| **File name** | YouTestMe Web Development Manual |
| --- | --- |
| **Author** |  |
| **Confidentiality** | Internal |
| **Last save date** | Monday, June-28-2021 at 8:57:00 AM |

Table of Contents

[1 Introduction 4](#_Toc75763081)

[2 Open Issues 4](#_Toc75763082)

[3 Mandatory Development Procedure 4](#_Toc75763083)

[4 Setting up Classroom2020 5](#_Toc75763084)

[4.1 Create DB schema 5](#_Toc75763085)

[4.1.1 Editing wrapper script 6](#_Toc75763086)

[4.1.2 Troubleshoot – editing other files 6](#_Toc75763087)

[4.1.3 Running script from command prompt 7](#_Toc75763088)

[4.2 Load data via Batch Project 7](#_Toc75763089)

[4.3 Load data from template databases 7](#_Toc75763090)

[4.4 Setting up www\_source project 7](#_Toc75763091)

[4.5 Using SVN Branches to do Developemnt 8](#_Toc75763092)

[4.5.1 Download Apache Tomcat 7.0 9](#_Toc75763093)

[4.6 Import projects into Eclipse 10](#_Toc75763094)

[4.6.1 Adding Tomcat server 10](#_Toc75763095)

[4.6.2 Troubleshoot: Can’t find server tab 15](#_Toc75763096)

[4.6.3 Setting up server Context 17](#_Toc75763097)

[4.6.4 Running a project 18](#_Toc75763098)

[4.6.5 Flying through application 18](#_Toc75763099)

[4.7 Setting up Sass preprocessor 22](#_Toc75763100)

[4.7.1 How to install Sass and Compass 22](#_Toc75763101)

[4.7.2 Setting environment 22](#_Toc75763102)

[4.7.3 Compass configuration file 24](#_Toc75763103)

[4.8 Sass project creation 25](#_Toc75763104)

[4.8.1 Variables 26](#_Toc75763105)

[4.8.2 Nesting 27](#_Toc75763106)

[4.8.3 Compass watch 27](#_Toc75763107)

[4.8.4 Practice Sass 27](#_Toc75763108)

[4.9 GDAO Project 28](#_Toc75763109)

[4.10 Build and Deployment Procedures 28](#_Toc75763110)

[4.11 Unix machines in Montreal Cloud 28](#_Toc75763111)

[5 Tagging the Code 29](#_Toc75763112)

[5.1 Naming convention 29](#_Toc75763113)

[5.2 Tags location 30](#_Toc75763114)

[5.3 Code Compatibility 31](#_Toc75763115)

[5.4 Questions and Answers 31](#_Toc75763116)

[6 Bug Tracking 32](#_Toc75763117)

[6.1 Google Drive 32](#_Toc75763118)

[7 Coding using JSF, Primefaces, GDAO and SQL 33](#_Toc75763119)

[7.1 General Java rules 33](#_Toc75763120)

[7.1.1 Formatter 33](#_Toc75763121)

[7.1.2 Declaring variables, constructors and methods 33](#_Toc75763122)

[7.1.3 Naming 34](#_Toc75763123)

[7.1.4 Comments 34](#_Toc75763124)

[7.1.5 Method principles 34](#_Toc75763125)

[7.1.6 If/While/For brackets 34](#_Toc75763126)

[7.1.7 Alignment 34](#_Toc75763127)

[7.1.8 Spacing 36](#_Toc75763128)

[7.1.9 Constructor declaration 36](#_Toc75763129)

[7.2 General SQL rules 37](#_Toc75763130)

[7.2.1 View Naming 37](#_Toc75763131)

[7.2.2 View creation 38](#_Toc75763132)

[7.2.3 View SQL Formatting 39](#_Toc75763133)

[7.3 YouTube Videos for Web Developers 39](#_Toc75763134)

[8 Layers (db, model, webview) 40](#_Toc75763135)

[8.1 Architecture diagram 40](#_Toc75763136)

[8.2 DB – layer (standardized in Dental, from DB-Analyzer tag 6.2) 41](#_Toc75763137)

[8.2.1 Custom Data classes – basic rules 41](#_Toc75763138)

[8.2.2 Language properties 42](#_Toc75763139)

[8.2.3 Sample data classes 42](#_Toc75763140)

[8.3 Model layer (standardized in dental) 44](#_Toc75763141)

[8.3.1 Custom Provider classes 44](#_Toc75763142)

[8.3.2 Base Provider 45](#_Toc75763143)

[8.3.3 Custom Entity classes 47](#_Toc75763144)

[8.3.4 Entity Report Classes 49](#_Toc75763145)

[8.4 Webview (Ready to use) 52](#_Toc75763146)

[8.4.1 (Important) Referencing and naming between xhtml and controller 52](#_Toc75763147)

[8.4.2 Controllers and xhtml rules 53](#_Toc75763148)

[8.4.3 Outer controller code 53](#_Toc75763149)

[8.4.4 Inner controller code 54](#_Toc75763150)

[8.4.5 Outer controller webpage 54](#_Toc75763151)

[8.4.6 Inner controller webpage (standalone webpage for single program) 55](#_Toc75763152)

[8.4.7 Facet example 56](#_Toc75763153)

[8.4.8 GDAO Statement 56](#_Toc75763154)

[8.4.9 Lazy example 58](#_Toc75763155)

[8.4.10 Lazy Example – upgraded 58](#_Toc75763156)

[8.4.11 Select entity lazy 58](#_Toc75763157)

[8.4.12 Select entity lazy (GDAO statement) 61](#_Toc75763158)

[8.5 Reusable search components 62](#_Toc75763159)

[9 System (application level) parameters management 66](#_Toc75763160)

[9.1 How to use **SystemParametersManager** 66](#_Toc75763161)

[9.1.1 Getting the instance 66](#_Toc75763162)

[9.1.2 Getting value of system paramter 66](#_Toc75763163)

[9.1.3 Changing value of system paramter 66](#_Toc75763164)

[9.1.4 Good practices 66](#_Toc75763165)

[9.1.5 Important rules 67](#_Toc75763166)

[10 Session level parameters 67](#_Toc75763167)

[10.1 Getting ApplicationUser object for currently logged in user 67](#_Toc75763168)

[11 Useful YouTestMe Tutorials 67](#_Toc75763169)

[12 Logging in Java 69](#_Toc75763170)

# Introduction

This document describes how to set up different YouTestMe projects step by step.

Also about learning Sass. First things first Sass saves you a bunch of time you don’t have to type same code again and again… You are able to reuse properties you have already set. But you’ll read more about it later on, next Sass is visually much better than Css.

# Open Issues

| **#** | **Issue** | **Comment** |
| --- | --- | --- |
|  | Which version of Eclipse should be used?  Please provide download link to proper version. | Verified with the latest version of Eclipse:  <https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2019-06/R/eclipse-inst-win64.exe>  See tutorial about setting environment in Eclipse (Section 9.) |

# Mandatory Development Procedure

\youtestmedoc\Procedures\Development Procedures\YTM Development Procedure.docx

# Setting up Classroom2020

After checking out repository, there are three steps that are required in order to set up project:

1. Create DB schema
2. Load data via Batch Loader
3. Set up www\_source project

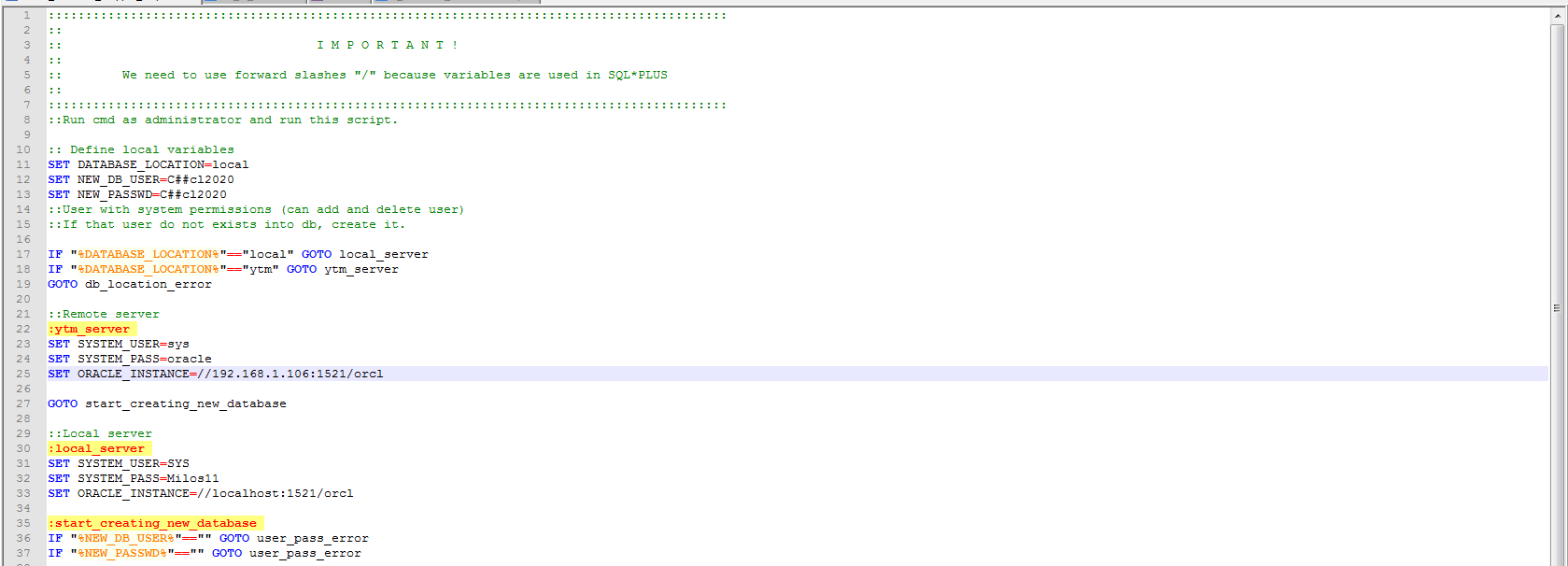
## Create DB schema

Database script that creates Classroom2020 schema can be found on this location:

**youtestme\trunk\db\dbmodel\scripts\create\_database\_wrapper\_script**

### Editing wrapper script

Before you run script, make sure that you set up the following things:



Script has to be edited with personal information:

1. If you are creating DB schema for your local database, it is required to create DB user. For creating a user in SQLPlus, see:

<https://www.youtube.com/watch?v=drpZujwefuA&list=PLNr69myqMnd-oRCI2u8J4AM2jTWhLRafH&index=1> starting from 2:10

1. Set DATABASE\_LOCATION to be either **local** or **ytm**(line 11)
2. Then put you new DB\_USER and DB\_PASSWORD. If DATABASE\_LOCATION = local, make sure that you create a user in SQLPlus for your local database and put credentials here.

