CommonDb - an embedded database framework for Scala applications

- CommonDb is a small framework intended to provide common functionality to Scala programs that use an embedded SQL database as a persistence mechanism.

- The framework provides a simple encapsulation of database creation, opening and schema upgrade - handling the details, deferring customisation to your code.
- The interface to your data creation/population/access code and UI is achieved using several callback adapters.
- Databases created with the framework are versioned, and a facility is provided to enable easy upgrade/migration of older databases to the current schema, triggered upon database open.
- A simple Data Access Object style is used for the framework's own tables; you have access to the DataSource and a Spring JDBC SimpleJdbcTemplate with which you can provide your own access code.
- Databases may be optionally encrypted with AES-256. The password is set upon creation, and can be provided on open, or your application can be called back to prompt for it.
- Type-safety is encouraged, by use of Representation Types, and Option.

- It is currently built for Scala 2.10.1.
- It uses Spring-JDBC 3.0.2.RELEASE to make JDBC far easier to work with.
- The framework uses the H2 1.2.128 embedded database engine.
- The API currently throws Spring JDBC's DataAccessException; future releases will wrap these in a scalaz Validation to make composition of failures cleaner.
- Logging is done via the slf4j API. Spring JDBC needs some additional slf4j packages which are listed as dependencies.

- The framework was written using Test-Driven Development: ScalaTest, EasyMock, JUnit.
- Jars are available from the Central Maven repository.
- It is open source, licensed under the Apache 2.0 license.
- Source is available on Google Code hosting.

- This guide is a short introduction to the framework, and should be read in conjunction with the framework's unit tests and sample code.

Release History

v0.1.0 February 2013, initial release for Scala 2.9.2

v0.1.1 September 2013, patch release for Scala 2.10.1. Case classes used for representation types cannot now be inherited; migration test bug fixed.

Getting Started

Maven settings

CommonDb is distributed via the central Maven repository. Dependencies on CommonDb are given by:

```
<dependency>
  <groupId>org.devzendo</groupId>
  <artifactId>CommonDbFramework</artifactId>
  <version>0.1.1</version>
</dependency>
<dependency>
  <groupId>org.devzendo</groupId>
  <artifactId>CommonDbFramework</artifactId>
  <version>0.1.1</version>
  <classifier>tests</classifier>
  <scope>test</scope>
</dependency>
```

Using CommonDb

Database storage

CommonDb stores persistent data in an embedded relational database, using the H2 SQL database engine. All files relating to a single database are stored in their own directory.

H2 was chosen as the persistence engine due to its heritage and maturity, coming from the same developer as the earlier Hypersonic database. It also permits databases to be completely encrypted using the AES-256 encryption algorithm. The user chooses whether the database is to be encrypted at creation time, and will be queried for the password upon subsequently re-opening the database.

Low Level Database Access APIs

A JDBC DataSource is available, for low-level access to the database.

High Level Database Access APIs

The framework itself uses Spring JDBC Templates to simplify access to the database via JDBC. The details of this API are not covered here; consult the Spring documentation. A SimpleJdbcTemplate and TransactionTemplate are available.

This assumes you are using Spring JDBC Templates to access your data using an implementation of the Data Access Object pattern – the essentials of this are covered in this guide, but see Martin Fowler's Patterns of Enterprise Application Architecture ('Mapper' pattern), or Eric Evans' Domain Driven Design ('Repository' pattern) for further details.

The support of Spring JDBC does not preclude the use of Hibernate to provide higher-level access using Object-Relational mapping; this has not yet been explored. Similarly for iBatis SQL maps. If you are interested in pursuing either of these strategies for data access, please contact us!

Database objects maintained by CommonDb

CommonDb databases always contain a record of the application's current code and schema version numbers. This allows databases created by older versions of the software to be automatically updated to the latest schema when they are opened by newer versions of the software. The framework allows you to provide implementations of its callback mechanisms with which you can provide upgrade steps to effect this process.

A sequence is also available, as a Long whose value starts at 0.

The domain layer and the DAO pattern

Although your application's data is stored in a relational database, this does not mean that you scatter SQL access code throughout your code. It is better to separate your application into layers, with a *data access layer* insulating the SQL and underlying relational tables from the rest of the application.

