Enterprise Client CCEE

Inhaltsverzeichnis

How to download Enterprise Client CCEE

Part of CaptainCasa Installation

The CaptainCasa installation includes a file...:

```
<installDir>
    resources
        addons
            eclnt_ccee.zip
            ...
        ...
    ...
```

This file contains the .jar file together with the source.

Maven

Add the dependency as follows.

```
    <repositories>
        ...
        <repository>
             <id>org.eclnt</id>
             <url>https://www.captaincasademo.com/mavenrepository</url>
        </repository>
        ...
    </repositories>

    <properties>
        <cc.version>20191015</cc.version>
    </properties>

    ...

    <dependencies>
        ...
        <dependency>
            <groupId>org.eclnt</groupId>
            <artifactId>eclntccee</artifactId>
            <version>${cc.version}</version>
        </dependency>
        ...
    </dependencies>
```

Database Management

Uuuuh – why an own database framework?

We know: there is Hibernate, there is JPA, there are others. They all provide mapping. But they also support so much more. What seems to be simple at the beginning turns out to have complexity at the end, if not designed carefully and with knowing the frameworks in detail.

We just needed some smart, fast framework for:

- Managing connections and transactions
- Mapping one table to one class, and doing flexible SQL
- Allowing to add any native SQL in a simple and controlled way

We particularly do **not** need:

- Lazy loading of object instances by pointer navigation
- Embedding the reading of data from SQL into some session concept, in which objects are buffered etc. etc.

Database Connection

ccee_config.properties

File “ccee_config.properties” contains the basic information that is required to access the database. There are two options to places the file:

- Option 1 – in the root package your code: Place the file directly into your Java source directory, so that it gets compiled accordingly

```
Example:

<project>/
  src/
    com/
      aaa/
      bbb/
    ccee_config.properties
```

- Option 2 – in some dedicated directory: you may define an environment variable “ccee_configDirectory”. The file is looked up within this directory in two steps:
  - Step 1: if your application is a CaptainCasa based web application then the file is searched within a sub-directory that has the same name as the context name of your deployed web application, which is internally available through API “HttpSessionAccess.getServletContext().getContextPath()”. - In Tomcat this is by default the name of the application as it is deployed within the tomcat/webapps-directory.

```
Calling Java program – and setting environment variable before:

set ccee_configDirectory=c:\temp\config
java ...


c:/
```

```
  temp/
    config/
      <nameOfWebApp>/
        ccee_config.properties
```

As consequence there is one config directory per deployed web application – and you can configure each web application individually.

- Step 2: if NOT running within the context of a CaptainCasa based web application or if Step 1 was not succesful then the file is directly read from the directory:

```
c:/
  temp/
    config/
      ccee_config.properties
```

Direct definition of database to access

The content of the file is:

```
db_url=jdbc:postgresql://localhost/testccee
db_driver=org.postgresql.Driver
db_username=postgres
db_password=postgres
db_sqldialect=postgres
```

The parameters “db_url”, “db_driver”, “db_username”, “db_password” are the typicaly JDBC parameters for logging on to a database.

The parameter “db_sqldialect” is required if using special query operations (e.g. querying for top 100 elements). Valid values are:

```
postgres
mssql
oracle
mysql
sybase
hsqldb
```

Context / Data source based definition of database to access

In typical applications servers (including Tomcat) you may configure the application to use data sources by defining a simple name on application side. The application server then contains the definition of what this data source actually is.

In this case you need to configure in ccee_confix.properties:

```
db_datasource=MYDATABASE
```

Please note: when the ccee framework accesses the database then it prepends “java:comp/env/jdbc/” in front of the name that you define. So when using the data source “MYDATABASE” then the actual lookup within the ccee functions is done by using “java:comp/env/jdbc/MYDATABASE”.

Dynamic definition of database to access

The connection can also be provided by some own logic. In this case you define a connection provider class name in the “ccee_config.properties” file:

```
db_connectionproviderclassname=xxx.yyy.MyConnectionProvider
```

The class must implement interface IDBConnectionProvider:

```
package org.eclnt.ccee.db;

import java.sql.Connection;
import java.util.ResourceBundle;

public interface IDBConnectionProvider
{
    public Connection createConnection();
}
```

In your implementation you may access the “ccee_config.properties” configuration by using method “Config.getConfigValue()”.

Tenant dependency

The reserved word “@TENANT@” can be used within any value that is defined in the “ccee_config.properties” file. It is replaced with the current tenant at runtime.

```
db_url=jdbc:postgresql://localhost/testccee?currentSchema=@TENANT@
db_driver=org.postgresql.Driver
db_username=postgres
db_password=postgres
db_sqldialect=postgres
```

Database Creation

File “ccee_dbcreatetables.sql” contains the statement to create the database. Statements are separated with “//”.

```
////////////////////////////////////////////////////////////////////////////
CREATE TABLE
    TESTPERSON
(
    personId varchar(50),

    firstName varchar(100),
    lastName varchar(100),
    birthDate date,
    birthTime time,

    PRIMARY KEY ( personId )
)
////////////////////////////////////////////////////////////////////////////
CREATE TABLE
    TESTCOMPANY
(
    companyId varchar(50),

    companyName varchar(100),

    PRIMARY KEY ( companyId )
)
```

The Java-class “DBCreateTables” parses this SQL file and executes statement by statement.

```
package test;

import static org.junit.Assert.assertTrue;
import static org.junit.jupiter.api.Assertions.*;

import org.eclnt.ccee.db.DBCreateTables;
import org.eclnt.ccee.log.AppLog;
import org.eclnt.jsfserver.session.UsageWithoutSessionContext;
import org.junit.jupiter.api.Test;

class TestTableCreation
{

    @Test
    void test()
    {
        UsageWithoutSessionContext.initUsageWithoutSessionContext();
```

```
        AppLog.initSystemOut();
        boolean success = false;
        try
        {
            new DBCreateTables().createTables();
            success = true;
        }
        catch (Throwable t)
        {
            success = false;
        }
        assertTrue(success);
    }

}
```

The processing will not stop if a statement fails, but will continue. As consequence it is possible to append new statements to “ccee_dbcreatetables.sql” and re-process “DBCreateTables” any time.