(line 12, line 13)

1. If DATABASE\_LOCATION = **local**, then put credentials for system user as well as oracle instance location on line 31, 32, 33. if DATABASE\_LOCATION = **ytm** then put credentials on line 23,24,25
2. Save this file, make note that you should NOT commit it in SVN repository.

### Troubleshoot – editing other files

Script will create all tables and views before putting “Success” message (note that it will put “Success” message regardless of success so make sure that you follow script and read what goes on), however there are some issues that you might encounter while loading so it is recommended to check the following before running script:

1. For someWindows users, there might be a problem with VARCHAR2 larger than 2000 letters. Since Classroom2020 has some fields with 30 000 letters, you have to edit the main **youtestme.sql**file. It can be found on:

**youtestme\trunk\db\dbmodel\scripts\sql\youtestme.sql**

Just search for “30000” and replace given fields with “2000”

1. Error with **ytm\_role**, if you are creating local db schema - you will see this error. In order to avoid it, you have to go to:

**youtestme\trunk\db\dbmodel\scripts\sql\create\_db\_user.sql**

and remove where says: “grant ytm\_role to &newuser; ” (line 20)

### Running script from command prompt

After above steps, if you are using Windows - run it from Command Prompt.In order to run it from command prompt do the following:

1. Open command prompt
2. If SVN repository is not on C: (or wherever Windows is installed), then just go to that folder, for example:

**cd C:\SVN\youtestme\trunk\db\dbmodel\scripts**

Then from there, write **create\_database\_wrapper\_script.bat**

1. If SVN repository is on different drive than Windows, then redirect it before using command from above. For example is SVN is on E:\ drive, and Windows is on C:\ - type **E:** and press enter. Then you can use commands from step 2.

After setting up DB schema via script, you should see tables and views in your database. To do so, you may use SQL Developer.

## Load data via Batch Project

Tutorial for this part can be found in:

**youtestmedoc\trunk\Procedures\Development Procedures\YTM – Creating Test Data Procedure, chapter 8.1**

## Load data from template databases

**Data is maintained in template databases**

**ytm-4-SE-DB-Dev IP: 192.99.24.208**

**Username and password for GC: ytm1, ytm1**

**Username and password for CL2020: ytm2, ytm2**

**Service name: pdytm1**

**Port: 1555**

**Check this document for explanation how to copy this data to your database:**

**\youtestmedoc\Procedures\Development Procedures\YTM Copy DB Data using DB Pump.docx**

## Setting up www\_source project

Please note that development should not be done in “trunk”. See next paragraph for explanation.

If the development and/or documentation repositories have not been yet checked out from the SVN repository, check out entire content of the following:

/youtestme/trunk – contains all the necessary code within included “www\_source” directory as well as some other relevant directories containing scripts, configuration etc…

/youtestmedoc/trunk – contains all the documentation

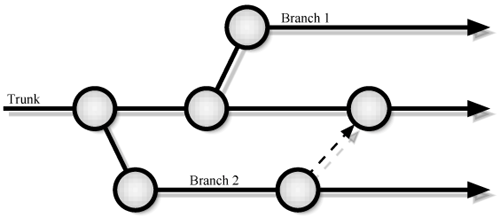
Now after data has been loaded to our database, the last thing to do is to set up the main project and run server from our PC. In order to do so, do the following:

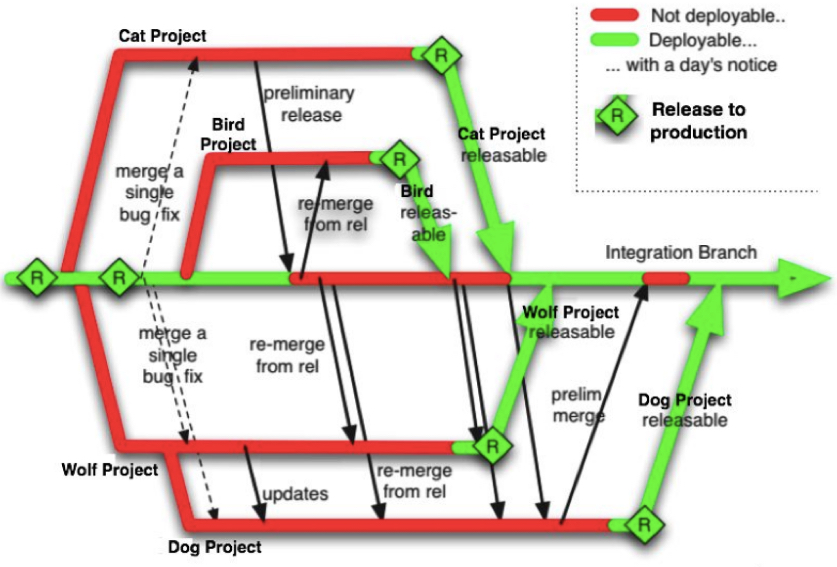
## Using SVN Branches to do Developemnt

Development should not be done in “trunk”. That branck is “integration” branch where we merge our code after testing is completed.

You should always create a branch for your project (usualy from “trunk”) and do development there.

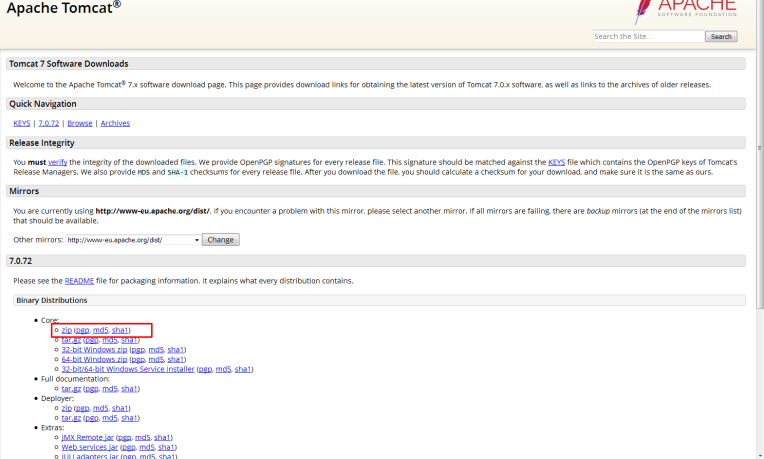
Graphical representation of this process is shown below.





### Download Apache Tomcat 7.0

Go to Tomcat [download](https://tomcat.apache.org/download-70.cgi) page and click on the highlighted link from below:



Note that you need [WinRar](http://winrar.en.softonic.com/) in order to extract it. Extract folder and leave as it is, there is no further installation.

## Import projects into Eclipse

Set workspace to youtestme\trunk\www\_source and import the following projects:

* ytm.db
* ytm.model
* ytm.webview

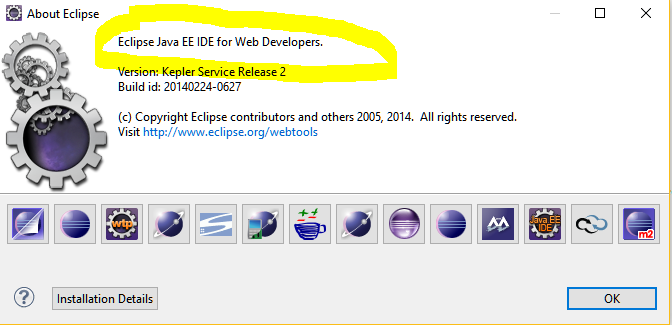
### Adding Tomcat server

After validating projects and building workspace there should be an Error with description:

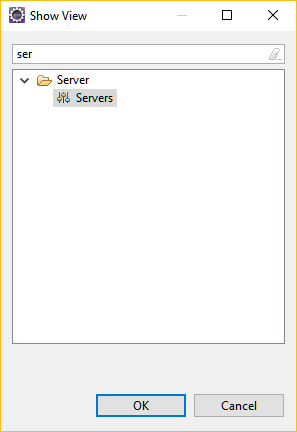
“Target runtime Apache Tomcat v7.0 is not defined.”

It is required to set apache server in order to remove this error. To do so, go to Servers tab and click on the link.

Note: you have to have Eclipse for Web development installed

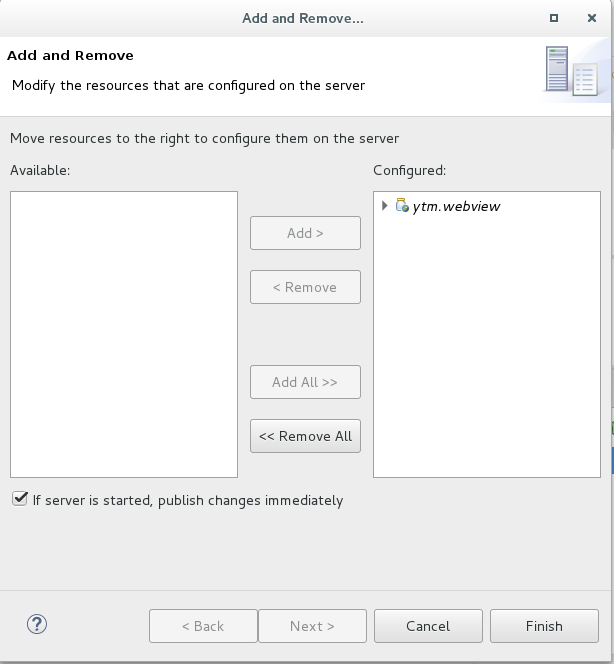


And you need to add tab for Servers:

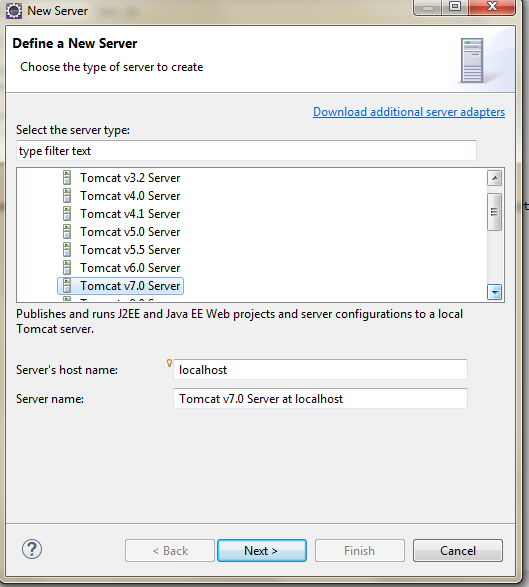


****

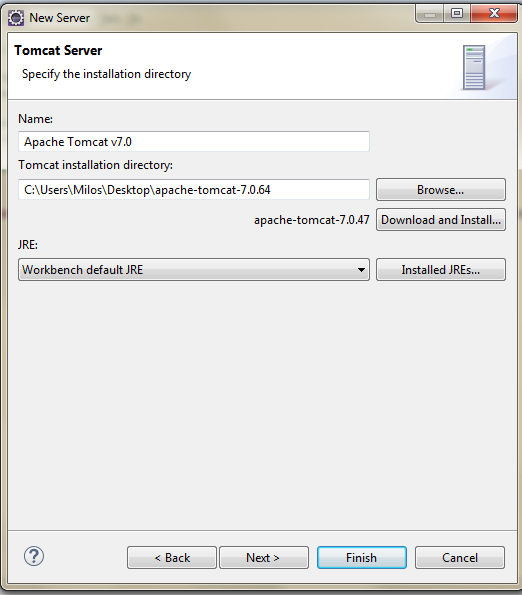
**Add "ytm.webview" to "Configured"**

****

After clicking, scroll up and there will be **Apache** folder, click on it and go to **Tomcat v7.0 Server –** then click next.

