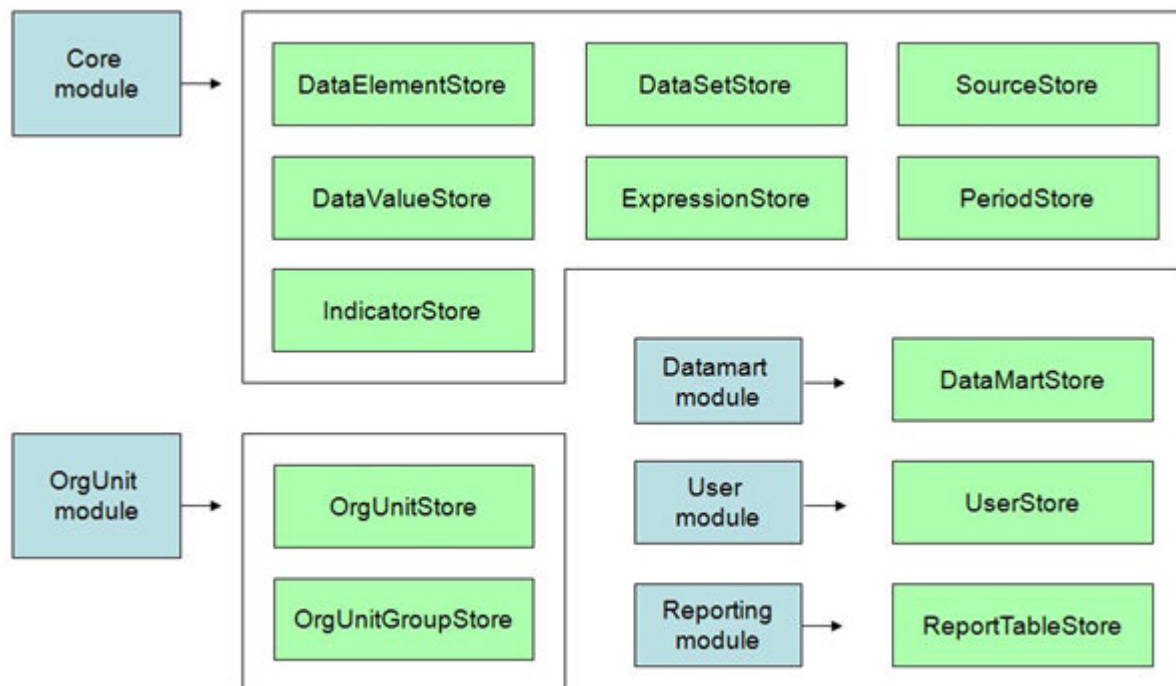


Fig. Persistence layer

6. The Business Layer

All major classes, like those responsible for persistence, business logic, and presentation, are mapped as Spring managed beans. “Bean” is Spring terminology and simply refers to a class that is instantiated, assembled, and otherwise managed by the Spring IoC container. Dependencies between beans are injected by the IoC container, which allows for loose coupling, re-configuration and testability. For documentation on Spring, please refer to springframework.org.

The services found in the `dhis-service-core` project basically provide methods that delegate to a corresponding method in the persistence layer, or contain simple and self-explanatory logic. Some services, like the ones found in the `dhis-service-datamart`, `dhis-service-import-export`, `dhis-service-jdbc`, and `dhis-service-reporting` projects are more complex and will be elaborated in the following sections.

6.1. The JDBC Service Project

The JDBC service project contains a set of components dealing with JDBC connections and SQL statements.