Mapping

The class DOFWSql provides simple access to one table that is mapped to one class (“data object class”). The class definition is a bean definition (“Pojo”). The bean's properties map to columns of the corresponding database table. The annotations “doentity” and “doproperty” are used to control the mapping.

```
package test;

import java.time.LocalDate;
import java.time.LocalTime;

import org.eclnt.ccee.db.dofw.annotations.doentity;
import org.eclnt.ccee.db.dofw.annotations.doproperty;

@doentity(table="testperson")
public class DOTestPerson
{
    String m_personId;
    String m_firstName;
    String m_lastName;
    LocalDate m_birthDate;
    LocalTime m_birthTime;

    @doproperty(key=true)
    public String getPersonId() { return m_personId; }
    public void setPersonId(String personId) { m_personId = personId; }

    @doproperty
    public String getFirstName() { return m_firstName; }
    public void setFirstName(String firstName) { m_firstName = firstName; }

    @doproperty
    public String getLastName() { return m_lastName; }
    public void setLastName(String lastName) { m_lastName = lastName; }

    @doproperty
    public LocalDate getBirthDate() { return m_birthDate; }
    public void setBirthDate(LocalDate birthDate) { m_birthDate = birthDate; }

    @doproperty
    public LocalTime getBirthTime() { return m_birthTime; }
    public void setBirthTime(LocalTime birthTime) { m_birthTime = birthTime; }
}
```

Annotations “doentity” and “doproperty”

The names of table and columns are derived in the following way:

- @doentity: if no name is explicitly defined (@doentity(table=”nameOfTable”)) then the table name is assumed to be the simple name of the class (class name without package name)
- @doproperty: if no name is explicitly defined (@doproperty(column=”nameOfColumn”))

then the column name is assumed to be the name of the property – following the Java property naming conventions (e.g. the name of a property with “set/getLastName” is “lastName”).

Please check the JavaDoc documentation for detailed information about both annotations. In addition of controlling the naming there are e.g. possibilities to define some special data type mapping rules to transfer the Java representation of the data object into an SQL representation on the database,

Data type mapping

If not using special definitions within the “doproperty” annotation definition then the mapping of data types is done in the following way:

Data type in Java class	Data type used on JDBC level
String	String
int, Integer	Integer
byte, Byte, long, Long	Byte, Long
float, Float, double, Double	Float, Double
LocalDate	java.sql.Date
LocalTime	java.sql.Time
LocalDateTime	java.sql.Timestamp
Date	java.sql.Timestamp
java.sql.Date/Time/Timestamp	java.sql.Data/Time/Timestamp
BigDecimal	BigDecimal
BigInteger	Long
boolean, Boolean	Boolean
UUID	String (“d3c26822-70d8-4a1d-977f-394b04e0fd67”)
byte[]	byte[]

Working with Strings – To trim or not to trim?

If defining a database columns with a data type “CHAR(4)” then the database will always pass back some string value which is filled with spaces at its end.

The ccee-layer supports some automated trimming when reading data from the database:

- The “doproperty”-annotation provides a “trim” definition. You may assign the following values:

```
public enum ENUMTrim
{
    trim,
    notrim,
    undefined
}
```

- You may also switch on “trimming” in general by seeting the parameter “db_autotrim” to “true” in the ccee_config configuration file:

```
...
...
db_autotrim=true
...
...
```

In case of auto-trimming, all strings that are read from the database are trimmed automatically. You may of course override by property using the property-specific definition.

SQL operations

The central class for all SQL operations is “DOFWSql”.

Saving to database

“DOFWSql.saveObject()” saves an instance to the database. It performs an update if the object exists – and and insert if the object does not exist.

```
DOTestPerson p = new DOTestPerson();
p.setPersonId(""+System.currentTimeMillis());
p.setFirstName("Test first name");
p.setLastName("Test last name");
p.setBirthDate(LocalDate.of(1969,6,6));
p.setBirthTime(LocalTime.of(13,30,0));

DOFWSql.saveObject(p);
```

Deleting from database

“DOFWSql.delete()” deletes instances.

```
DOFWSql.delete
(
    DOTestCompany.class,
    new Object[] {"companyId","0001"}
);
```

You may pass a series of criteria:

```
DOFWSql.delete
(
    DOTestPerson.class,
    new Object[]
    {
        "lastName","Test last name",
        "firstName","Test first name"
    }
);
```

Each pair of “column name” and “value” is interpreted as an equals condition (“=”). The pairs are concatenated with an “AND” operator.

You may pass a complex query:

```
DOFWSql.delete
(
    DOTestPerson.class,
    new Object[]
    {
       "lastName", LIKE, "A%",
       AND,
       "firstName",LIKE,"%A"
    }
);
```

Querying list of objects from database

“DOFW.query()” queries instances from the database.