This approach allows the underlying persistence technology to be changed if needed.

Typically, for simple databases, JDBC access using Spring JDBC is sufficient. Spring JDBC makes writing database access code far easier than with raw JDBC. You will need to implement code that performs CRUD (Create, Read, Update, Delete) operations on the database, transforming the data returned by JDBC into objects to be used by the rest of your application. This is the approach taken by the framework's own Data Access Objects, and in the unit test/example code.

For more complex databases, it may be necessary to use an Object-Relational Mapping framework such as Hibernate (which is also made easier with Spring).

The interfaces and data transfer objects you develop as part of the data access layer are not specific to either of these approaches; they insulate you from the actual implementation.

The choice of using Spring JDBC or Spring Hibernate depends on the complexity of your database schema (greater complexity suggests using ORM), whether you intend to issue many batch operations (JDBC) ...

Your implementation of the data access layer works with the JdbcTemplate and DataSource/Connection to implement to the DAO objects/interfaces.

DAO interfaces typically contain:

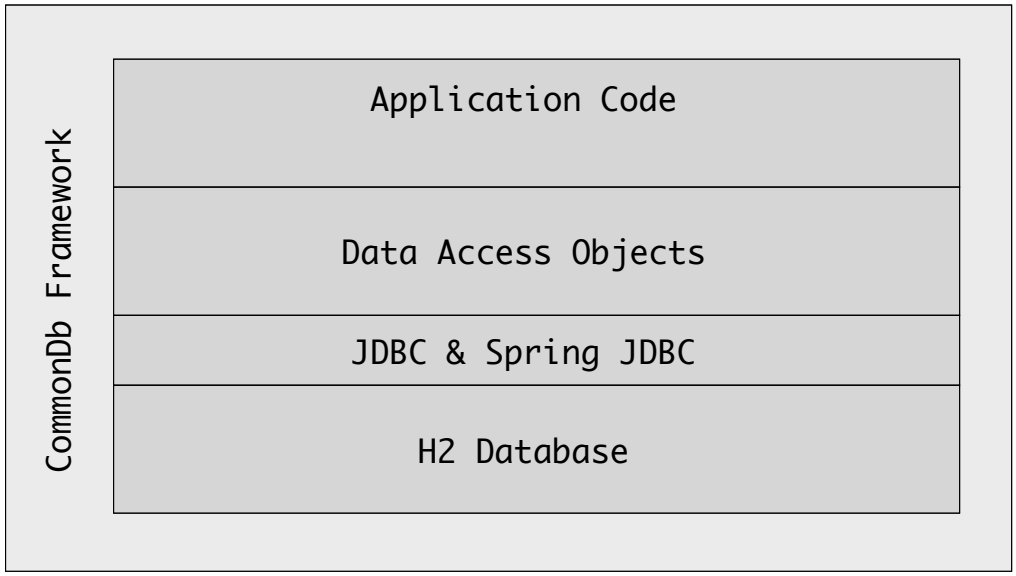
- Finder methods; allow finding of objects by criteria other than by their primary key
- Object graph / aggregate methods
- CRUD – Create, Read (by primary key), Update, Delete methods

They do not typically contain transaction-related code; this is usually implemented at a higher level. A transaction that contains several operations can be created by calling createTransactionTemplate() on the DatabaseAccess instance.

As the DAOs are interfaces to which you provide a concrete implementation, they should also be

easy to implement using mock objects, to allow other layers of the application to be tested without needing a real database. Since H2 is embedded, it is also easy to test against a real database – in some cases, necessary. (All the framework's unit tests and the example code do this.)

The layering looks like this:



The structure of the DAO layer can be either fine-grained (a thin abstraction layer atop the tables and CRUD operations on them), or coarse-grained (persisting entire object graphs; a style more suited to the use of ORM). Each DAO interface is responsible for one domain object; if a domain object has an independent lifecycle, it should have its own DAO interface.