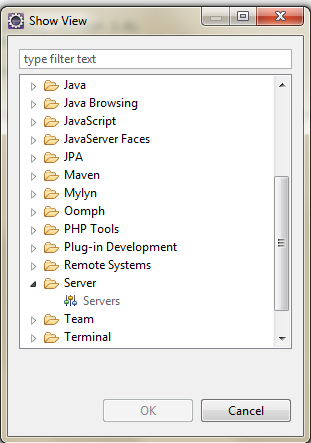


In the next step, click browse and go to give path to your tomcat directory. After that, click **finish**.



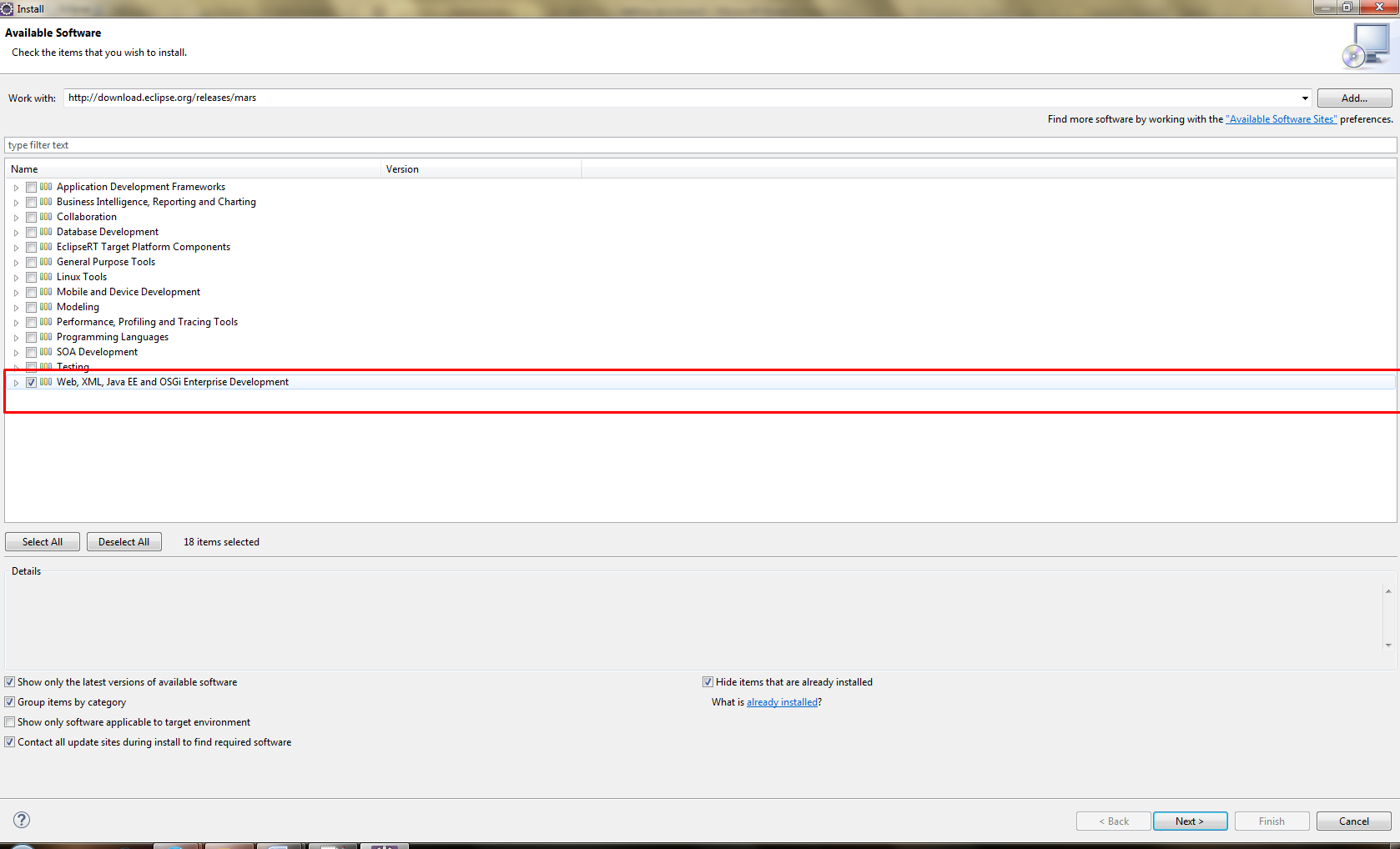
### Troubleshoot: Can’t find server tab

If you cannot find server tab, go to Window/Show View/Other../Server/ and click **Servers**, there should appear servers tab.



However, if there is no Server folder inside Show View, then you have to go to Help/Install new Software

When window opens, click on the dropdown menu and find Mars (or any other new Eclipse version) release.

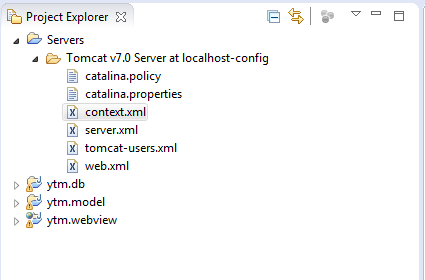


Click on the highlighted and click **Next**. Accept licence agreement, click **Finish** and let Eclipse install this module for you.

Restart Eclipse and find Servers inside Show View.

### Setting up server Context

After successfully setting up tomcat, you will see new folder inside Project Explorer tree.



Now you have to make connection between Tomcat server and Oracle database. To do so, go to Servers/context.xml, click on the source tab and paste this before enclosing </Context>tag :

<Resource name=*"jdbc/UCPPool"* auth=*"Container"*

factory=*"oracle.ucp.jdbc.PoolDataSourceImpl"* type=*"oracle.ucp.jdbc.PoolDataSource"*

description=*"UCP Connection Pool in Tomcat"* connectionFactoryClassName=*"oracle.jdbc.pool.OracleDataSource"*

minPoolSize=*"2"* maxPoolSize=*"50"* inactiveConnectionTimeout=*"20"* user=*"gc\_api"*

password=*"2ytm1"*

url=*"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=192.168.1.8)(PORT=1555))(CONNECT\_DATA=(SERVICE\_NAME=pdytm1)))"*

connectionPoolName=*"UCPPool"* validateConnectionOnBorrow=*"true"*

sqlForValidateConnection=*"select 1 from DUAL"* />

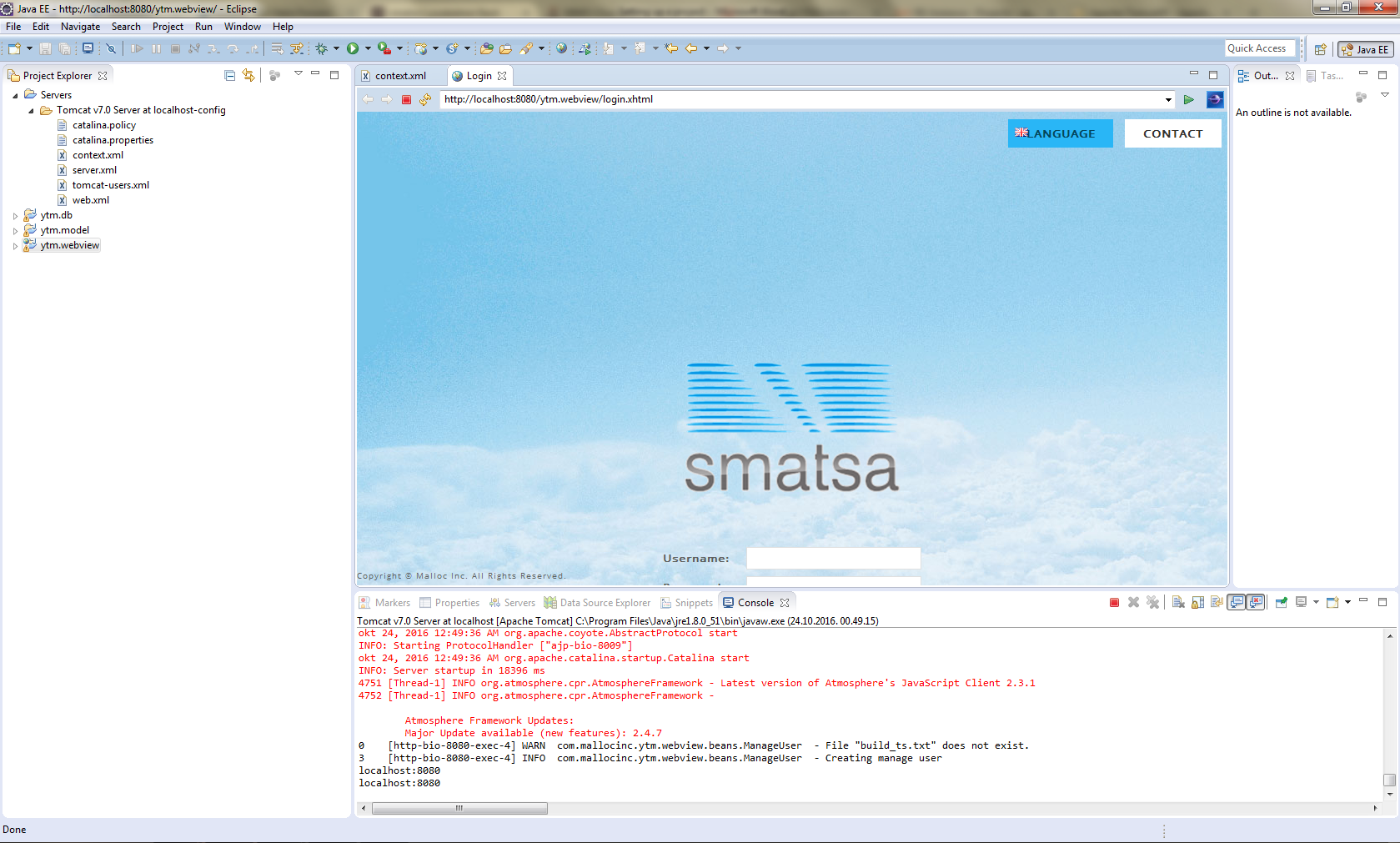
Relevant fields:

* **user**, put database username here
* **password,** put database password here
* **url**, put the relevant database fields into given format – basically you will have to edit HOST, PORT and SERVICE\_NAME parameters

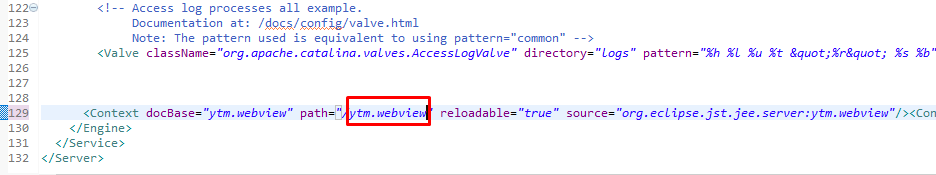
Finally, save this file.