There are two query-methods:

- query(Class clazz, Object[] criteria)

- query(Class clazz, Object[] criteria, Object[] orderBy)

The following query scans the full table – there are no selection criteria specified:

```
List<DOTestPerson> ps = DOFWSql.query
(
    DOTestPerson.class,
    new Object[] {}
);
```

The following query scans the table for certain column values. An implicit “AND” condition is assumed.

```
List<DOTestPerson> ps = DOFWSql.query
(
   DOTestPerson.class,
   new Object[]
   {
       "firstName","Captain",
       "lastName","Casa"
   }
);
```

The following query scans the table with a complex query:

```
List<DOTestPerson> ps = DOFWSql.query
(
   DOTestPerson.class,
   new Object[]
   {
       "firstName",LIKE,"C%",
       AND,
       BRO,
           "lastName",LIKE,"Casa%",
           OR,
           "lastName",LIKE,"Cassa%",
       BRC
   }
);
```

The constants for LIKE, AND, BRO (bracket open), BRC (bracket close), etc. are available via interface ICCEEConstants. So the best way of using these constants is to implements the interface by your application class:

```
public class MyXyzClass implements ICCEEConstants
{
    // now the AND/OR/... are available!
}
```

By using the query()-method that provides the “orderBy” parameter you may pass sort information into the query:

```
List<DOTestPerson> ps = DOFWSql.query
(
   DOTestPerson.class,
   new Object[]
   {
       "firstName","Captain",
       "lastName","Casa"
   },
   new Object[]
   {
       "firstName",
       "lastName"
   }
);
```

By default an ascending sort order is assumed. You may fine control in the following way:

```
List<DOTestPerson> ps = DOFWSql.query
(
   DOTestPerson.class,
```

```
    new Object[]
    {
        "firstName","Captain",
        "lastName","Casa"
    },
    new Object[]
    {
        "firstName",DESC,
        "lastName", ASC
    }
);
```

Queries for one object

By using the methods...

- DOFWSql.queryOne(...)

...you can query for exactly one object. Either the first object of the result set or null is returned.

Query for limited number of objects

By using the mehtods...

- DOFWSql.queryTop(...)

...you can define a maximum number of objects that the database returns – regardless if the number of matching objects actually is higher.

Pleasy pay attention: because the SQL syntax is not consistent throughout various databases, you need to define the “db_sqldialect” carefully in the configuration.

Querying for NULL value

NULL always has some special treatment within databases. When checking for NULL values use the NULL constant which is part of the ICCEEConstants-interface.

```
List<DOTestPerson> tps = DOFWSql.query
(
    DOTestPerson.class,
    new Object[] {"birthDate",ISNOT,NULL}
);
```

“Free Style” Queries

The SQL operations shown in the previous chapter are a quite nice abstraction of the database processing. They...

- hide the actual SQL statement
- hide the mapping of data between the Java-object and the SQL-database
- ensure a compatibility across multiple databases

Of course they are limited! For example they only operate on one class/table. And there is a limitation “by purpose”: they should streamline the access to the data for all the “80%” cases of working with the database – while being open to “free style” arrange SQL operations for the remaining “20%”.

“Guided SQL” queries

“Guided SQL” means that there is still some layer covering complexity, but that you are already on an “SQL-level” of developing.

```
List<Object[]> lines = DOFWSql.queryGuidedSql
(
    DOTestPerson.class,
    new String[]
    {
        "?p(firstName)",
        "TRIM(?p(lastName))",
        "CONCAT(?p(firstName),?p(lastName))",
        "?p(birthDate)"
    },
    "?p(firstName) LIKE ?v(firstName) AND CONCAT(?p(firstName),?p(lastName)) LIKE ?v()",
    "CONCAT(?p(firstName),?p(lastName)),?p(firstName)",
    new Object[] {"%A%","%A%"}
);
for (Object[] line: lines)
{
    System.out.println("***********");
    for (Object o: line)
    {
        System.out.println(o);
    }
}
```

You pass...

- the class (table) to query
- the columns you want to query – either by directly naming their property of by also using SQL functions
- the condition string – again it may contain e.g. SQL functions
- the order string
- the values of the parameters that are referenced within the condition string

Within the string definitions you can use placeholders:

- “?p(xxx)” is a placeholder for a property – it is transferred at runtime into the column name that is assigned to this property
- “?v(xxx)” is a placeholder for a value with reference to some property. Meaning: the data type conversion from Java to database and the conversion from database to Java is following the data type of the corresponding property. If there is not property reference that can be used then just pass “?v()”.

At runtime the strings are parsed and corresponding replacements are done within the string. The resulting SQL is executed as PreparedStatement.

You see: this is no “free style” SQL yet! But it already does a lot of things that you would normally have to do on your own:

- Conversion of property names to column names
- Value-conversion in both directions
- Adding the tenant condition if there is a tenant column in the class definition

Free style queries

Last but not least there is a simple possibility to add and run any type of SQL. Free style querying is done in the following way:

```
public static int readNumberOfIssuesWithLabel(final String itemId,
                                               final String labelTypeId,
                                               final String labelValueId)
{
    final ObjectHolder<Integer> result = new ObjectHolder<Integer>();
    result.setInstance(0);
    new DBAction()
    {
        @Override
```

```
        protected void run() throws Exception
        {
            PreparedStatement ps = createStatement
            (
                "SELECT DISTINCT WKMISH_ISSID"
                + " FROM WKMISH"
                + " INNER JOIN WKMISL ON"
                + "   WKMISH.WKMISH_ISSID = WKMISL.WKMISL_ISSID"
                + "   AND WKMISH.WKM_TENANT = WKMISL.WKM_TENANT"
                + " WHERE WKMISH.WKM_TENANT=?"
                + "   AND WKMISH_FK_ITEMID=?"
                + "   AND WKMISL_FK_LBLTYP=?"
                + "   AND WKMISL_FK_LBLVAL=?"
            );
            int counter = 1;
            ps.setString(counter++,getTenant());
            ps.setString(counter++,itemId);
            ps.setString(counter++,labelTypeId);
            ps.setString(counter++,labelValueId);
            ResultSet rs = ps.executeQuery();
            int resultCounter = 0;
            while (rs.next())
                resultCounter++;
            result.setInstance(resultCounter);
        }
    };
    return result.getInstance();
}
```

(Please do not check if it really makes sense to do the query in the way it is shown... - this is just an example!)