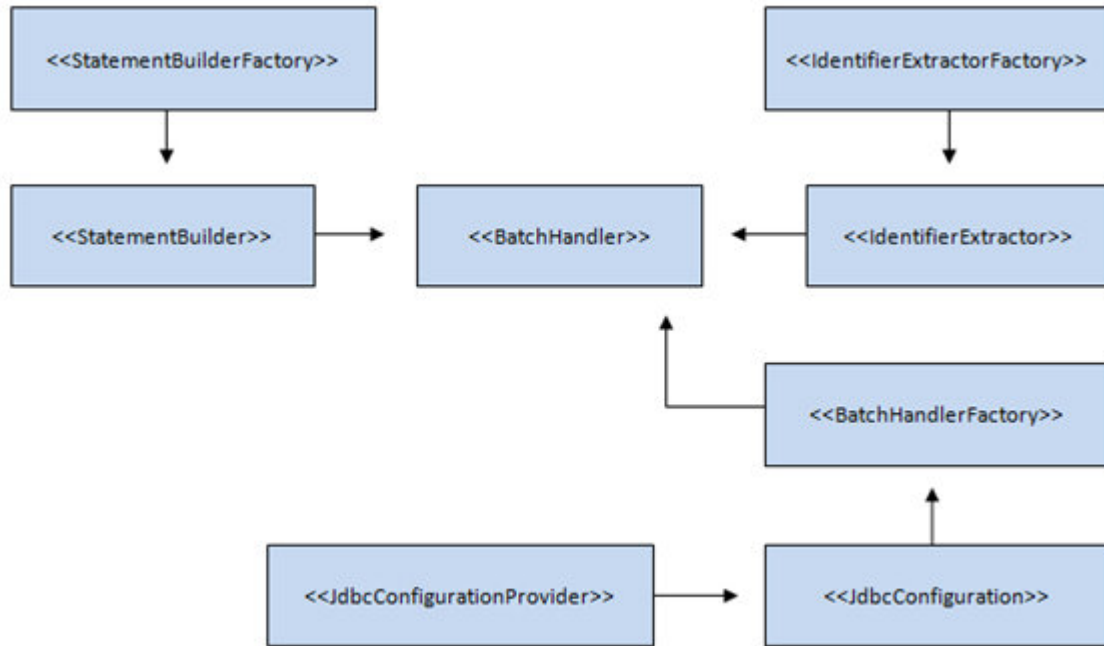


Fig. JDBC BatchHandler diagram

The *BatchHandler* interface provides methods for inserting, updating and verifying the existence of objects. The purpose is to provide high-performance operations and is relevant for large amounts of data. The *BatchHandler* object inserts objects using the *multiple insert SQL* syntax behind the scenes and can insert thousands of objects on each database commit. A typical use-case is an import process where a class using the *BatchHandler* interface will call the *addObject(Object, bool)* method for every import object. The *BatchHandler* will after an appropriate number of added objects commit to the database transparently. A *BatchHandler* can be obtained from the *BatchHandlerFactory* component. *BatchHandler* implementations exist for most objects in the API.

The *JdbcConfiguration* interface holds information about the current DBMS JDBC configuration, more specifically dialect, driver class, connection URL, username and password. A *JdbcConfiguration* object is obtained from the *JdbcConfigurationProvider* component, which currently uses the internal Hibernate configuration provider to derive the information.

The *StatementBuilder* interface provides methods that represents SQL statements. A *StatementBuilder* object is obtained from the *StatementBuilderFactory*, which is able to determine the current runtime DBMS and provide an appropriate implementation. Currently implementations exist for PostgreSQL, MySQL, H2, and Derby.

The *IdentifierExtractor* interface provides methods for retrieving the last generated identifiers from the DBMS. An *IdentifierExtractor* is obtained from the *IdentifierExtractorFactory*, which is able to determine the runtime DBMS and provide an appropriate implementation.

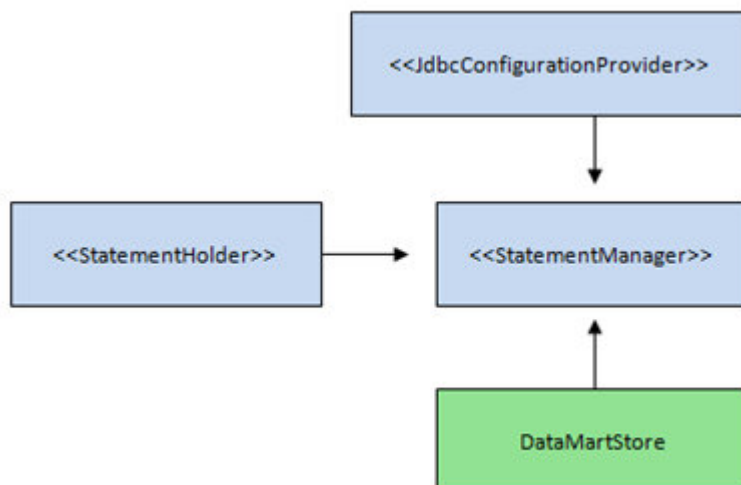


Fig. JDBC StatementManager diagram

The *StatementHolder* interface holds and provides JDBC connections and statements. A *StatementHolder* object can be obtained from the *StatementManager* component. The *StatementManager* can be initialized using the *initialise()* method closed using the *destroy()* method. When initialized, the *StatementManager* will open a database connection and hold it in a *ThreadLocal* variable, implying that all subsequent requests for a *StatementHolder* will return the same instance. This can be used to improve performance since a database connection or statement can be reused for multiple operations. The *StatementManager* is typically used in the persistence layer for classes working directly with JDBC, like the *DataMartStore*.

6.2. The Import-Export Project

The import-export project contains classes responsible for producing and consuming interchange format files. The import process has three variants which are import, preview and analysis. Import will import data directly to the database, preview will import to a temporary location, let the user do filtering and eventually import, while the analysis will reveal abnormalities in the import data. Currently supported formats are:

- DXF (DHIS eXchange Format)
- IXF (Indicator eXchange Format)
- DHIS 1.4 XML format
- DHIS 1.4 Datafile format
- CSV (Comma Separated Values)
- PDF (Portable Document Format)