### Running a project

To run the project, right click on ytm.webview folder, and click **Run as/Run on Server.** Another window will appear, click **Finish**. Tomcat will start up and it will take some time for it to run (usually around 30 seconds). Finally, this is the screen that appears when Tomcat is up and running:



**Note: You need to remove ytm.webview path inside server.xml**



**It should be like this:**



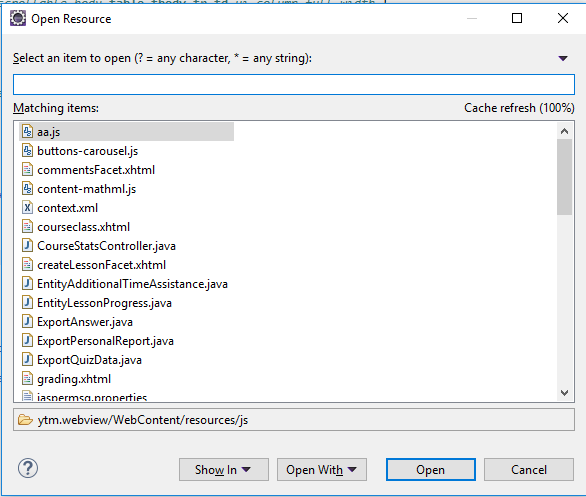
From this point, your Project is up and ready for development.

### Flying through application

Some useful shortcuts to start with:

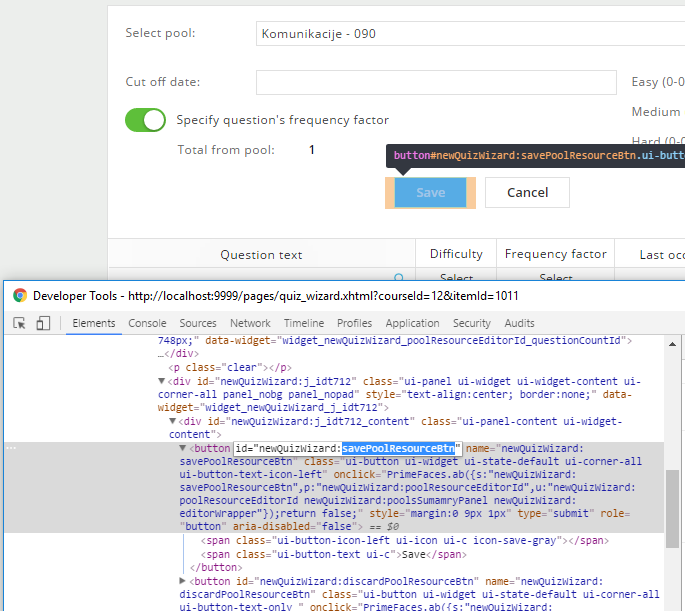
Ctrl + R

Search for any file (see picture below)



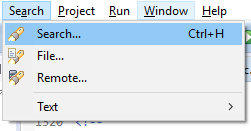
If you want to find specific element there are a lot of nesting with templates in our application and you can easily get lost.

Right click on element you want to find, click inspect element, and copy last part of id (see picture below)

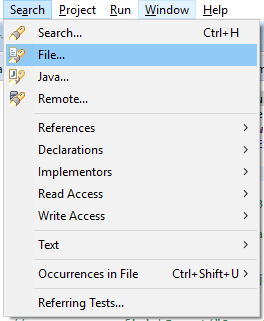


**File search:**

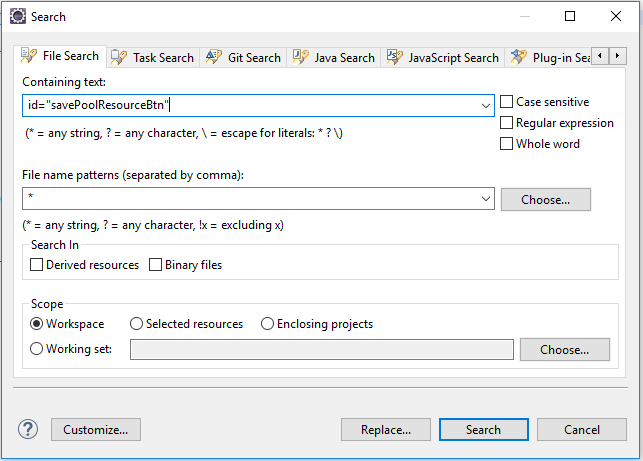
**Search will open differently, if you are in .xhtml, .css or some similar file you will open it as in picture below or Ctrl + H**



**If you are in .java, .xml or some similar file:**



**Be aware that this search can take a while, and use it only when you can’t find something. (On good SSD computer it will last for less than a sec, but that same search can last up to 15 sec on regular one.)**



## Setting up Sass preprocessor

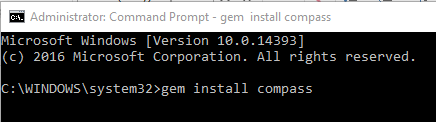
### How to install Sass and Compass

To install Sass first you need to install [Ruby](http://rubyinstaller.org/).   
NOTE: When installing Ruby make sure to add variable to PATH, that can be done automatically by checking that option in installer.

After Ruby installed:

Open your **cmd** and type **gem install compass, and gem install ruby\_gntp.**

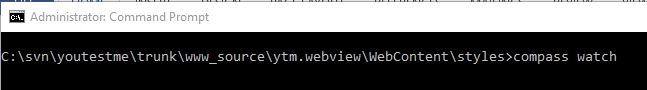
This will automatically install everything you need including Sass.



### ****Setting environment****

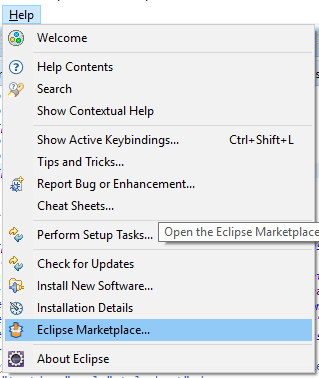
Compass environment is created and it is just necessary to go to our sass directory

**www\_source\ytm.webview\WebContent\styles** and type: **compass watch or start script for it compass\_watch\_dev.bat**

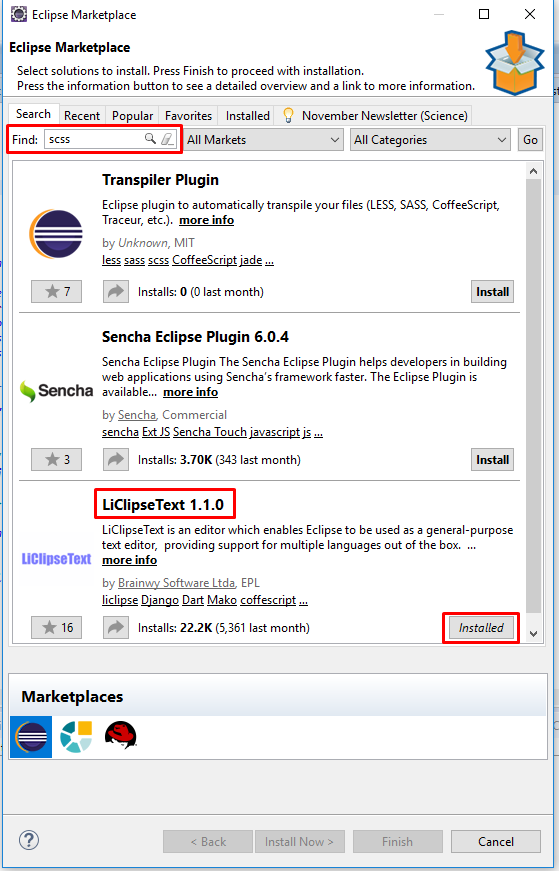


**It is necessary to download editor for sass files and to set your project to refresh automatically when you change your sass file.**

**To install plugin go to Help > Eclipse Marketplace**

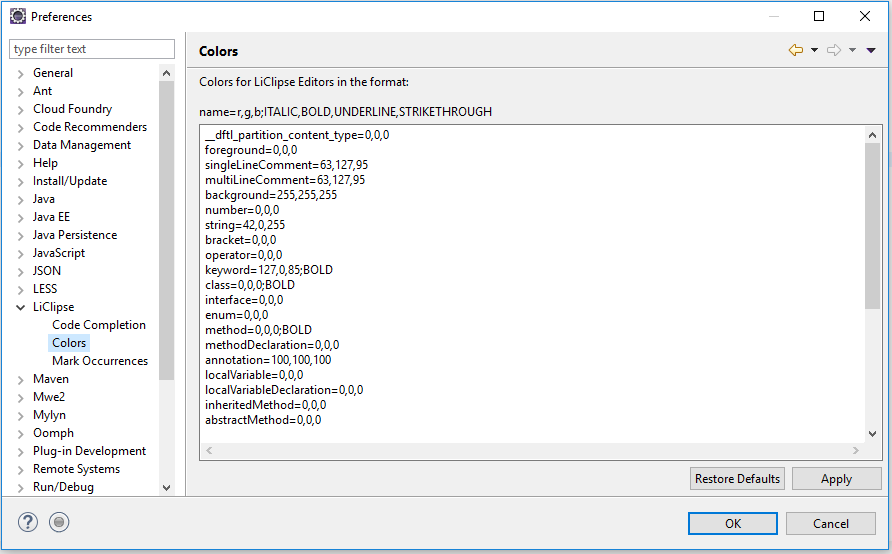


**And search scss**

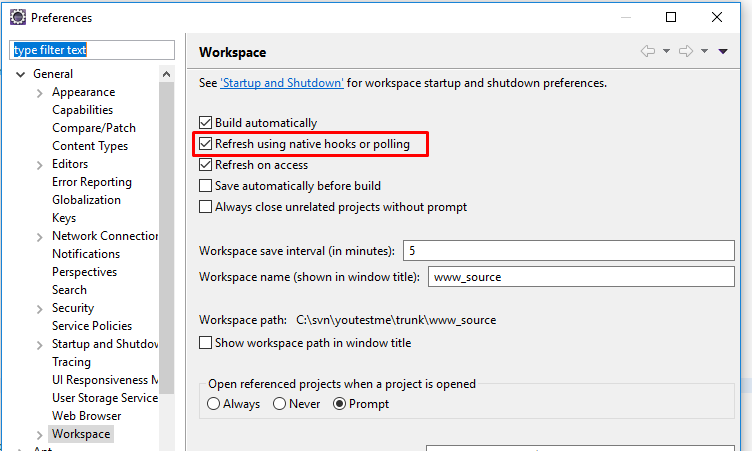


**At this moment only LiClipseText 1.1.0 is editor for .scss files**

**You can also change colors and add some new to this editor.**

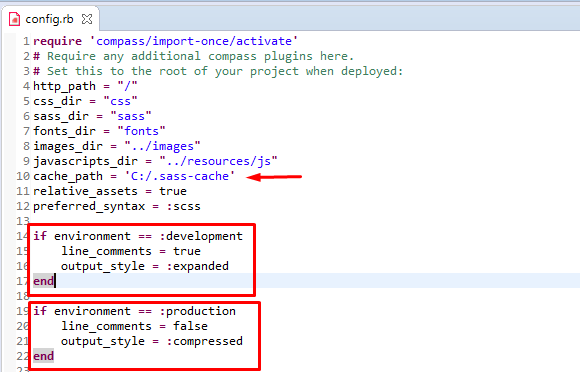


**Make sure that you have enabled Refresh using native hooks or pooling in Window > Preferences > Workspace, only this will put new generated css by compass inside eclipse, otherwise you will need to manually refresh project or open .css file so that it can take the new code.**