The query is done by a prepare statement, which is obtained within the processing of a DBAction (please check the next chapter on transaction management as well!).

Please note: because the processing of the query is done within the “run()”-method of DBAction, there are certain rules:

- Variables are only visible from the outer processing to the inner processing if they are defined as final variables.
- For transferring single values, there is a class “ObjectHolder” which is created in the outside processing and which is populated in the inside processing.

The condition definition of a query...

You already saw from the previous examples that an important part of e.g. querying the database is to define the conditions for the query. This is done by passing an object array (Object[]), example:

```
    new Object[]
    {
       "lastName", LIKE, "A%",
       AND,
       "firstName",LIKE,"%A"
    }
```

The interface ICCEEConstants contains all the comparators and logical operators that you may select from:

- The comparators are
    - IS, ISNOT, LIKE, GREATER, LOWER, GREATEREQUAL, LOWEREQUAL, IN, BETWEEN
- The logical operators are
    - AND, OR, BRO (“(“), BRC (“)”)

Please pay attention: even though the constants are defined as String-constants you must never redefine the String on your own!

```
WRONG:
    new Object[]  { "lastName", “LIKE”, "A%", }
CORRECT:
    new Object[]  { "lastName", LIKE, "A%", }
```

For the value argument (the one on the right side of a comparison) you may pass an object that holds the same data type as the one that is used for the corresponding property within the data object class. The conversion to the corresponding JDBC data type is done automatically.

The IN and the BETWEEN comparator

There are are two comparators for which you need to use special object types for the value argument: the IN and the BETWEEN comparator.

- Use class “ValuesIN” for building a collection of objects to pass for IN
- Use class “ValuesBETWEEN” for building the from/to parameters to pass for BETWEEN

Example:

```
new Object[]
{
    "companyName",IN,new ValuesIN<String>(new String[] {"AAA0","AAA1"}),
    OR,
    "companyName",BETWEEN,new ValuesBETWEEN<String>("AAA1","AAA3"),
    OR,
    “companyName”,IS,”AAA10”
}
```

Querying only some columns

When using the default query-methods then always all object properties are loaded that are mapped to corresponding database columns.

You may use the methods with the nam “queryColumnData” in order to only load selected properties/columns:

```
List<DOTestOrder> dot = DOFWSql.queryColumnData
(
    DOTestOrder.class,
    new Object[] {"orderId","orderName"},  // selection of columns
    new Object[] {“orederName”,LIKE,”A%”}  // where condition
);
```

Of course it's now up to you to handle the objects that are returned back with great care – because only these properties are loaded that you explicitly selected!

Transaction management

By default each database operation runs in some own transaction. But – of course! - this should not be the way to use a database.

Class DBAction

By using class “DBAction” you can define operations that run within one transaction.

```
new DBAction()
{
    protected void run() throws Exception
    {
        for (int i=0; i<10; i++)
```

```
        {
            DOTestPerson p = new DOTestPerson();
            p.setPersonId(""+(System.currentTimeMillis()+i));
            p.setFirstName("Test first name A");
            p.setLastName("A Test last name");
            p.setBirthDate(LocalDate.of(1969,6,6));
            p.setBirthTime(LocalTime.of(13,30,0));
            DOFWSql.saveObject(p);
        }
    }
};
```

The code that is executed in the “run()” method is embedded into the opening and closing of the transaction – including the corresponding error management, if a transaction fails.

Nesting DBAction operations

Of course you can nest DBAction operations. The rule is: the outest DBAction is the one to control the transaction. Only this DBAction instance is the one to pass the commit to the database.

Internally the transaction/connection management to the database is done by so called thread-binding. The outest DBAction binds the transactional information to the current thread – and releases it after the processing. The inner DBActions recognize that there is already some transaction bound to the thread and as consequence process their activities within this transaction.

Suppressing Compiler Warnings

Code like the following...

```
new DBAction()
{
    protected void run() throws Exception
    {
        ...
        ...
    }
};
```

...makes the Java-compiler believe that you create an object without using it. Result: dependent on your compiler settings (i.e. which type of compiler messages are interpreted as info/warning/error) you may receive a warning “The allocated object is never used”.

There are two ways to go:

- Either: You may of course switch off these messages by telling the compiler to ignore these messages.
- Or: You need to tell the compiler that the object really is needed.

For this reason, DBAction provides a method “noWarning()” which you may use in the following way:

```
new DBAction()
{
    protected void run() throws Exception
    {
        ...
        ...
    }
}.noWarning();
```

Working with more than one database

By default CCEE assumes that your application is working with one database and that you want to work within one transactional context for this database. But there are scenarios in which you want to overcome this default. Examples:

- You may work with two databases. Maybe the administration data of your system (user, rights, …) is stored in a different database than your application data. Or you may have one database for transactional data and one for reporting data.
- You may work in different transactional contexts within one database. Maybe you want to commit some data immediately (e.g. you have a DB based counter, and always want to update the current counter after picking a new one).

For this reason CCEE provides the ability to define different contexts – each one being represented by a name, that you may freely assign.

Configuration files

If you use a dedicated context then the configuration files are to be named in the following way:

- “ccee_config_<nameOfContext>.properties”
- “ccee_dbcreatetables_<nameOfContext>.sql”

Example:

```
ccee_config_admindb.properties
ccee_dbcreatetables_admindb.sql
```

APIs

In all APIs you now have to pass the name of the context. All methods of the APIs are available both for the default case (working with one context) and for the multi-context-case. The name of the context is always the first parameter:

Example:

```
List<DOUser> users = DOFWSql.query
(
   “admindb”, // context name
   DOUser.class,
   new Object[] {}
);
```

Working with default context and with special contexts

You can explicitly work both with the default context and with special contexts at the same point of time. Internally the default context is just a normal context with a name that is pre-defined by CaptainCasa.