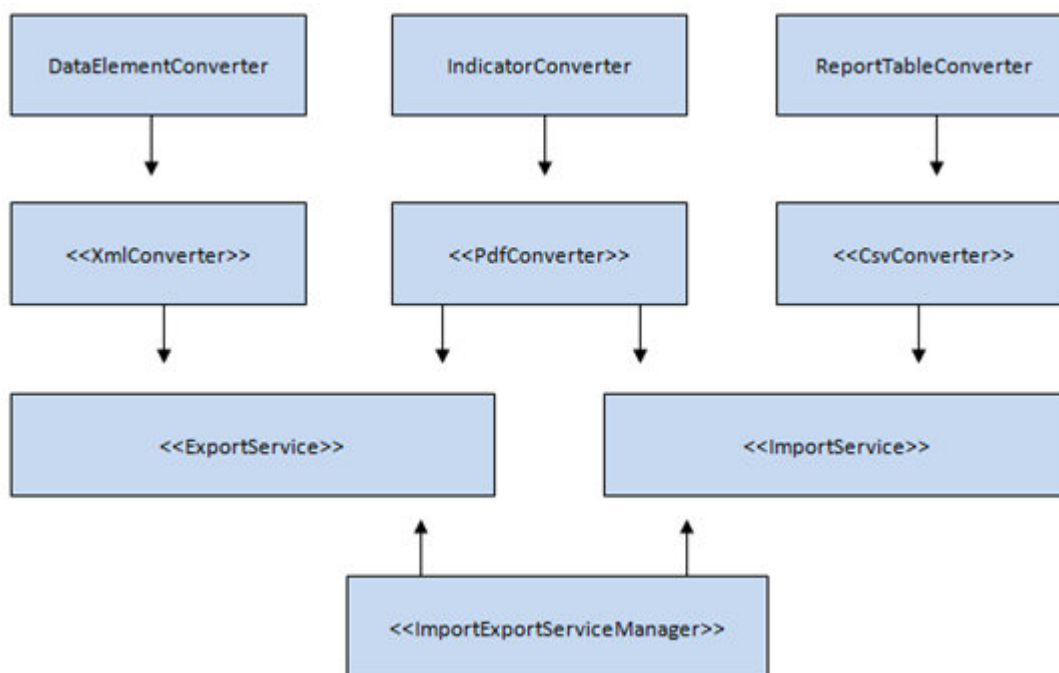


Fig. Import-export service diagram

The low-level components doing the actual reading and writing of interchange format files are the converter classes. The most widely used is the *XmlConverter* interface, which provides a *write(XmlWriter, ExportParams)* and a *read(XmlReader, ImportParams)* method. Most objects in the API have corresponding *XmlConverter* implementations for the DXF format. Writing and reading for each object is delegated to its corresponding *XmlConverter* implementation.

The *ExportParams* object is a specification which holds the identifiers of the objects to be exported. The converter retrieves the corresponding objects and writes content to the *XmlWriter*. *XmlConverter* implementations for the DXF format exist for most objects in the API. For instance, the *write* method of class *DataElementConverter* will write data that represents *DataElements* in DXF XML syntax to the *XmlWriter*.

The *ExportService* interface exposes a method *InputStream exportData(ExportParams)*. The *ExportService* is responsible for instantiating the appropriate converters and invoke their export methods. To avoid long requests prone to timeout-errors in the presentation layer, the actual export work happens in a separate thread. The *ExportService* registers its converters on the *ExportThread* class using its *registerXmlConverter(XmlConverter)* method, and then starts the thread.

The *ImportParams* object contains directives for the import process, like type and strategy. For instance, the read method of class *DataElementConverter* will read data from the *XmlReader*, construct objects from the data and potentially insert it into the database, according to the directives in the *ImportParams* object.

The *ImportService* interface exposes a method *importData(ImportParams, InputStream)*. The *ImportService* is responsible for instantiating the appropriate converters and invoke their import methods. The import process is using the *BatchHandler* interface heavily.

The *ImportExportServiceManager* interface provides methods for retrieving all *ImportServices* and *ExportServices*, as well as retrieving a specific *ImportService* or *ExportService* based on a format key. This makes it simple to retrieve the correct service from using classes since the name of the format can be used as parameter in order to get an instance of the corresponding service. This is implemented with a Map as the backing structure where the key is the format and the value is the *Import-* or *ExportService* reference. This map is defined in the Spring configuration, and delegates to Spring to instantiate and populate the map. This allows for extensibility as developing a new import service is simply a matter of providing an implementing the *ImportService* interface and add it to the map definition in the Spring configuration, without touching the *ImportExportServiceManager* code.