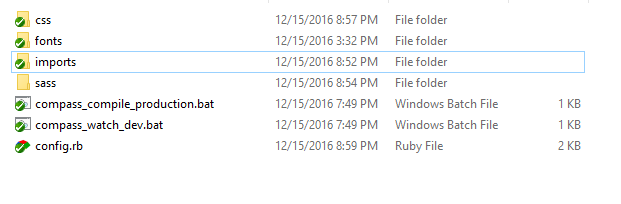
### Compass configuration file

Cache\_path is important to be as short as it can, if cache\_path is to long it won’t compile. NOTE: this is problem only on windows.



Use **compass\_watch\_dev.bat** for development and watching if file is changed,

And **compass\_watch\_production.bat** when you are finished and you want to commit to SVN.



## Sass project creation

If you do not have eclipse, you can type this things in your Command Prompt and get started with all tutorials and with coding:

1. compass create ProjectNameGoesHere
2. cd ProjectNameGoesHere
3. compass watch

Once when you have created a project you can edit your files with “Sublime text” editor. When the project has been created and you opened the files with Sublime text you should watch following tutorial: <https://youtu.be/wz3kElLbEHE>

Follow all the steps and you will see why to use Sass instead of CSS. Things that you need to focus on are:

1. Variables
2. Nesting
3. Importing

**Important thing is that you don’t have to worry about creating CSS file, because Sass file will be automatically converted into CSS. Also the important thing is not to change or write into CSS files!**

**Files should be named like “\_variables.scss”, also any other file which is not styles.scss should be named with underscore first and then the actual name of a file (for example: \_mixins.scss). That’s because these files get imported into styles.scss.**

### **Variables**

Variables enable you to store any custom value you want for either to specify color or to store font, literally everything. You should create a new .scss file to store all variables which you will import in the other .scss files. Next thing is to get known with the way of writing code, here is the example:

**$black: #000000;** -This is the way to store black color under a variable name “black”, you can use it in your styles.scss instead of copying color numbers. Everything you want to have a customized values or properties should be stored into Variables so you can use it over and over again.

First to make other variables to show you how it works.

**$white**: #FFFFFF;

**$text-color-1**: #212121;

**$border-1:** 1px solid #e0e0e0;

What I did here is that I assigned values to this variables and some properties which I can use in styles.scss.

Here is the code example in styles.scss:

@import “compass”;

@import “folderName/variables”; //this is the way how to import files into styles.scss

body{

width: 100%;

color: **$text-color-1;**

}

Instead to search for color number you can easily say text-color-1 and go on. More about variables on the links that I mentioned before.

### Nesting

When writing HTML you've probably noticed that it has a clear nested and visual hierarchy. CSS, on the other hand, doesn't.

Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. Be aware that overly nested rules will result in over-qualified CSS that could prove hard to maintain and is generally considered bad practice.

Nesting styles is simple enough. You just enclose a selector (or selectors) inside the curly braces of another selector. There is really no limit to the amount of levels deep that you can nest elements. You can write whatever element within any element you want to style. Here is the example of nesting:

**.left-sidebar-links{**

**a{**

**display: block;**

**color: $lightBlue;**

**opacity: .7;**

**&: hover{**

**opacity: 1;**

**}**

**}**

**}**

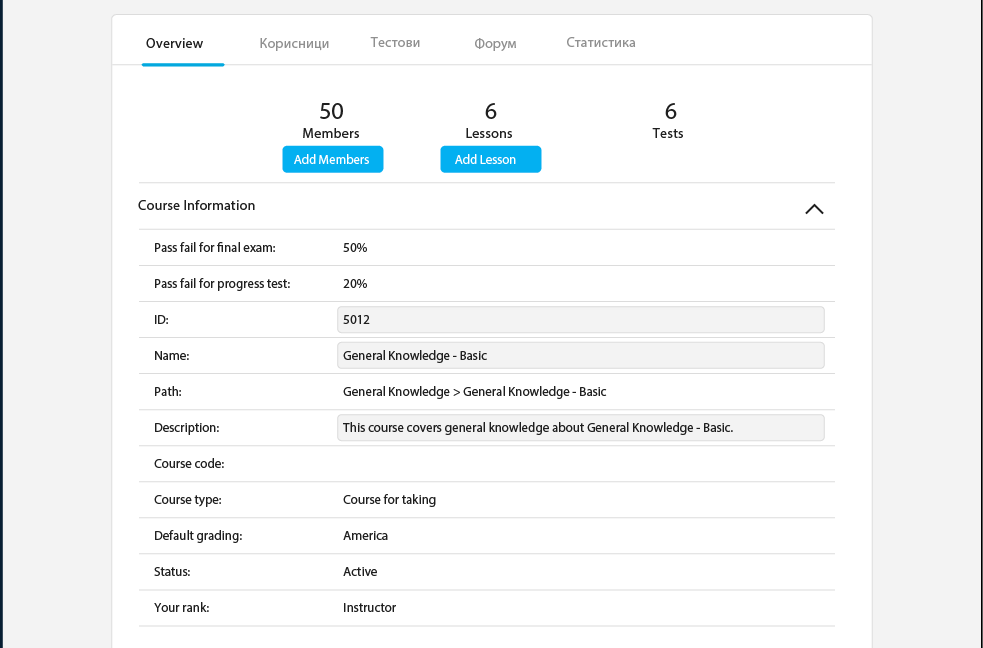
It allows you to modify parts that you want to change, but to maintain simplicity. It’s just the example, you can learn more on the links and tutorials.

### Compass watch

You use compass watch to see where you have errors in your code and to fix them easily. When you enabled compass watch at the beginning it is automatically updating with every save and compass is displaying your errors (line number, and what type of error it is).

### Practice Sass

For practicing it is good to first make a simple page in HTML and CSS. Containing just tr and td, 3 td contain textInput. You have to change input background-color, borders for the whole table, and to make just border-bottom. Table contain 2 columns, one for labels and the other for outputText. You should have 2 files, one is index.html and the other one is styles.css. You do not make a project in CMD for this. You can see table example here:



You will also get a illustrator file, you can find it at location: (youtestmedoc\trunk\Dev Tutorials\Interns\Sass Practice.ai) to examine colors, input width, height, etc.

When you finish with .css, you will have to make a new Sass project and to convert .css into .sass manually and then to compare files and see the differences.

## GDAO Project

Checkout this repository: <https://svn.youtestme.com/dba/trunk>

Look at this tutorial for guide: Z:\Development\Tutorials\DB Analyzer , and this one <https://svn.youtestme.com/youtestmedoc1/trunk/Procedures/Development%20Procedures/Setting%20up%20DBA%20in%20Eclipse%20and%20generating%20code.docx>.

## Build and Deployment Procedures

\youtestmedoc\trunk\Procedures\Development Procedures\YTM Application Build Procedure.docx

## Unix machines in Montreal Cloud

Login information:

-username: **ytmadmin**

-password: **1!@Malloc2**

Switch on corresponding ytm user:

- **su - ytm1** (or another ytm user)

-password: **2ytm1**

# Tagging the Code

SVN tags will be used to control version of the code for purposes such as:

* testing
* delivery to the client
* troubleshooting and support

## Naming convention

Format:

***[application\_identifier]-[major\_version].[minor\_version].[bug\_fixes][release identifier]***

Examples:

GC-3.0.1r

CL-1.1.0s

BLI-1.0.0

RES-2.0.0

CLDB-3.0.1

GCDB-1.0.6s

|  |  |
| --- | --- |
| **Element** | **Description** |
| Application identifier | Alphanumeric designator for application or module, for example:  GC - Get Certified  CL - Classroom 2020  RES - Resources  BLI - Batch Loader of initial data  CLDB - Classroom database  GCDB- Get certified database  Note: no "-" character is allowed in this designator |
| Major version | Integer indicating major release of the application |
| Minor version | Integer indicating that application has new or changed features |
| Bug fixes | Integer indicates that application has only bug fixes |
| Release identifier | Optional indicator showing if tag is ready us release, stabile, beta or none:   * "r" - released to the client or ready for release * "b" - beta release, tested and stabile version suitable to be given to early adopters for a trial and experimenting * "s" - stabile, tested version of the code, can be used for demos * if no indicator is present than this tag is intermediary tag used for testing or marking the code for whatever practical reasons |

Important - each tag should have in the SVN comments field:

1. List of new features or bugs fixed
2. Any other useful information to the QA are development team

## Tags location

Tags can be organized in subdirectories. There is no limit in number and depth of directories.

This is proposal and opened for discussion.

| **Application** | **Tag location** |
| --- | --- |
| Classroom 2020 batch - SMATSA | http://svn.mallocinc.com/youtestme/tags/SMATSA/ |
| Classroom 2020 - Mainstream | http://svn.mallocinc.com/youtestme/tags/CL/ |
| Classroom 2020 - Database | http://svn.mallocinc.com/youtestme/tags/DB/ |
| Classroom 2020 - Batch loader | http://svn.mallocinc.com/youtestme/tags/BL/ |
| Get Certified - Mainstream | http://svn.mallocinc.com/getcertified/tags/GC/ |
| Get Certified - Database | http://svn.mallocinc.com/getcertified/tags/DB/ |
| Get Certified - Batch Loader | http://svn.mallocinc.com/getcertified/tags/BL/ |

## Code Compatibility

Bear in mind that YouTestMe system is build and delivered from several components (tags) - for example:

* application tag such as CL-3.0.1r
* "res" tag (tomcat and scripts)
* database model
* other utility programs and scripts such as Batch loader

We need to keep track of compatibility of the tags. In below example subsequent release of the YTM Classroom 2020 has only a bug fix in the database (everything) else is the same:

| **System version**  **(no tag)** | **Application**  **(tag)** | **Resources**  **(tag)** | **Database**  **(tag)** | **Utility**  **(tag)** |
| --- | --- | --- | --- | --- |
| YTMCL-1.0.1r | CL-3.0.1r | RES-1.0.0r | CLDB-1.0.6s | BLI- 1.0.0r |
| YTMCL-1.0.2r | CL-3.0.1r | RES-1.0.0r | CLDB-1.0.7s | BLI- 1.0.0r |

## Questions and Answers

**Question**: What is System version and why it does not have a tag?

**Answer**: That is the version of the YouTestMe software. It consists of components like: web application, database, scripts, libraries, software (Tomcat). Each of the components has its own version and table above ensures that we use compatible versions in software release. For example, database model has to be tagged since anybody can change anything in the model at any time and not knowing that with that change makes web application currently under testing incompatible or useless.