Tenant Management

The database management of the ccee-library is “tenant-aware”. It supports different strategies of separating data between multiple tenants.

- Explicit tenant column as key of each table.

  In this case every table has one leading column, that is part of the key of the table. When acessing the databse through DOFWSql functions the tenant will be automatically

added into the generated database SQL statement.

- Different database schemes for each tenant.

  There is a database schema for each tenant – so some table “Person” will occur multiple time as “tenant1.Person” and “tenant2.Person”.

- Different databases for each tenant.

  There is an individual database instance (database-URL) for each tenant.

The nice thinkg: all three aspects can be combined – if you want, which makes the decision how to store data very flexible.

Configuration and Usage

By tenant column

If the database table is...

```
CREATE TABLE
    TESTPERSON
(
    tenant varchar(10),
    personId varchar(50),

    firstName varchar(100),
    lastName varchar(100),
    birthDate date,
    birthTime time,

    PRIMARY KEY ( tenant, personId )
)
```

...then the Java class for the mapped Pojo looks as follows:

```
package test;

import java.time.LocalDate;
import java.time.LocalTime;

import org.eclnt.ccee.db.dofw.annotations.doentity;
import org.eclnt.ccee.db.dofw.annotations.doproperty;

@doentity(table="testperson", tenantColumn=”tenant”)
public class DOTestPerson
{
    String m_personId;
    String m_firstName;
    ...
    ...

    @doproperty(key=true)
    public String getPersonId() { return m_personId; }
    public void setPersonId(String personId) { m_personId = personId; }

    @doproperty
    public String getFirstName() { return m_firstName; }
    public void setFirstName(String firstName) { m_firstName = firstName; }

    ...
    ...
}
```

You see:

- In the “doentity”-annotation the tenant column is defined.
- The tenant-column is not mapped into the class!

Consequence: you just access the database as normal (DOFWSql...), and the database management will take care of adding the “WHERE tenant='...'” to any SQL statement that is sent to the database.

Of course: if you define your own “native” database statements then you need to take care of the tenant column!

By schema, by database

This is to be defined in the ccee_config.properties file: you may add the placeholder @tenant@ into the definition of the URL and into the definition of the explicit schema that you want to use.

How is the tenant set at runtime?

The database management uses the CaptainCasa-tenant interface “ITenantAccess” in order to choose the right tenant for its operations.

In the tenant management there is the class “DefaultTenantAccess” which allows to bind the current tenant...

- ...to the http dialog session (DefaultTenantAccess.associateTenantWithCurrentSession)
- ...to the current thread (DefaultTenantAccess.associateTenantWithCurrentThread)

The tenant is passed as simple string.

This means: if a database function inside the ccee-framwork is requesting the information which tenant is currently to be used then it first checks if there is a tenant in the current http session – and then, if it cannot find one, checks if there is a tenant in the current thread.

...by http session within CaptainCasa UI processing

This is the default for all processing that is directly invoked by the user interface. You bind the tenant to the http session (e.g. after logon of the user) – and from now on any database activity that is directly called from the UI is living in the corresponding tenant.

...by thread

The following examples are typical cases in which it is useful to bind the tenant to the thread:

- You start some separate thread from the UI processing. This thread is not managed by e.g. the servlet engine anymore and does not have access to the http session as consequence.
- You have some code being called “from outside” (e.g. REST-API, job scheduler, etc.).

Dealing with large data (Clob etc.)

You may use a special management for properties/columns that contain large amounts of data, e.g. “Clob”-data (character large object).

Within the “doproperty” annotation there is a flag “onlyReadWithSingleReadOperations”. Example:

```
@doentity(table="TESTTEXT")
public class DOTestText
{
    UUID m_textId;
    String m_textContent;

    @doproperty(key=true)
    public UUID getTextId() { return m_textId; }
    public void setTextId(UUID textId) { m_textId = textId; }
```

```
    @doproperty(onlyReadWithSingleReadOperations=true)
    public String getTextContent() { return m_textContent; }
    public void setTextContent(String textContent) { m_textContent = textContent; }
}
```

If this property is set to “true” then the corresponding property will not be read from the database during normal query operations.

If will only be read if accessing the object in “single read operations”, which are:

- “queryTop(...)” with top being set to “1”
- “rereadObject(...)”

In all other query operations it will left out and will be passed back as “null” as consequence.

Example

The following example queries data from in a “mass way”. When looping through the data, then every 5th object is “deeply” read:

```
List<DOTestText> tts = DOFWSql.query
(
    DOTestText.class,
    null
);
int counter = 0;
for (DOTestText tt: tts)
{
    counter++;
    {
        UUID id = tt.getTextId();
        String tc = tt.getTextContent();
        if (tc != null && tc.length() > 100)
            tc = tc.substring(0,100);
        System.out.println(id + " // " + tc);
    }
    if (counter % 5 == 0)
    {
        boolean stillExists = DOFWSql.rereadObject(tt);
        if (stillExists == false)
            throw new Error("Object does not exist anymore: but should exist...!");
        {
            UUID id = tt.getTextId();
            String tc = tt.getTextContent();
            if (tc != null && tc.length() > 100)
                tc = tc.substring(0,100);
            System.out.println("REREAD:\n" + id + " // " + tc + “\n”);
        }
    }
}
```

The output is:

```
2806be1d-c3ad-4454-b1a6-1d1f2dce72eb // null
4365b59d-18b2-4b10-a29a-2d1aea3afdb2 // null
0cdf0fc4-47d4-4cb8-9ee9-988083ab6e5a // null
7f134067-6d55-49a5-aa32-c47222511b06 // null
d520bafb-0236-40b8-91d2-c20389e3ef83 // null
REREAD:
d520bafb-0236-40b8-91d2-c20389e3ef83 //
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456

96bfe6bb-94d4-41a5-9525-6902ee2d4cb6 // null
895c19f1-7079-476f-a376-c2b90d7d6195 // null
f0b022af-d303-4502-9ece-d42b49dbaf73 // null
6d3e24fd-669f-4d32-a66d-4b5b1db25b38 // null
7c01ab26-dac8-4e92-a144-3b0655cda131 // null
REREAD:
7c01ab26-dac8-4e92-a144-3b0655cda131 //
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456
```

```
fd23eaae-0d6d-42f9-987b-a13544955c6f // null
06260529-f7c6-4b2c-8bf3-c9c481302995 // null
6bbd20e2-4b4b-4684-9d03-cd69fa241adc // null
…
```

You see: the normal query does not read the property “textContent”. If re-reading the object then the “textContent” is read.