The example here will use the former, fine-grained style. We'll define DAO interfaces that describe the operations for each table in our sample application, and build domain objects (Data Transfer or Value Object pattern) that map onto rows in the tables, and that are the objects that the DAO interfaces work with. We'll build implementations of the DAO interfaces using Spring JDBC templates.

Before introducing the sample application, the CommonDb API needs some introduction.

The CommonDb API

Read the online ScalaDoc, or browse the source in your IDE as you read this!

Creating a database

To create a database, you need a DatabaseAccessFactory:

```
val databaseAccessFactory = DatabaseAccessFactory()
val codeVersion = CodeVersion("1.0")
val schemaVersion = SchemaVersion("0.4")
val directory = new File("/path/to/storage")
val database = databaseAccessFactory.create(directory, "mydatabase",
        None, // Password
        codeVersion, schemaVersion,
        None, // CreateWorkflowAdapter
```

```
        None, // UserDatabaseCreator
        None) // UserDatabaseAccess creation function
// do stuff
database.get.close()
```

Here, database is an instance of DatabaseAccess, which can provide a JDBC DataSource, a Spring-JDBC SimpleJdbcTemplate and a means for creating transactions via a Spring-JDBC TransactionTemplate; it has been populated with the Versions and Sequence tables, for which DAOs are available. The CodeVersion and SchemaVersion given have been stored in the Versions table.

Note that in the above, no user database customisation is provided - this would require an instance of UserDatabaseCreator and a UserDatabaseAccess function.

The CreateWorkflowAdapter can be provided to:

- Inform the user of the progress of the database creation
- To inform the user of any DataAccessException that might have occurred during the database creation

The UserDatabaseCreator can be provided to:

- To create and populate your user tables

The UserDatabaseAccess abstract class can be subclassed to:

- provide a Scala API for accessing your database: this is the class from which your application's DAOs are obtained. The type returned from the DatabaseAccessFactory create(...) or open(...) methods is DatabaseAccess[U <: UserDatabaseAccess] - the framework's interface to the database, parameterised with your application's interface to the database.
- provide a hook to detect when the database is closed

The UserDatabaseAccess factory function is a function from DatabaseAccess[U <: UserDatabaseAccess] => U, i.e. given the framework's interface to the database, construct an instance of a subclass of UserDatabaseAccess for your application's interface to it.

Consult the unit tests TestCreatingUserDatabase and TestUserDatabaseAccess for a simple example of using a CreateWorkflowAdapter, UserDatabaseCreator and UserDatabaseAccess factory function to create and populate user database data, and a simple DAO. Also see the example code.

Opening a database

To open an existing database, again use the factory:

```
val databaseAccessFactory = DatabaseAccessFactory()
val codeVersion = CodeVersion("1.0")
val schemaVersion = SchemaVersion("0.4")
val directory = new File("/path/to/storage")
val database = databaseAccessFactory.open(directory, "mydatabase",
        None, // Password
        codeVersion, schemaVersion,
        None, // OpenWorkflowAdapter
        None, // UserDatabaseMigrator
        None) // UserDatabaseAccess creation function
// do stuff
database.get.close()
```

Again, no user database customisation is provided.

The OpenWorkflowAdapter can be provided to:

- Inform the user of the progress of the database open
- Prompt the user repeatedly should the password be incorrect for an encrypted database
- Indicate back to the framework that the opening of an encrypted database should be abandoned (perhaps the user has forgotten the password)
- Inform the user that the database schema is older than the current one and to request permission to migrate the database to the current schema
- Prompt the user to agree to a schema upgrade, and, if granted, the UserDatabaseMigrator will be called to start the migration
- Notify the user of migration refused/success/failure/impossibility (a migration is impossible if the database is of a more recent SchemaVersion than the application passed to open(...))
- Inform the user of any DataAccessException that might have occurred during open(...)

Should a migration be required, and permitted by the user, the UserDatabaseMigrator is used to:

- start a migration: given the currently stored SchemaVersion, apply appropriate upgrade steps inside a transaction.
- commit the migration transaction upon completing the migration
- after a successful migration, record the SchemaVersion passed to open(...) in the Versions table
- roll back the migration transaction if a Spring-JDBC DataAccessException is thrown during migration. This is a fatality - the database cannot be opened if it cannot be migrated

Consult the unit tests TestOpeningDatabaseWorkflow and TestOpeningEncryptedWorkflow for examples of the progress reporting and password (re-)entry. Consult TestDatabaseMigrationWorkflow for an example of database migration.