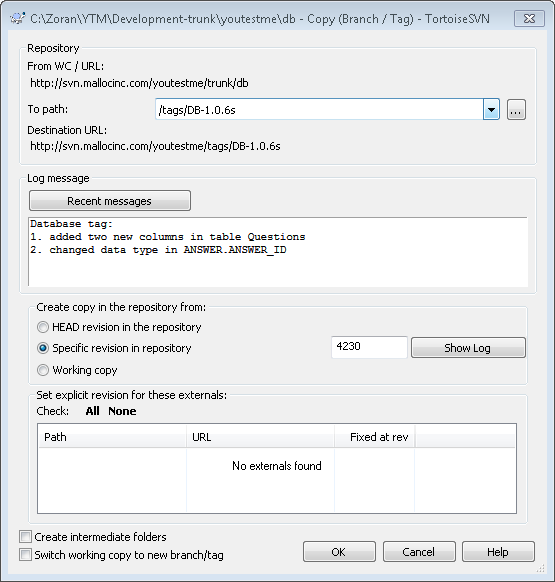
**Question**: Why database has its own tag?

**Answer**: If we tag entire trunk we are assuming that somehow database model is kept in sync with our code automatically, meaning that we are absolutely aware about all changes to the database model and 100% confident about their impact to application. That is very risky in small teams and impossible in larger teams.

Alternative solution is to do all development in the branch(es) where development of certain releases are clearly separated.

Note:

* each project has table line one shown above in corresponding project directory, for example:   
  \youtestmedoc\Projects\YTM Classroom 2020\YTM CL2020 - Compatibility List.xlsx
* Last three columns may be pretty much static in stabile environment
* Above table should be sufficient to anybody to recreate entire YTM system to exact state how it was delivered to the client
* tags can be created for subset of directories in repository, for example:



# Bug Tracking

## Google Drive

<https://docs.google.com/spreadsheets/d/1h5PWaXsRqT49AqI-h5Pb3KeRqMu2PvS2e-fY9Fw8UE8/edit?pref=2&pli=1#gid=1762665861>

# Coding using JSF, Primefaces, GDAO and SQL

## General Java rules

These rules from below will be used to standardize code and make it cleaner and easier to read. Please note that our current codebase does not follow all of these concepts, so whenever you see a possibility to use these concepts, **make sure to use them**. Each daily small change to our code is good and will eventually make our code much more readable and easier to work with.

In addition to this, take your time to read **Clean Code by Bob Martin**, it is very helpful book. Ask for the access on the shared drive within YouTestMe local network, it is located there on this path:

Z:\Development\Tutorials\Helpful Java books

, there are also other books there that will be very benefitial to your further career as Java developer so feel free to use them.

### Formatter

* **Use formatter** to format each class. Formatter is located here: <http://svn.mallocinc.com/youtestme/trunk/cfg/Eclipse/ytm_eclipse_java_code_formatting.xml>
* If you find some code that is badly formatted using formatter, send a bug to older developers so they can make a revision and eventual changes to formatter

### Declaring variables, constructors and methods

* When creating class, excluding the import part that are at the very beginning, this should be the order of class components:
  + Final global variables/Enums
  + Public global variables
  + Private global variables
  + Constructor(s)
  + Public method
  + Private Method that is called by public method from above (If necessary)
  + Public method
  + On the bottom of the class, put getters and setters of global variables (If necessary)

### Naming

* Use lower camel case naming when creating a method or an instance of particular object and upper camel case when creating a class or interface
* Classes and class instances should contain a **noun** within their name
* Methods should contain a **verb** within their name
* Evade buzz words, when creating code **always try to create code that could be easily read by other developers**

### Comments

* Use comments when really necessary, otherwise **try to evade it**. Meaningful naming and design should speak for itself instead of comments. Generally, comments are hard to maintain and if there is no other option **make sure that comments are always up to date**.
* In general, comments should give you information **why** does this code exist, **not what** it does – this should be explained by the code itself.
* To write a comment, type **/\*** and **enter**

### Method principles

* Method name should include a meaning **verb** that explains what that particular method does
* Avoid having big methods, rather create a method that would be max size of ~20 lines. If method has a lot of logic, separate logic in multiple private methods
* Create private methods within class and **always** put them below related public method(s), don’t put them on the bottom of the class – this way it makes other developers much easier to read your code.
* Methods should **NOT** have more than 3 parameters – if there are more than 3 already, find a way to wrap parameters into **meaningful** object
* Each method should have one intent – do not use **and** in the method naming. For example don’t name doThis**And**That() method, instead create a name for that method that should represent wrapped meaning for both this and that.

### If/While/For brackets

* Make sure that there is a bracket **{ }** for each **if, while or for** statement, **regardless of whether it contains one line of code or not** , same goes for the **else** part.

**Example:**

If(boolean) {

//Do something

}

else {

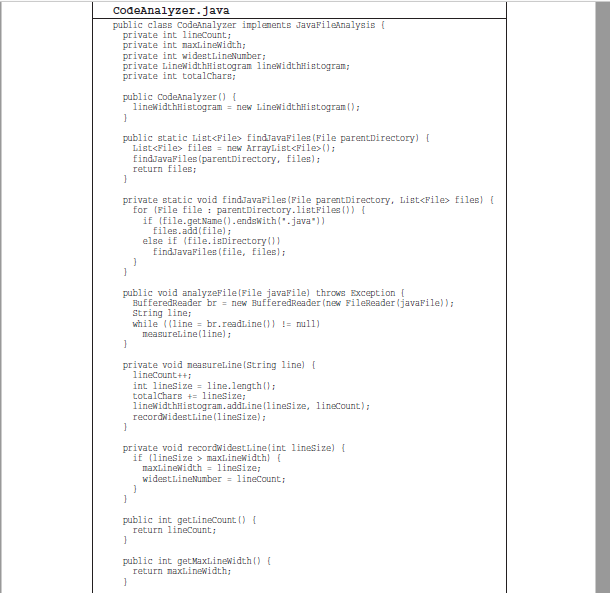
//Do something else

}

* **Note:** Since we have started using Java 8, please take a look at **streams API** and its implementation and replacement of **for each statement.** It also has other benefits so make sure to check them.

### Alignment

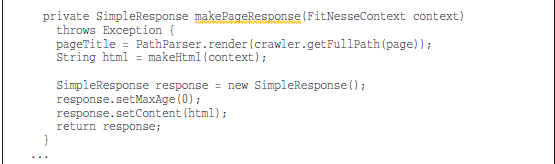
* Alignment should be mostly fixed with formatter, however the general idea is to have the methods with the same alignment, as well as the bracketed code. This is an example of good alignment:



### Spacing

* Separate methods with empty line
* Dense lines of code should be related to one another – if there is a close association within multiple lines of code, then there is no need for spacing. Same goes with variable declaration as well.

**Example:**



Notice how response part is separated from the other parts.

### Constructor declaration

Sometimes there is a need to have multiple constructors for different purposes within the same class, which leads to a confusion – in what situation is which constructor being called? For that reason, a good practice is to **declare private constructors and public static methods that return the object using those private constructors.** This way we can give meaningful name to a reference to constructor and prevent confusion. This is an example:



## General SQL rules

Same as with Java rules, in order to make our SQL scripts more readable, it is necessary to follow these rules. Please note that some of these rules are not fully followed at this moment, so when you find an SQL file that can be edited, **make sure that you edit** those files by following the rules.

### View Naming

* When naming views, make sure that it contains **V** prefix and **underscore** instead of space
* Generally, the next noun that comes after “V\_” should be the **root table** from which we create view, for example if we start FROM USER (Or from another view that has root table USER), then created view should start with V\_USER. What comes after V\_USER is up to developer, but it should contain some meaningful name for the intent of created view. For example, V\_USER\_COURSE\_SCORE is meaningful and self-explanatory name.

### View creation

* Make sure that you have put new view into appropriate position in **create\_views.sql** script, also make sure that view is put and commit in the right place
* When view is created in SQLDeveloper, it adds the following in **CREATE** statement:

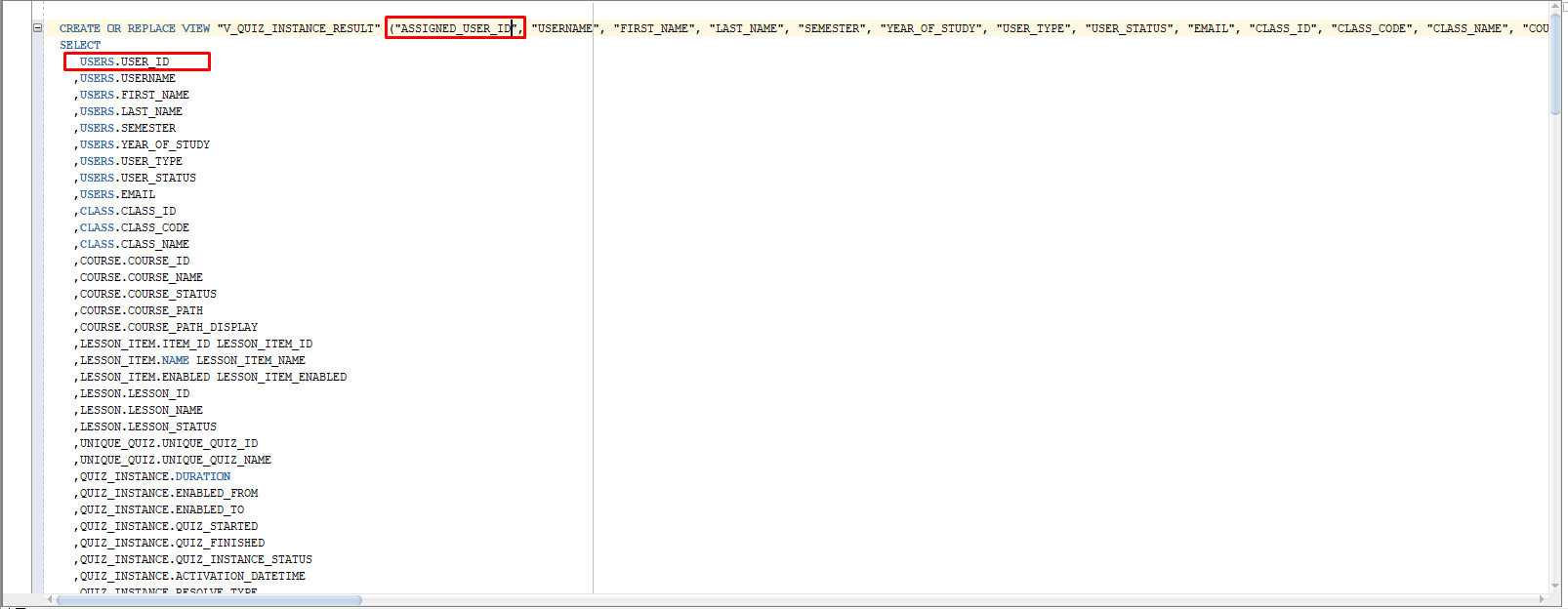
CREATE OR REPLACE **FORCE EDITIONABLE** VIEW **"YTM4"."**V\_QUIZ\_INSTANCE\_RESULT" **(LIST OF COLUMNS)** AS