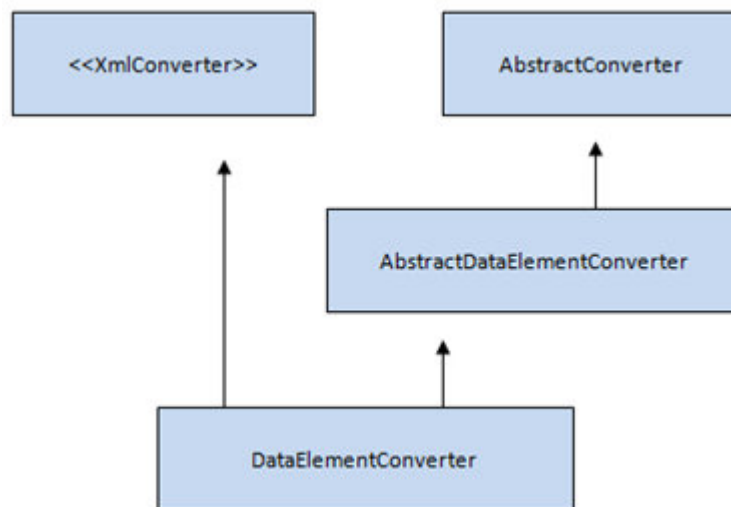


Fig. Import-export converter diagram

Functionality that is general for converters of all formats is centralized in abstract converter classes. The *AbstractConverter* class provides four abstract methods which must be implemented by using converters, which are *importUnique(Object)*, *importMatching(Object, Object)*, *Object getMatching()* and *boolean isIdentical(Object, Object)*. It also provides a *read(Object, GroupMemberType, ImportParams)* method that should be invoked by all converters at the end of the read process for every object. This method utilizes the mentioned abstract methods and dispatches the object to the analysis, preview or import routines depending on the state of the object and the current import directives. This allows for extensibility as converters for new formats can extend their corresponding abstract converter class and reuse this functionality.

6.3. The Data Mart Project

The data mart component is responsible for producing aggregated data from the raw data in the time and space dimension. The aggregated data is represented by the *AggregatedDataValue* and *AggregatedIndicatorValue* objects. The *DataSetCompletenessResult* object is also included in the data mart and is discussed in the section covering the reporting project. These objects and their corresponding database tables are referred to as the *data mart*.

The following section will list the rules for aggregation in DHIS 2.

- Data is aggregated in the time and space dimension. The time dimension is represented by the *Period* object and the space dimension by the *OrganisationUnit* object, organised in a parent-child hierarchy.
- Data registered for all periods which intersects with the aggregation start and end date is included in the aggregation process. Data for periods which are not fully within the aggregation start and end date is weighed according to a factor “number of days within aggregation period / total number of days in period”.
- Data registered for all children of the aggregation *OrganisationUnit* is included in the aggregation process.

- Data registered for a data element is aggregated based on the aggregation operator and data type of the data element. The aggregation operator can be *sum* (values are summarized), *average* (values are averaged) and *count* (values are counted). The data type can be *string* (text), *int* (number), and *bool* (true or false). Data of type *string* can not be aggregated.
 - Aggregated data of type *sum – int* is presented as the summarized value.
 - Aggregated data of type *sum – bool* is presented as the number of true registrations.
 - Aggregated data of type *average – int* is presented as the averaged value.
 - Aggregated data of type *average – bool* is presented as a percentage value of true registrations in proportion to the total number of registrations.
- An indicator represents a formula based on data elements. Only data elements with aggregation operator *sum* or *average* and with data type *int* can be used in indicators. Firstly, data is aggregated for the data elements included in the indicator. Finally, the indicator formula is calculated.
- A calculated data element represents a formula based on data elements. The difference from indicator is that the formula is on the form “data element * factor”. The aggregation rules for indicator apply here as well.