The Sample Application

BeanMinder is a very simple home finances program that allows you to record credit or debit transactions for a number of accounts. Accounts are held with a certain bank, and have an account code. Credit or debit transactions are made on a certain date, and are later reconciled with your bank statements. The list of transactions contains a running balance so the UI doesn't have to compute it.

There are two views into the data[1]:

Accounts: lists the accounts and their initial/current balances, from where they can be edited or deleted (with cascading deletion of all associated transactions), and from where a new account can be added. With an account selected, you can switch to the Transactions view.

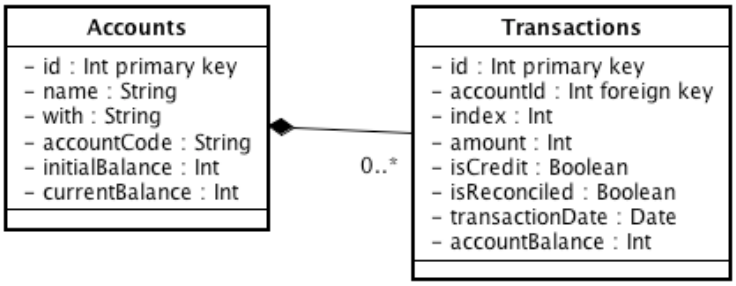
Transactions: lists the transactions in the chosen account, which can be edited or deleted, and from where new transactions can be added.

Although this is limited, it will illustrate several aspects of CommonDb : schema creation, data access objects, domain objects. Admittedly, this is a very simple application.

The source code for the sample application, and its tests can be found in the BeanMinderExample module directory of the CommonDb source repository, under the org.devzendo.commondb.beanminder package. It is built as part of CommonDb, which is a multi-module Maven project. BeanMinderExample has a dependency on CommonDbFramework, and also its tests (it reuses a base trait for tidying up test databases in a temporary directory).

The sample database schema

Here's the database schema that the application will create initially:

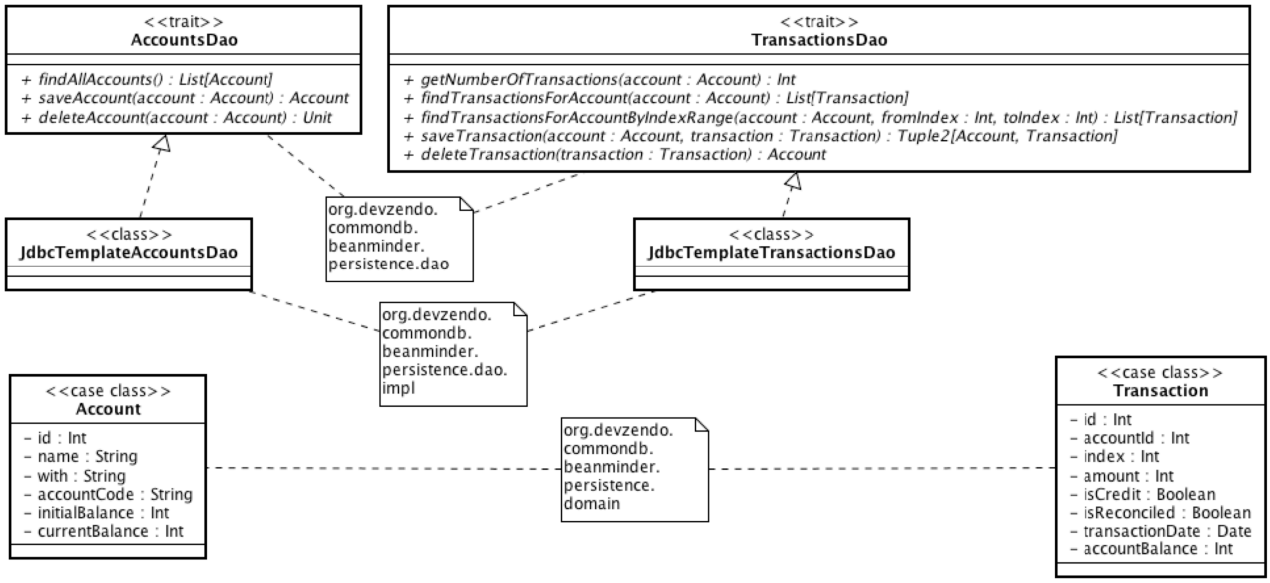


CommonDb doesn't enforce *convention over configuration*, as Ruby on Rails does for table names, but it's a good idea, so I use it here. If you haven't used Rails, the idea is that table names are plural, whereas the objects that you use that represent the rows of the tables are singular. We'll see these objects later, when I discuss the Data Access Object layer.

1 Well there would be, if this was a full application example. However, the example is limited to the use of CommonDb for persistence.

The sample application DAO layer

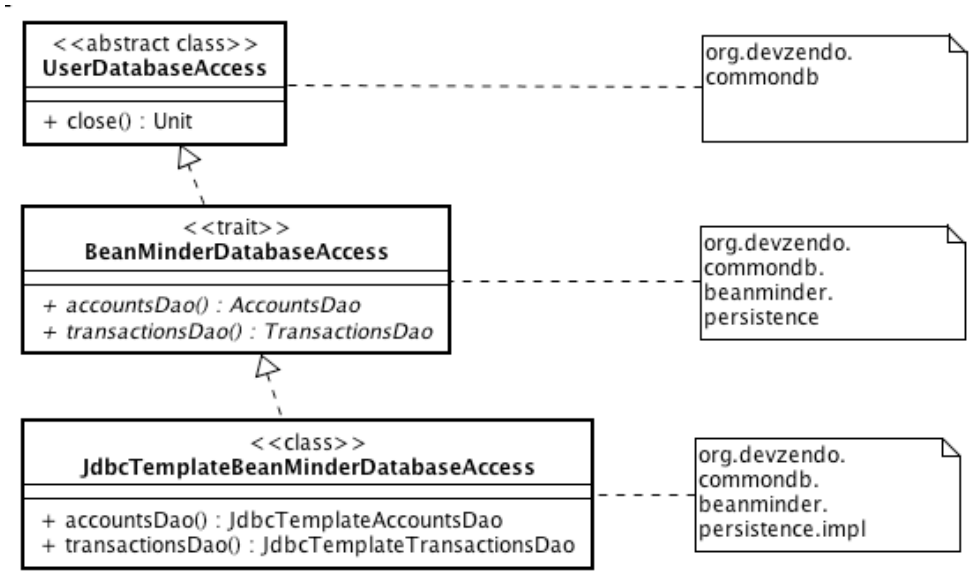
Looking back at the description of the sample application, and the functionality required, the following objects and methods are needed to comprise this technology-independent portion of the DAO layer. The DAOs are traits; the Value Objects are case classes; there are framework-dependent classes for the concrete DAOs. Also note the package hierarchy and fine-grained nature of the operations:



Also note that in the Account and Transaction case classes shown here, the attribute types are String, Int, Boolean, etc. These are the underlying types. In the actual code, representation types and Enumerations are used for these attribute types, to promote type safety outside the AccountsDao (with the exception of the Ints used for id attributes, which have no domain meaning and are only used by the AccountsDao). Therefore, the Account has a signature of id: Int, name: AccountName, withBank: BankName, accountCode: AccountCode, initialBalance: InitialBalance and currentBalance: CurrentBalance. AccountCode is a RepresentationType[String], InitialBalance is a RepresentationType[Int], etc.