**NEVER** commit a view with bolded declarations, this is how this part of view should look like:

CREATE OR REPLACE VIEW **"**V\_QUIZ\_INSTANCE\_RESULT" AS

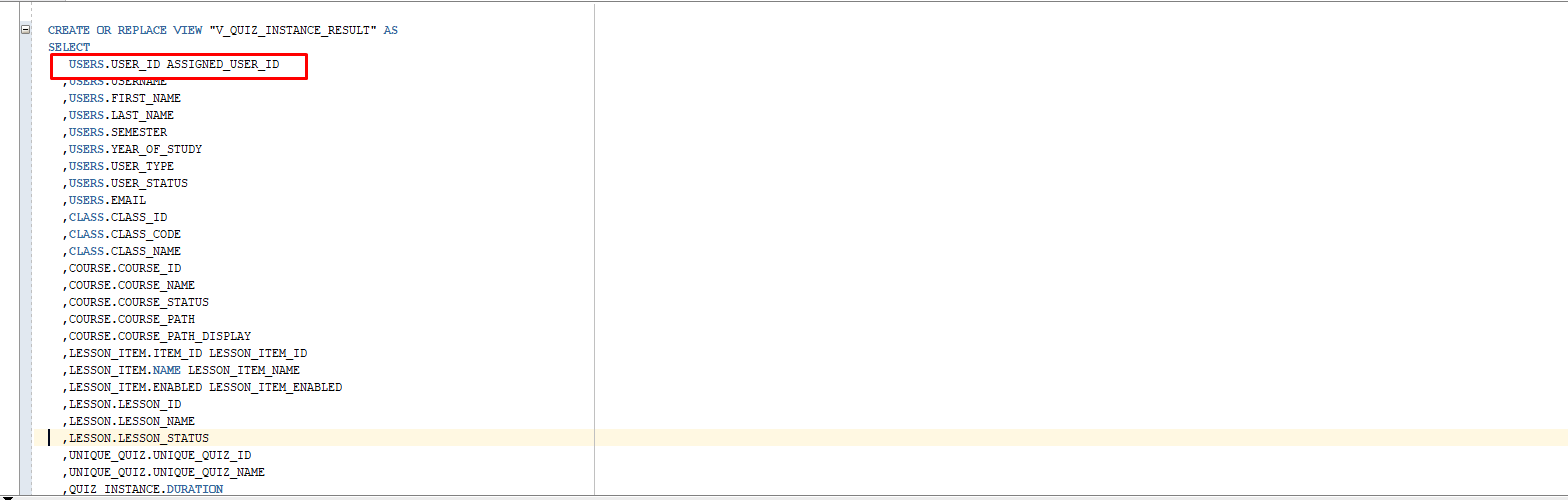
* Put aliases when declaring columns in SELECT statement if necessary, **DON’T** put aliases in brackets of CREATE statement. These are the examples:

**BAD EXAMPLE:**



This usually leads to many problems when editing view. If a person that edits the view deletes stuff from brackets, he has a hard time to put alias names as they were before. Even by not deleting brackets, it is much harder to keep track of aliases this way because we have to compare the order of brackets and declared columns in SELECT statement.

**GOOD EXAMPLE:**



By not putting them into brackets but next to related column, we make sure that each alias is easier to edit and read.

### View SQL Formatting

SQLFormatter tool is used for formatting views SQL. The tool runs from batch script located in db\dbmodel-postgres folder of GetCertified project. The script formats all \*.sql files in the db\dbmodel-postgres\views folder and its subfolders that starts with ‘V\_\*’ according to the naming convention for the views files. Whenver the existing view is changed or the new one is added, this script has to be ran to format all changes.

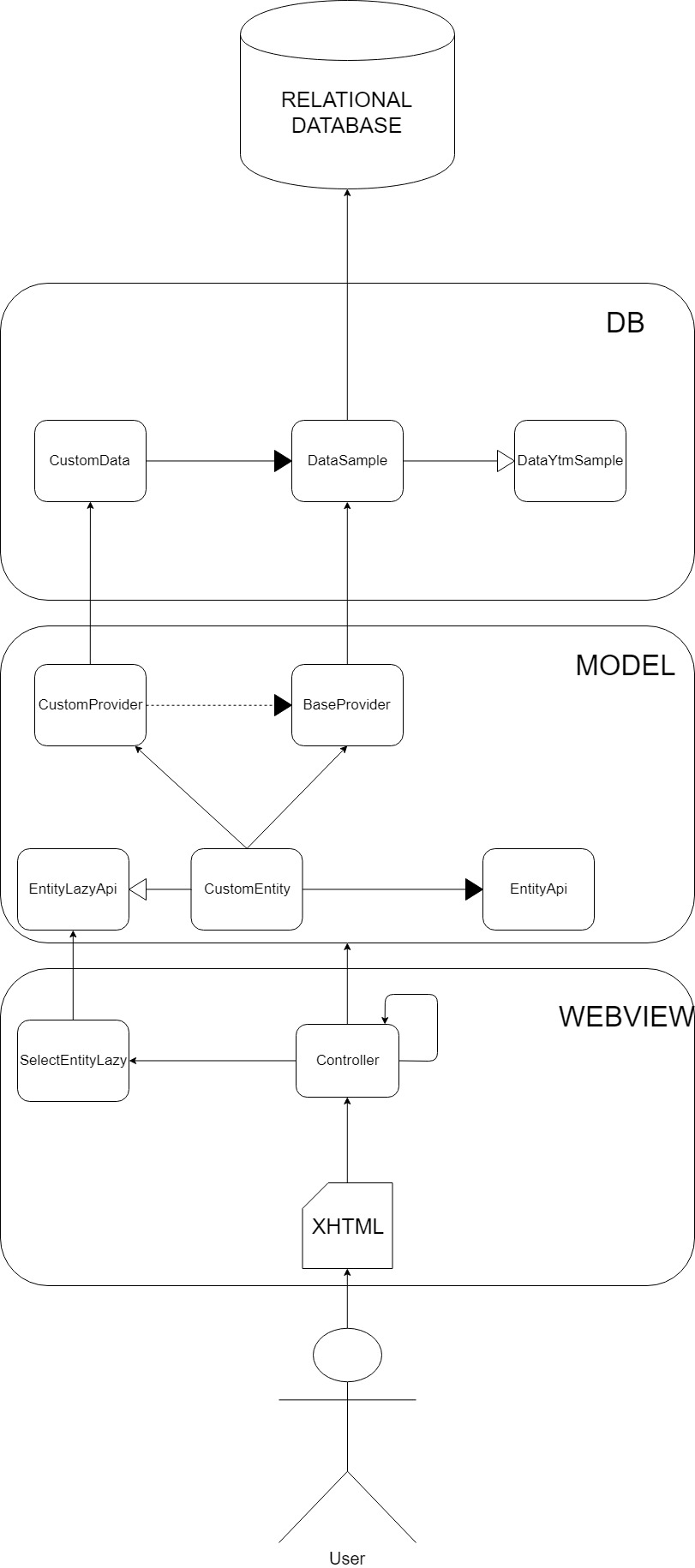
## YouTube Videos for Web Developers

|  |  |
| --- | --- |
| **Name** | Description |
| [**Model** and **logic** layer](https://www.youtube.com/watch?v=bgwaYONDKTY&index=1&list=PLNr69myqMnd8b_36ZX-NP34q2_FmP5Sun) |  |
| [**Webview** layer and example](https://www.youtube.com/watch?v=snAY1K7fLM8&index=2&list=PLNr69myqMnd8b_36ZX-NP34q2_FmP5Sun) |  |
| [2018-Gdao standardized web development](https://www.youtube.com/playlist?list=PLNr69myqMnd-GynnGY2TiyEZvVxywGfjd) |  |
| Add menu item | Add menu item and link it to the screen |
| Create standard YTM screen | Create stadard YTM screen that is a startinh point to start adding elements. Explan usage of CSS SAAS franework |
| Create table | Create standard table and link it to database table. Also explain satndard features line button that has action on the table. |
| Create Tabs | Create tabs panel with some standar features |

# Layers (db, model, webview)

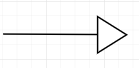
## Architecture diagram

This diagram represents YTM architecture:



Legend:

 represents „is a“ relation (extends)

 represents „has a“ relation (implements)

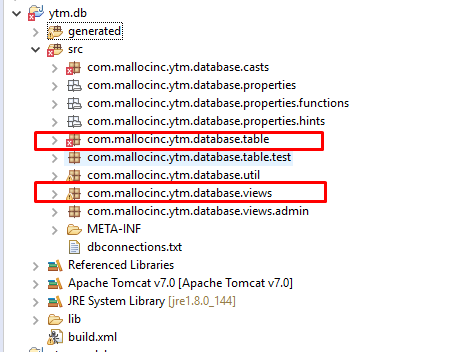
 represents „is a but not necessarily“ relation (can extend but not necessarily)

 represents „uses“ relation

## DB – layer (standardized in Dental, from DB-Analyzer tag 6.2)

### Custom Data classes – basic rules

These classes are used for performing direct operations on the database. They are located here:



**Note**: Since there are **sample data classes** that contain methods for the most commonly used operations on the database, **data classes should be created only if there is a need for a custom method** that is not defined within the sample class.

Package “**table**” contains all data classes related to tables in the Database while, “**views**” contains all view based data classes. Ideally, these Data classes are being called by provider methods from the “model” sub project, mostly by **BaseProvider** class – but there are some exceptions to this rule.