Pay attention when reading and saving...

It's now you take care about reading and saving the object properly! If you read the object with normal queries then the corresponding properties will be “null”. If you now save the object without rereading them by a single-read-operation then the “null” value will be written to database.

Make sure that your object is “deeply read” before it is passed to some detail processing!

Configuration issues

The configuration of the database management is kept in the “ccee_config” properties file as explained at the beginning of this section.

Use for own configuration - API

You may add own configuration properties to the configuration file. Example:

```
db_url=jdbc:postgresql://localhost/testccee
db_driver=org.postgresql.Driver
db_username=postgres
db_password=postgres
db_sqldialect=postgres

own_param1=...
own_param2=...
```

You may access the configuration by using the “Config” class (package org.eclnt.ccee.config):

```
String ownParam1 = Config.getConfigValue(“ownParam1”);
```

If working with multiple database contexts use the following method:

```
String ownParam1 = Config.getConfigValue(“myContext”,“ownParam1”);
```

Multiple contexts - cascading configuration

By default you define one “ccee_config” file for each context (see chapter “Working with multiple databases”). By adding the following definition to the default “ccee_config.properties” file...

```
db_url=jdbc:postgresql://localhost/testccee
db_driver=org.postgresql.Driver
db_username=postgres
...
...
...
config_cascading=true
...
```

...you may centralize certain definitions: when a config parameter is read, then first it is checked if there is a parameter definition in the properties file belonging to the context

("ccee_config_<contextName>.properties"). If not finding the parameter in this specific file then the parameter is read from the default configuration file.

Job Scheduling

From update 20181210 on we added a job scheduling framework to the CCEE-framework. Internally it is based on top of the “Quartz”-framework (http://www.quartz-scheduler.org).

Adding Scheduling to your application

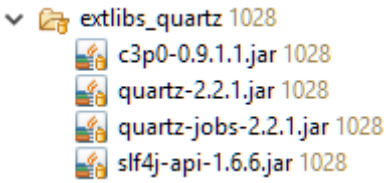
Libraries - Processing

The CaptainCasa library “eclnt_ccee.jar” contains the CaptainCasa-specific runtime issues on top of Quartz. You need to add the Quartz libraries into your project in addition.

The nicest way is to load the libraries via Maven. We internally test with Quarty version 2.1. Please use the same version.

```
        <dependency>
            <groupId>org.quartz-scheduler</groupId>
            <artifactId>quartz</artifactId>
            <version>2.2.1</version>
        </dependency>
        <dependency>
            <groupId>org.quartz-scheduler</groupId>
            <artifactId>quartz-jobs</artifactId>
            <version>2.2.1</version>
        </dependency>
```

After resolving the dependencies the following libraries are available:



If using the default CaptainCasa directory structure you may copy the libraries into the webcontent/WEB-INF/lib directory of your project.

Libraries - User Interface

CaptainCasa provides some pre-built user interfaces for defining jobs and for executing their execution. The user interfaces are part of CaptainCasa's “page bean extension” package. This package comes as addon “eclnt_pbc.zip” within the “/resources” package of your installation. Copy the contained “eclbt_pbc.jar” into the webcontent/WEB-INF/lib folder of your project.

Database

The database needs to be extended by four tables:

```
////////////////////////////////////////////////////////////////////////////
CREATE TABLE
    CCEEActiveScheduler
(
    schedulerId varchar(50),
    schedulerInstanceId varchar(50),
    timestampActivation timestamp,
```

```
    PRIMARY KEY ( schedulerId )
)
////////////////////////////////////////////////////////////////////////////////
CREATE TABLE
    CCEEJob
(
    tenant varchar(50),
    id varchar(50),
    className varchar(100),
    parameters varchar(2000),
    timing varchar(100),

    PRIMARY KEY ( tenant,id )
)
////////////////////////////////////////////////////////////////////////////////
CREATE TABLE
    CCEEJobExecution
(
    tenant varchar(50),
    id varchar(50),
    jobId varchar(50),
    jobClassName varchar(100),
    jobParameters varchar(2000),
    status varchar(10),
    jobStarted timestamp,
    jobEnded timestamp,

    PRIMARY KEY ( tenant,id )
)
////////////////////////////////////////////////////////////////////////////////
CREATE TABLE
    CCEEJobExecutionProtocol
(
    tenant varchar(50),
    id varchar(50),
    protocol text,

    PRIMARY KEY ( tenant,id )
)
```

Create the tables in your default database - which is the one that is addressed by the ccee_config.xml (see “Database Management” for more information).

The SQL script above was executed on PostgreSQL. The only “critical” field which may be different from database to database is the “protocol”-field in table CCEEJobExecutionProtocol. Please use the “CLOB” data type of your database, if data type “text” is not available in your DB environment.