Interfacing User Access with CommonDb

To make the AccountsDao and TransactionsDao available to CommonDb, further classes are required:



To ensure your code works with the correct types, you would use the DatabaseAccessFactory thusly (see the BeanMinderUnittest for this code):

```
val databaseAccessFactory = DatabaseAccessFactory[BeanMinderDatabaseAccess]()
// this is a JdbcTemplateDatabaseAccessFactory[BeanMinderDatabaseAccess]
val beanMinderUserDatabaseFactory =
        new ((DatabaseAccess[BeanMinderDatabaseAccess]) => BeanMinderDatabaseAccess) {
        def apply(databaseAccess: DatabaseAccess[BeanMinderDatabaseAccess]) =
                new JdbcTemplateBeanMinderDatabaseAccess(databaseAccess)
        }
val userDatabaseAccess = databaseAccessFactory.create(directory, name, None,
        codeVersion, schemaVersion,
        Some(new BeanMinderCreateWorkflowAdapter),
        Some(new JdbcTemplateBeanMinderUserDatabaseCreator),
        Some(beanMinderUserDatabaseFactory))
// this is an Option[DatabaseAccess[BeanMinderDatabaseAccess]]
```

Note the factory function that instantiates the JdbcTemplateBeanMinderDatabaseAccess given the underlying DatabaseAccess[BeanMinderDatabaseAccess].

The JdbcTemplateBeanMinderDatabaseAccess class definition is a little tricky:

```
class JdbcTemplateBeanMinderDatabaseAccess(
        override val databaseAccess: DatabaseAccess[BeanMinderDatabaseAccess]) extends
        UserDatabaseAccess(databaseAccess) with BeanMinderDatabaseAccess {  ...
```

The UserDatabaseAccess abstract class holds a reference to the framework's interface to the database, the DatabaseAccess[BeanMinderDatabaseAccess], so the definition above constructs this abstract superclass, and mixes in the BeanMinderDatabaseAccess trait to provide implementations of your DAO accessors.

Creating the database schema

When a user creates a new database – i.e. a set of H2 database files within a directory - with CommonDb, the framework will create some objects of its own (the Versions table, the Sequence sequence) and populate them with initial information.

We need to tell the framework about the database schema we need for BeanMinder, and hook into the database initialisation process to create these tables.

This is done in the JdbcTemplateBeanMinderUserDatabaseCreator, as mentioned above. When the database is being created, the createApplicationTables(...) and populateApplicationTables(...) methods of this callback are called:

```
class JdbcTemplateBeanMinderUserDatabaseCreator extends UserDatabaseCreator {
    def createApplicationTables(access: DatabaseAccess[_]) {
        val ddl = List(
            "CREATE TABLE Accounts("
                + "id INT IDENTITY,"
                + "name VARCHAR(40) NOT NULL,"
                + "with VARCHAR(40),"
                + "accountCode VARCHAR(40) NOT NULL,"
                + "initialBalance INT,"
                + "currentBalance INT"
                + ")",
            "CREATE TABLE Transactions("
                + "id INT IDENTITY,"
                + "accountId INT NOT NULL,"
                + "FOREIGN KEY (accountId) REFERENCES Accounts (id) ON DELETE CASCADE,"
                + "index INT NOT NULL,"
                + "amount INT NOT NULL,"
                + "isCredit BOOLEAN,"
                + "isReconciled BOOLEAN,"
                + "transactionDate DATE,"
                + "accountBalance INT"
                + ")")
        ddl.foreach(access.jdbcTemplate.getJdbcOperations.execute(_))
    }
```

In this example, there is no population code.

Domain Objects

Account case classes have a factory method in a companion object that create new Accounts with their id set to -1. The AccountsDAO uses this value when saving the Account in determining whether to insert or update the table (via a PreparedStatement). Similarly for Transactions.

Representation types and Enumerations are used for all attributes of the Account and Transaction, for type safety. These are escaped in the JdbcTemplateAccountsDao when the underlying values are set into a PreparedStatement. The only exception is the id: Int, which has no meaning in the domain, and is used only by the JdbcTemplateAccountsDao.

Domain objects are immutable; when updating an Account with a new Transaction, the saveTransaction method will return an updated Account domain object (with updated current balance) and an updated Transaction (with updated id and index).

User Data Access Objects

The AccountsDAO and TransactionsDAO have their contracts defined in traits, and their concrete implementations use the Spring-JDBC SimpleJdbcTemplate to access their underlying tables.

Several implementation methods are defined as being private to the implementation package; these are used between the two DAO implementations.