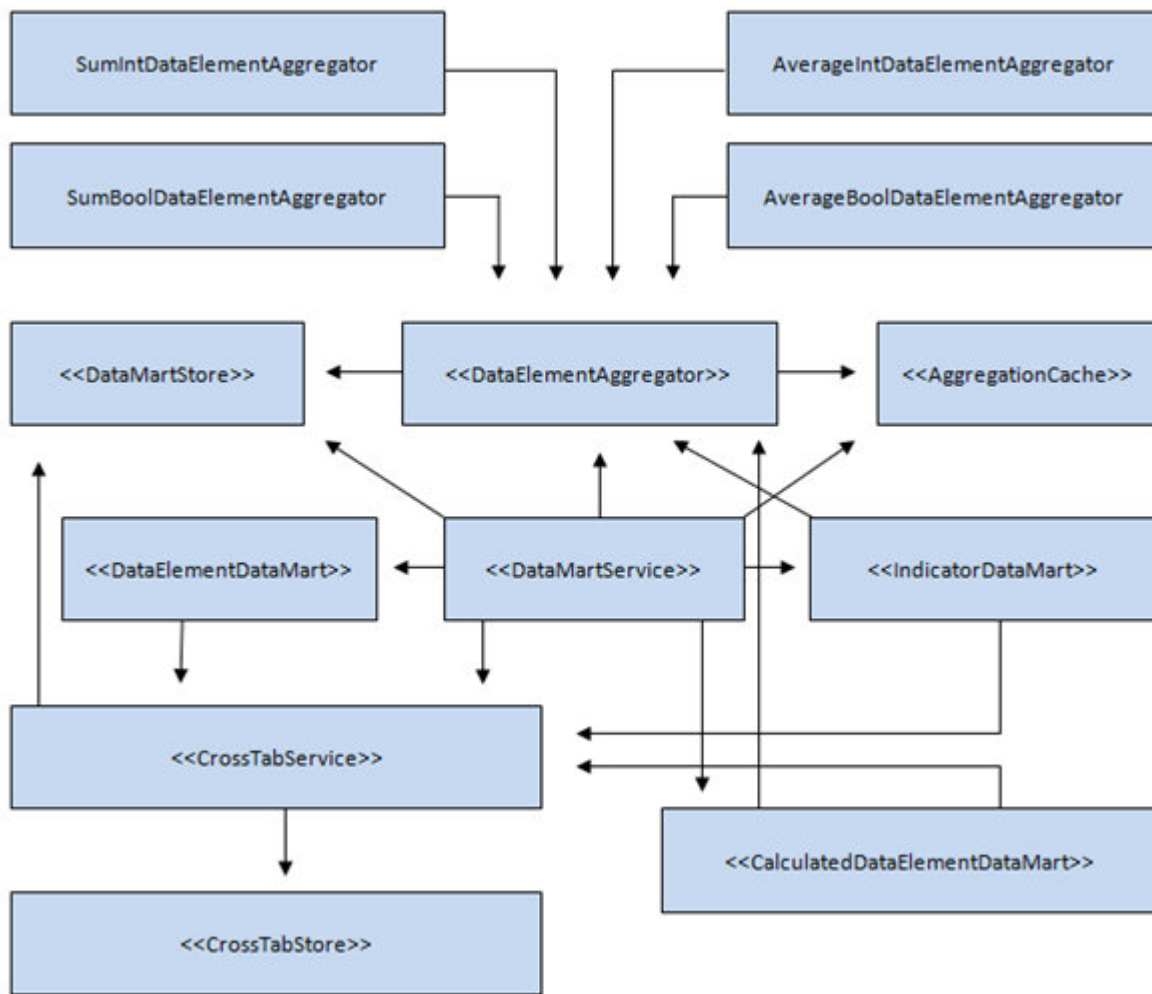


Fig. Data mart diagram

The *AggregationCache* component provides caching in *ThreadLocal* variables. This caching layer is introduced to get optimal caching [9]. The most frequently used method calls in the data mart component is represented here.

The *DataElementAggregator* interface is responsible for retrieving data from the crosstabulated temporary storage and aggregate data in the time and space dimension. This happens according to the combination of data element aggregation operator and data type the class represents. One implementation exist for each of the four variants of valid combinations, namely *SumIntDataElementAggregator*, *SumBoolDataElementAggregator*, *AverageIntDataElementAggregator* and *AverageBoolAggregator*.

The *DataElementDataMart* component utilizes a *DataElementAggregator* and is responsible for writing aggregated data element data to the data mart for a given set of data elements, periods, and organisation units.

The *IndicatorDataMart* component utilizes a set of *DataElementAggregators* and is responsible for writing aggregated indicator data to the data mart for a given set of indicators, periods, and organisation units.

The *CalculatedDataElementDataMart* component utilizes a set of *DataElementAggregators* and is responsible for writing aggregated data element data to the data mart for a given set of calculated data elements, periods, and organisation units.

The *DataMartStore* is responsible for retrieving aggregated data element and indicator data, and data from the temporary crosstabulated storage.

The *CrossTabStore* is responsible for creating, modifying and dropping the temporary crosstabulated table. The *CrossTabService* is responsible for populating the temporary crosstabulated table. This table is used in an intermediate step in the aggregation process. The raw data is de-normalized on the data element dimension, in other words the crosstabulated table gets one column for each data element. This step implies improved performance since the aggregation process can be executed against a table with a reduced number of rows compared to the raw data table.

The *DataMartService* is the central component in the data mart project and controls the aggregation process. The order of operations is:

- Existing aggregated data for the selected parameters is deleted.
- The temporary crosstabulated table is created and populated using the *CrossTabService* component.
- Data element data for the previously mentioned valid variants is exported to the data mart using the *DataElementDataMart* component.
- Indicator data is exported to the data mart using the *IndicatorDataMart* component.
- Calculated data element data is exported to the data mart using the *CalculatedDataElementDataMart* component.
- The temporary crosstabulated table is removed.

The data element tables are called “aggregateddatavalue” and “aggregatedindicatorvalue” and are used both inside DHIS 2 for e.g. report tables and by third-party reporting applications like MS Excel.

6.4. The Reporting Project

The reporting project contains components related to reporting, which will be described in the following sections.

6.4.1. Report table

The *ReportTable* object represents a crosstabulated database table. The table can be crosstabulated on any number of its three dimensions, which are the descriptive dimension (which can hold data elements, indicators, or data set completeness), *period* dimension, and *organisation unit* dimension. The purpose is to be able to customize tables for later use either in third-party reporting tools like BIRT or directly in output formats like PDF or HTML inside the system. Most of the logic related to crosstabulation is located in the *ReportTable* object. A *ReportTable* can hold:

- Any number of data elements, indicators, data sets, periods, and organisation units.
- A *RelativePeriods* object, which holds 10 variants of relative periods. Examples of such periods are *last 3 months*, *so far this year*, and *last 3 to 6 months*. These periods are relative to the reporting month. The purpose of this is to make the report table re-usable in time, i.e. avoid the need for the user to replace periods in the report table as time goes by.
- A *ReportParams* object, which holds report table parameters for reporting month, parent organisation unit, and current organisation unit. The purpose is to make the report table re-usable across the organisation unit hierarchy and in time, i.e. make it possible for the user to re-use the report table across organisation units and as time goes by.
- User options such as regression lines. Value series which represents regression values can be included when the report table is crosstabulated on the period dimension.