Generate the tables by API

The SQL script for creating the tables is also available as resource file of the eclnt_ccee.jar library. The location is:

```
org/eclnt/ccee/quartz/data/cceejobtables.sql
```

The execution of the file is available as API:

```
CCEEJobLogic.createUpdateJobTables();
```

Developing a job

A job is a Java class supporting the interface “ICCEEJob”:

```
package org.eclnt.ccee.quartz.logic;

public interface ICCEEJob
{
    public void executeJob(String parameters,
                           CCEEJobExecutionContext jobExecutionContext);
}
```

The “executeJob” method is the one that is executed in the context of the job

processing. There are two parameters:

- “parameters” - this is a string that is part of the job definition. The format of the string is up to you. If it contains some XML, JSON or whatever string is completely up to your preferences.
- “jobExecutionContext” - this is some context with useful methods. One of them e.g. being a “addToProtocol(...)” method, which you can use to write some log into a job protocol.

You job may implement interface “IjobConstants” in addition, which is a collection of all Java-constants that are used within the job management.

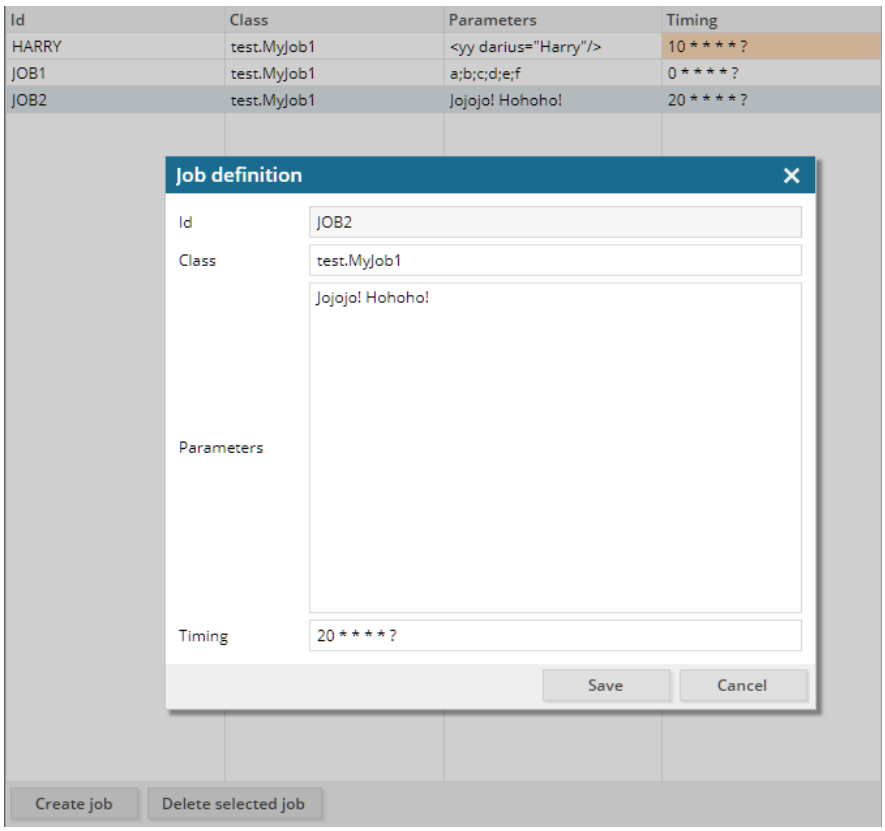
Example: the following class is a valid job implementation:

```
package test;

import org.eclnt.ccee.quartz.logic.CCEEJobExecutionContext;
import org.eclnt.ccee.quartz.logic.ICCEEJob;
import org.eclnt.ccee.quartz.logic.IJobConstants;

public class MyJob1 implements ICCEEJob, IJobConstants
{
    @Override
    public void executeJob(String parameters,
                           CCEEJobExecutionContext jobExecutionContext)
    {
        System.out.println("JOB STARTED ==========================");
        System.out.println("parameters: " + parameters);
        jobExecutionContext.addToProtocol(PROTOCOL_INFO,"Jappa");
        jobExecutionContext.addToProtocol(PROTOCOL_INFO,"Dappa");
        jobExecutionContext.addToProtocol(PROTOCOL_INFO,"Duuuu");
        System.out.println("JOB ENDED   ==========================");
    }
}
```

Setting up and executing jobs

Setup

The setup of a job is done by adding corresponding items to the database table “CCEEJob”. Within the “eclnt_pbc.jar” library there is a page bean component “JobDefinitionList” which you can either directly call or which you can embed into your pages:



The page bean components shows all the jobs of the current tenant and allows to edit the details (double click) or create new job definitions.

Each job definition consists out of:

- an Id
- the name of the class which is to be executed
- the parameters string which is passed as configuration into the job execution
- the timing definition – this is the definition when the job is executed by the job scheduler. It internally uses the Quartz-cronSchedule definition which is based on the syntax of Unix-cron jobs definitions. Please check the Quartz-documentation on details of this timing definition, e.g. http://www.quartz-scheduler.org/documentation/quartz-2.x/tutorials/crontrigger.html

(Re)Starting the Scheduler

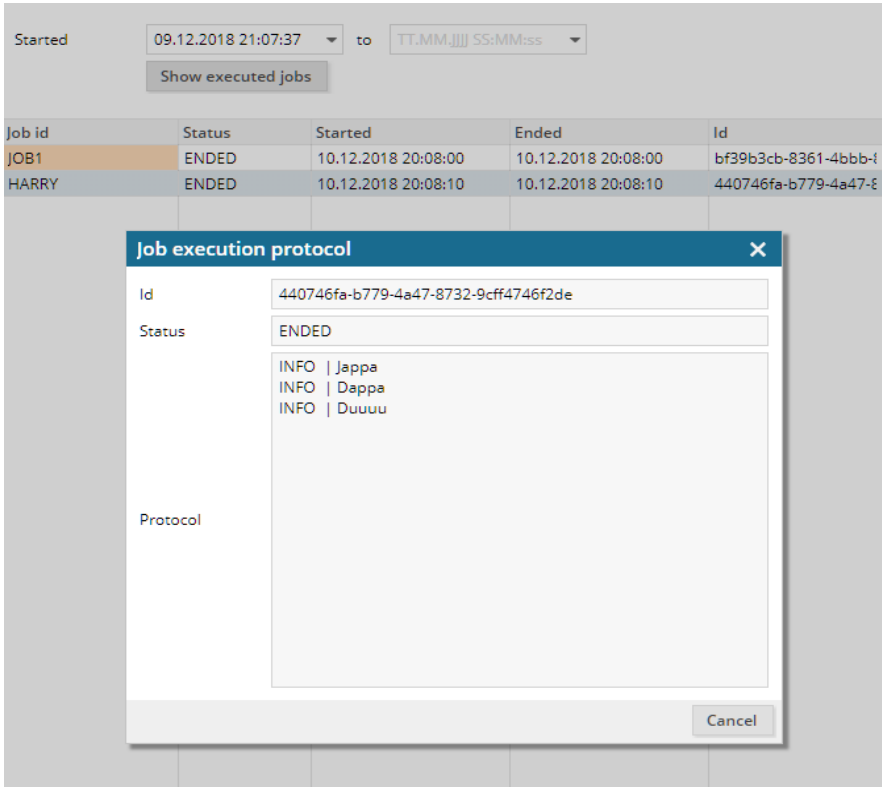
The actual scheduling is called by calling Java-API:

```
org.eclnt.ccee.quartz.logic.QuartzSchedulerManager.setup();
```

Calling this method will transfer all job definitions into the scheduling and will start the scheduling processing. After calling this method the jobs will be executed according to their job definitions.

Monitoring

The page bean component “JobExecutionList” shows the executed jobs together with their status. By double clicking one item you can take a look into the job protocol:



At execution point of time each job execution is registered with a separate unique id. The job passes the following status:

- STARTED: the job is started and is running
- ENDED: the job has succesfully ende
- ERROR: an exception/error occurred during job processing

Architectural issues

Transaction Management

The execution of a job is done within one database transaction – when using the database access framework that is part of CCEE. This means: the transaction that commits the data that is updated by your application is the same transaction that is committed to set the job status from “STARTED” to “ENDED”.

Tenant Management

The job management is fully tenant-aware – using the CaptainCasa tenant management (DefaultTenantManager). This means: the tenant is part of the normal environment data when a job is executed.

Utilities

Configuration

The class “Config” provides access to the configuration contained in “ccee_config.properties”.

```
String value = Config.getConfigValue(“db_url”);
```

Please note: the result should not be buffered on any level, because it might depend on the tenant that is the active one when calling the function.

You are free to also add own configuration parameters to “ccee_config.properties” - and access them via the Config class. When adding own parameters, define some prefix (“xxx_”) that reflects your company/project so that the probability of name conflicts is decreased.

Logging

The class “AppLog” provides a default Java logging.

```
AppLog.L.log(LL_INF,”This is an info message”);
AppLog.L.log(LL_ERR,”This is an error message”,exc);
```

By default the AppLog-logging logs to the same log that the CaptainCasa server processing uses.

For testing (e.g. in JUnit tests) you may explicitly configure the log to output its content to the console. You do so by calling the method:

```
AppLog.initSystemOut();
```

JAXB Helper

The class JAXBUtil transfers simple Bean-instances into an XML representation – and vice versa. The simple bean instance must be annotated with “@XmlRootElement”.

Transforming object to XML

Example: the class DOTestPerson...

```
@XmlRootElement
@doentity(table="testperson")
public class DOTestPerson
{
    String m_personId;
    String m_firstName;
    String m_lastName;
    LocalDate m_birthDate;
    LocalTime m_birthTime;

    @doproperty(key=true)
    public String getPersonId() { return m_personId; }
    public void setPersonId(String personId) { m_personId = personId; }

    @doproperty
    public String getFirstName() { return m_firstName; }
```

```
    public void setFirstName(String firstName) { m_firstName = firstName; }

    @doproperty
    public String getLastName() { return m_lastName; }
    public void setLastName(String lastName) { m_lastName = lastName; }

    @doproperty
    public LocalDate getBirthDate() { return m_birthDate; }
    public void setBirthDate(LocalDate birthDate) { m_birthDate = birthDate; }

    @doproperty
    public LocalTime getBirthTime() { return m_birthTime; }
    public void setBirthTime(LocalTime birthTime) { m_birthTime = birthTime; }
}
```

...can be transformed to XML in the following way...

```
List<DOTestPerson> ps = DOFWSql.query(DOTestPerson.class,new Object[] {});
for (DOTestPerson p: ps)
{
    String xml = JAXBUtil.marshalSimpleObject(p);
    System.out.println(xml);
}
```

...so that the output is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<doTestPerson>
    <birthDate/>
    <birthTime/>
    <firstName>A Test first name</firstName>
    <lastName>Test last name A</lastName>
    <personId>1531557272960</personId>
</doTestPerson>

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<doTestPerson>
    <birthDate/>
    <birthTime/>
    <firstName>A Test first name</firstName>
    <lastName>Test last name A</lastName>
    <personId>1531557273031</personId>
</doTestPerson>

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<doTestPerson>
    <birthDate/>
    <birthTime/>
    <firstName>A Test first name</firstName>
    <lastName>Test last name A</lastName>
    <personId>1531557273105</personId>
</doTestPerson>
```

Transforming XML to object

```
String xml = "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>\r\n" +
        "<doTestPerson>\r\n" +
        "    <firstName>Test first name A</firstName>\r\n" +
        "    <lastName>A Test last name</lastName>\r\n" +
        "    <personId>HUHU</personId>\r\n" +
        "</doTestPerson>\r\n" +
        "";

DOTestPerson p = (DOTestPerson)JAXBUtil.unmarshalSimpleObject(xml,DOTestPerson.class);
```