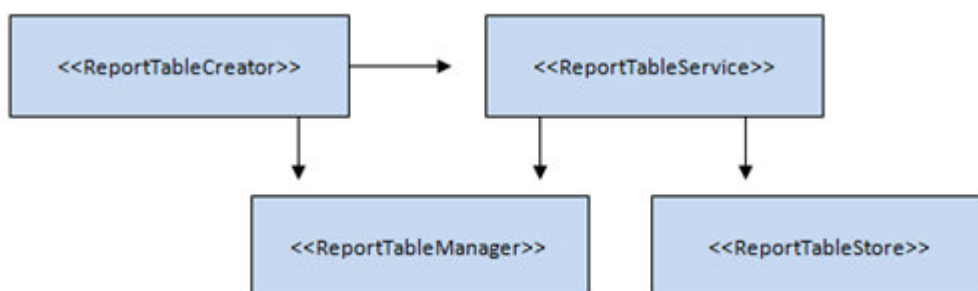


Fig. Report table diagram

The *ReportTableStore* is responsible for persisting *ReportTable* objects, and currently has a Hibernate implementation.

The *ReportTableService* is responsible for performing business logic related to report tables such as generation of relative periods, as well as delegating CRUD operations to the *ReportTableStore*.

The *ReportTableManager* is responsible for creating and removing report tables, as well as retrieving data.

The *ReportTableCreator* is the key component, and is responsible for:

- Exporting relevant data to the data mart using the *DataMartExportService* or the *DataSetCompletenessService*. Data will later be retrieved from here and used to populate the report table.
- Create the report table using the *ReportTableManager*.
- Include potential regression values.
- Populate the report table using a *BatchHandler*.
- Remove the report table using the *ReportTableManager*.

6.4.2. Chart

The *Chart* object represents preferences for charts. Charts are either *period based* or *organisation unit based*. A chart has three dimensions, namely the *value*, *category*, and *filter* dimension. The value dimension contains any numbers of indicators. In the period based chart, the category dimension contains any number of periods while the filter dimension contains a single organisation unit. In the organisation unit based chart, the category dimension contains any number of organisation units while the filter dimension contains a single period. Two types of charts are available, namely bar charts and line charts. Charts are materialized using the JFreeChart library. The bar charts are rendered with a *BarRenderer* [2], the line charts with a *LineAndShapeRenderer* [2], while the data source for both variants is a *DefaultCategoryDataSet* [3]. The *ChartService* is responsible for CRUD operations, while the *ChartService* is responsible for creating *JfreeCharts* instances based on a *Chart* object.

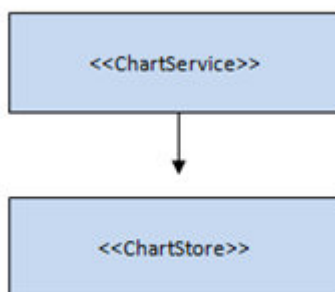


Fig. Chart diagram

6.4.3. Data set completeness

The purpose of the data set completeness functionality is to record the number of data sets that have been completed. The definition of when a data set is complete is subjective and based on a function in the data entry screen where the user can mark the current data set as complete. This functionality provides a percentage completeness value based on the number of reporting organisation units with completed data sets compared to the total number of reporting organisation units for a given data set. This functionality also provides the number of completed data sets reported *on-time*, more specifically reported before a defined number of days after the end of the reporting period. This date is configurable.

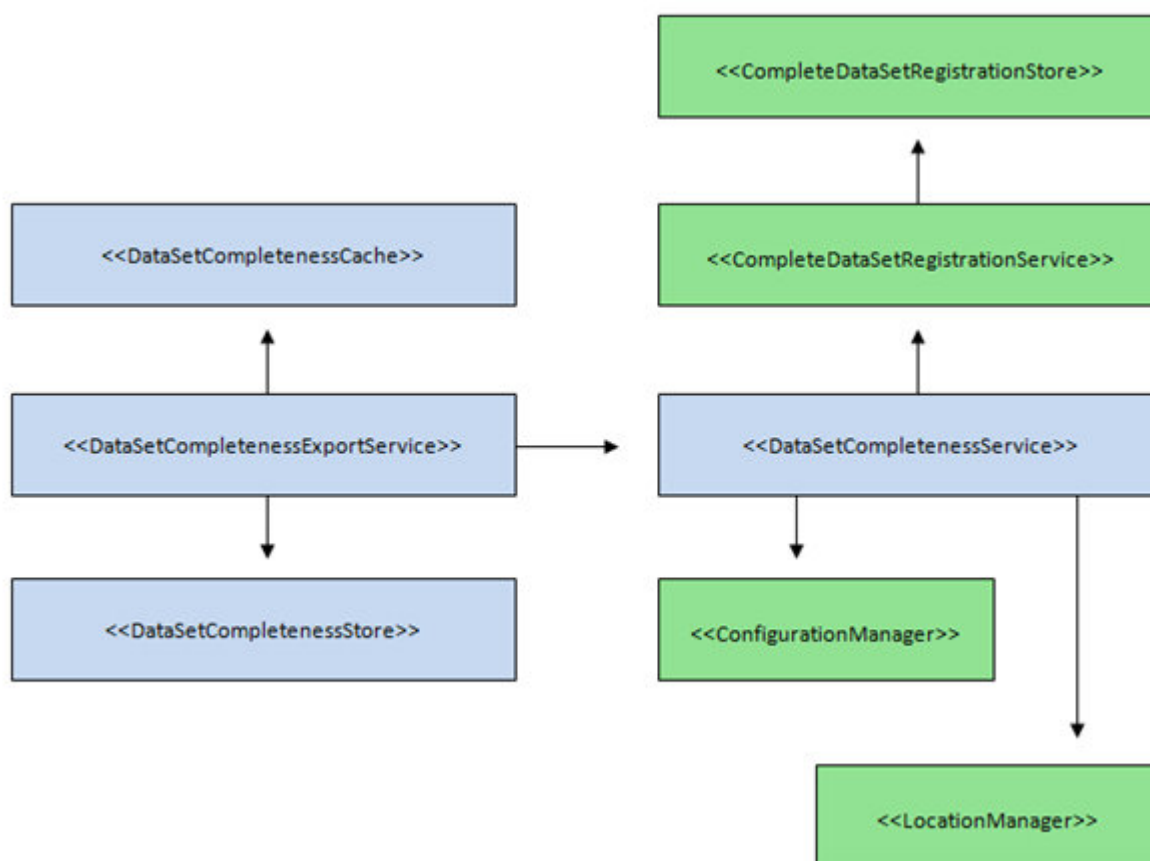


Fig. Data set completeness diagram

The *CompleteDataSetRegistration* object is representing a data set marked as complete by a user. This property holds the data set, period, organisation unit and date for when the complete registrations took place. The *CompleteDataSetRegistrationStore* is responsible for persistence of *CompleteDataSetRegistration* objects and provides methods returning collections of objects queried with different variants of data sets, periods, and organisation units as input parameters. The *CompleteDataSetRegistrationService* is mainly delegating method calls the store layer. These components are located in the *dhis-service-core* project.

The completeness output is represented by the *DataSetCompletenessResult* object. This object holds information about the request that produced it such as data set, period, organisation unit, and information about the data set completeness situation such as number of reporting organisation units, number of complete registrations, and number of complete registrations on-time. The *DataSetCompletenessService* is responsible for the business logic related to data set completeness reporting. It provides methods which mainly returns collections of *DataSetCompletenessResults* and takes different variants of period, organisation unit and data set as parameters. It uses the *CompleteDataSetRegistrationService* to retrieve the number of registrations, the *DataSetService* to retrieve the number of reporting organisation units, and performs calculations to derive the completeness percentage based on these retrieved numbers.

The *DataSetCompletenessExportService* is responsible for writing *DataSetCompletenessResults* to a database table called “*aggregateddatasetcompleteness*”. This functionality is considered to be part of the data mart as this data can be used both inside DHIS 2 for e.g. report tables and by third-party reporting applications like MS Excel. This component is retrieving data set completeness information from the *DataSetCompletenessService* and is using the *BatchHandler* interface to write such data to the database.

6.4.4. Document

The *Document* object represents either a *document* which is uploaded to the system or a *URL*. The *DocumentStore* is responsible for persisting *Document* objects, while the *DocumentService* is responsible for business logic.

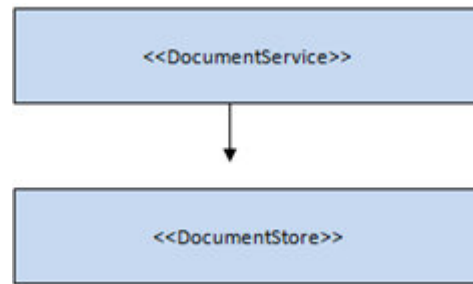


Fig. Document diagram

6.4.5. Pivot table

The *PivotTable* object represents a pivot table. It can hold any number of indicators, periods, organisation units, and corresponding aggregated indicator values. It offers basic pivot functionality like pivoting and filtering the table on all dimensions. The business logic related to pivot tables is implemented in Javascript and is located in the presentation layer. The *PivotTableService* is responsible for creating and populating *PivotTable* objects.

6.4.6. The External Project

The *LocationManager* component is responsible for the communication between DHIS 2 and the file system of the operating system. It contains methods which provide read access to files through *File* and *InputStream* instances, and write access to the file system through *File* and *OutputStream* instances. The target location is relative to a system property “dhis2.home” and an environment variable “DHIS2_HOME” in that order. This component is used e.g. by the *HibernateConfigurationProvider* to read in the Hibernate configuration file, and should be re-used by all new development efforts.

The *ConfigurationManager* is a component which facilitates the use of configuration files for different purposes in DHIS 2. It provides methods for writing and reading configuration objects to and from XML. The *XStream* library is used to implement this functionality. This component is typically used in conjunction with the *LocationManager*.

7. The Presentation Layer

The presentation layer of DHIS 2 is based on web modules which are assembled into a portal. This implies a modularized design where each module has its own domain, e.g. the *dhis-web-reporting* module deals with reports, charts, pivot tables, documents, while the *dhis-web-maintenance-dataset* module is responsible for data set management. The web modules are based on *WebWork* and follow the MVC pattern [5]. The modules also follow the Maven standard for directory layout, which implies that Java classes are located in *src/main/java*, configuration files and other resources in *src/main/resources*, and templates and other web resources in *src/main/webapp*. All modules can be run as a standalone application.

Common Java classes, configuration files, and property files are located in the *dhis-web-commons* project, which is packaged as a JAR file. Common templates, style sheets and other web resources are located in the *dhis-web-commons-resources* project, which is packaged as a WAR file. These are closely related but are separated into two projects. The reason for this is that other modules must be able to have compile dependencies on the common Java code, which requires it to be packaged as a JAR file. For other modules to be able to access the common web resources, these must be packaged as a WAR file [6].

7.1. The Portal

DHIS 2 uses a light-weight portal construct to assemble all web modules into one application. The portal functionality is located in the *dhis-web-portal* project. The portal solution is integrated with *WebWork*, and the following section requires some prior knowledge about this framework, please refer to opensymphony.com/webwork for more information.

7.1.1. Module Assembly

All web modules are packaged as WAR files. The portal uses the Maven WAR plug-in to assemble the common web modules and all web modules into a single WAR file. Which modules are included in the portal can be controlled simply through the dependency section in the POM file [7] in the *dhis-web-portal* project. The web module WAR files will be extracted and its content merged together.

7.1.2. Portal Module Requirements

The portal requires the web modules to adhere to a few principles:

- The web resources must be located in a folder `src/main/webapp/<module-artifact-id >`.
- The `xwork.xml` configuration file must extend the `dhis-web-commons.xml` configuration file.
- The action definitions in `xwork.xml` for a module must be in a package where the name is `<module-artifact-id>`, namespace is `/<module-artifact-id>`, and which extends the `dhis-web-commons` package.
- All modules must define a default action called `index`.
- The `web.xml` of the module must define a redirect filter, open-session-in-view filter, security filter, and the `WebWork FilterDispatcher` [8].
- All modules must have dependencies to the `dhis-web-commons` and `dhis-web-commons-resources` projects.

7.1.3. Common Look-And-Feel

Common look and feel is achieved using a back-bone Velocity template which includes a page template and a menu template defined by individual actions in the web modules. This is done by using static parameters in the `WebWork xwork.xml` configuration file. The action response is mapped to the back-bone template called `main.vm`, while static parameters called `page` and `menu` refers to the templates that should be included. This allows the web modules to display its desired content and left side menu while maintaining a common look-and-feel.

7.1.4. Main Menu

The main menu contains links to each module. Each menu link will redirect to the index action of each module. The menu is updated dynamically according to which web modules are on the classpath. The menu is visibly generated using the `ModuleManager` component, which provides information about which modules are currently included. A module is represented by the `Module` object, which holds properties about the name, package name, and default action name. The `ModuleManager` detects web modules by reading the `WebWork Configuration` and `PackageConfig` objects, and derives the various module names from the name of each package definition. The `Module` objects are loaded onto the `WebWork` value stack by `WebWork` interceptors using the `ModuleManager`. These values are finally used in the back-bone Velocity template to produce the menu mark-up.

8. Framework Stack

The following frameworks are used in the DHIS 2 application.

8.1. Application Frameworks

- Hibernate (www.hibernate.org [<http://www.hibernate.org>])
- Spring (www.springframework.org [<http://www.springframework.org>])
- WebWork (www.opensymphony.com/webwork [<http://www.opensymphony.com/webwork>])
- Velocity (www.velocity.apache.org [<http://www.velocity.apache.org>])
- Commons (www.commons.apache.org [<http://www.commons.apache.org>])
- JfreeChart (www.jfree.org/jfreechart/ [<http://www.jfree.org/jfreechart/>])
- JUnit (www.junit.org [<http://www.junit.org>])

8.2. Development Frameworks

- Maven ([apache.maven.org](http://www.apache.maven.org) [<http://www.apache.maven.org>])
- Bazaar ([bazaar-vcs.org](http://www.bazaar-vcs.org) [<http://www.bazaar-vcs.org>])

9. Definitions

[1] “Classpath” refers to the root of a JAR file, `/WEB-INF/lib` or `/WEB-INF/classes` in a WAR-file and `/src/main/resources` in the source code; locations from where the JVM is able to load classes.

[2] `JFreeChart` class located in the `org.jfree.chart.renderer` package.

[3] `JFreeChart` class located in the `org.jfree.data.category` package.

[4] Operations related to creating, retrieving, updating, and deleting objects.

[5] Model-View-Controller, design pattern for web applications which separates mark-up code from application logic code.

- [6] The WAR-file dependency is a Maven construct and allows projects to access the WAR file contents during runtime.
- [7] Project Object Model, the key configuration file in a Maven 2 project.
- [8] Represents the front controller in the MVC design pattern in WebWork.
- [9] Hibernate second-level cache does not provide satisfactory performance.