*General rules:*

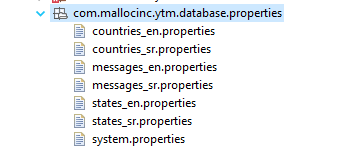
1. Each Data class should extend from Sample data class (chapter „Sample data classes“)
2. Every class responds to its object in database.
3. Never obtain connection in these classes. Connections are passed from providers and also every operation done on Connection objects should be done in provider classes. That would mean that you should **never use methods of Connection object inside Data classes** – just pass them as a parameter where necessary.
4. **Use batch commands where possible**. Batch commands are faster compared to iterating through Objects and inserting them one by one. Take a look at the examples for batch commands (batchInsert and batchUpdateTo). These are located in the sample data classes.
5. Each Data class method **has to throw GDAOException**.

*Naming rules:*

1. Data classes should always start with “**Data**” prefix, and if VIEW it should have a “**V**” prefix. For example:
   * DataCourse (Table)
   * DataVUserFullInfo (View)
2. **Use low level naming** – don’t use abstract method names. Since these Data classes will be called for various modules, they should be called as “technical” as possible. For example:
   * **Good**:getUserByUserId
   * **Bad**: getUserForLogin (It does not necessarily mean that this method will be called just for login method)

### Language properties

Language properties are located inside the following package:

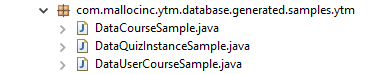


Currently we have English (en) and Serbian (sr) versions of messages and for each new message - **both of these files have to be edited**. These messages are accessed and displayed when required and every type of descriptive text that is visible inside web application should be stored in these property files.

That means that you should **never hardcode messages** – always use property files.

### Sample data classes

These data classes are automatically generated by GDAO and stored inside:



Sample data classes implement the interface called **YtmDataSample** that contains all commonly used methods within the YTM application.

In case of a new custom method, when creating **Data class** for new table or view, it **should extend the sample class**, so you can already have the following methods ready

These are the methods that are currently being implemented within Sample data classes:

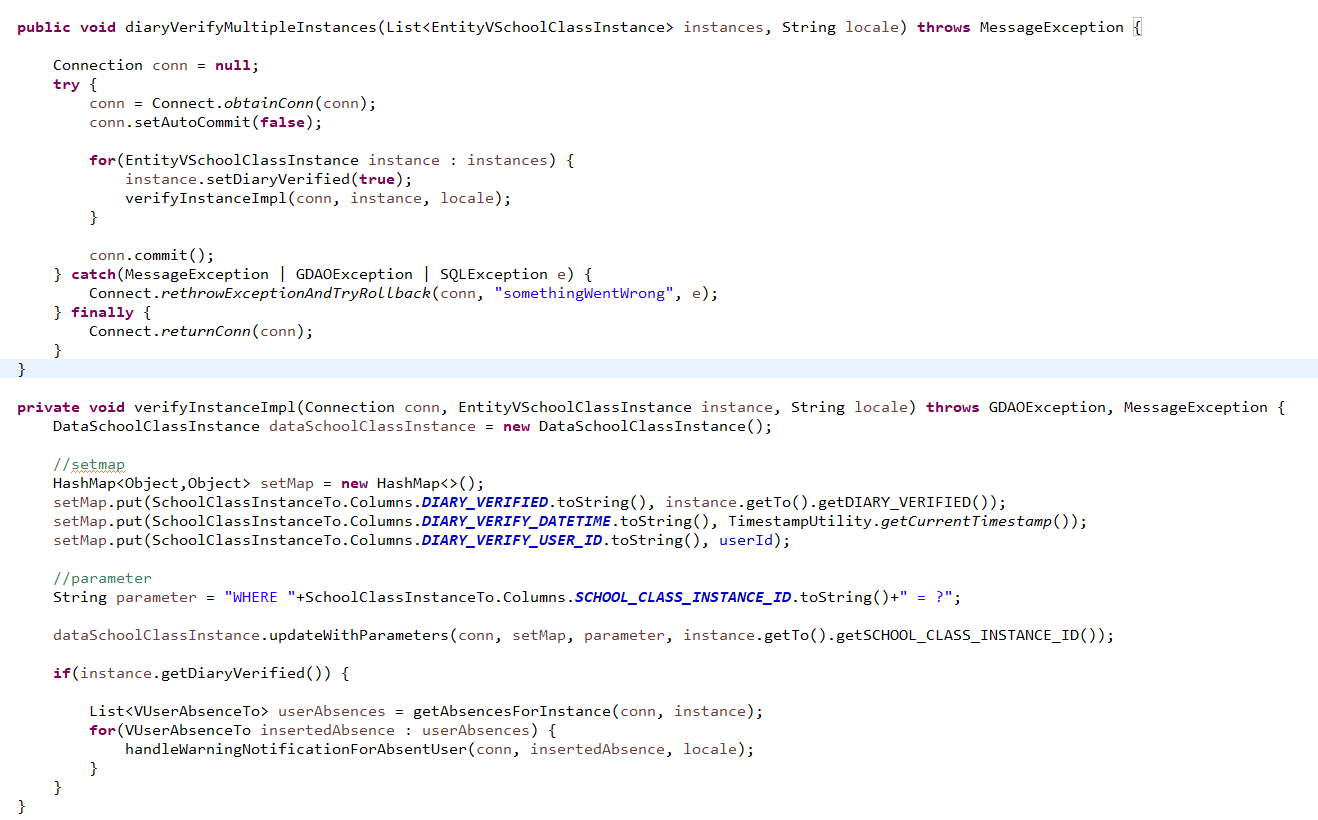
* insertTo(Connection, UsersTo)
* batchInsert(Connection, ArrayList<UsersTo>)
* updateTo(Connection, UsersTo)
* batchUpdateTo(Connection, ArrayList<UsersTo>)
* updateWithParameters(Connection, HashMap<Object, Object>, String)
* deleteByColumnId(Connection, BigDecimal)
* getRecordsForPaging(Connection, int, int, String, String, Map<String, Object>)
* getRecordsFilterCount(Connection, Map<String, Object>)
* getSingleRecordByParameter(Connection, String)
* getRecordsByParameter(Connection, String)
* getSingleRecordWithParameters(Connection, HashMap<Object, Object>)
* getMultipleRecordsWithParameters(Connection, HashMap<Object, Object>)
* selectUsingQuery(Connection, String)
* deleteUsingQuery(Connection, String)
* countUsingQuery(Connection, String)
* selectUsingQuery(Connection, String, Object...)
* deleteUsingQuery(Connection, String, Object...)
* countUsingQuery(Connection, String, Object...)
* updateWithParameters(Connection, HashMap<Object, Object>, String, Object...)
* getTabInstance(Connection)

## Model layer (standardized in dental)

### Custom Provider classes

**NOTE:** After the implementation of BaseProvider, this class is much less used since all basic operations can and should be using BaseProvider + GdaoStatement classes. When it comes to multiple transactions that **have to be atomic**, Providers come into good use. An **atomic transaction** is an *indivisible* and *irreducible* set of database operations such that either *all* occur, or *nothing* occurs**.** **After refactoring, provider classes should contain methods that contain atomic transactions, as well as custom methods that are not covered by the BaseProvider.**

This is an example of a method that contains **atomic transaction**:



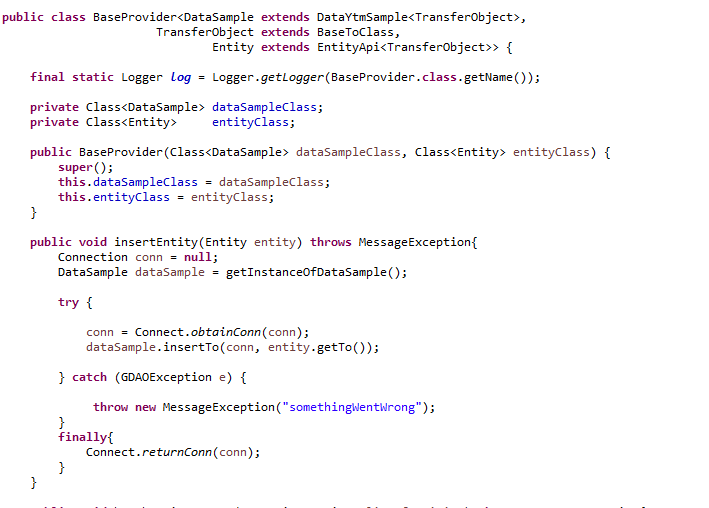
Notice the Connection object – we first instantiate it and set **auto commit to false** (since it is true by default). Then after all the operations in the database, we commit the changes if nothing went wrong, otherwise throw an exception. Finally, we must **return the connection** since it has done its job for this situation, otherwise the connection will remain open and will eventually clog the database.

These are the rules for the provider method:

1. Every provider suits a certain DB object or relation, therefore we can have CourseProvider (methods from DataCourse, DataVCourse ...) , CourseClassProvider (DataVClassCourse, DataCourseClass)
2. Provider method that calls methods from dataClasses must not call any other provider method that calls dataClasses. (Must not create connection twice without closing it)
3. Provider methods that work directly with data classes should return either void or (after refactoring) Entities.
4. Public Provider methods should throw only Message Exception. Currently, there are situations when GDAO is being returned and this should be refactored.

### Base Provider

BaseProvider is a generic class that represents the **wrapper class for Data class on the model layer**. It wraps all the methods from DataYtmSample interface that Data class implements. In the provider’s methods, a connection with the base is taken and Transfer objects are encapsulated in Entities. The picture below shows the part of the Base Provider class definition. Class is parametrized with three other classes (DataSample, TransferObject and Entity). Also, from the definition of insert method, it can be seen that the communication is switched from Transfer objects to Entities and vice versa.

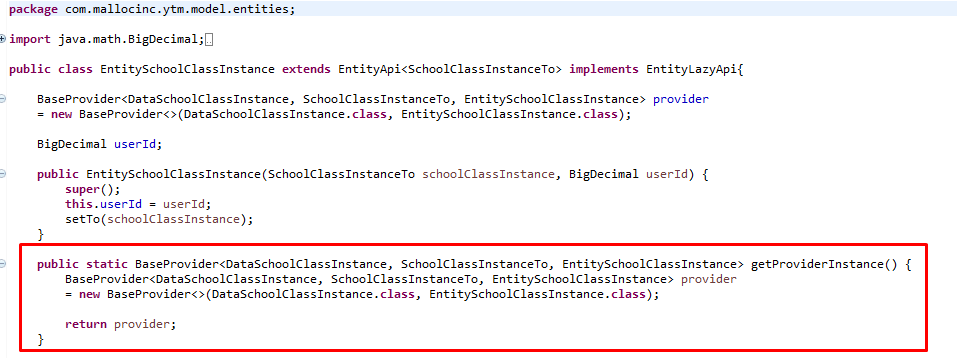


The code below shows how the BaseProvider object is instantiated using the appropriate parameters with. Constructor of the BaseProvider class takes two parameters which represent the class literals of DataSample and Entity class. These parameters are necessary for dynamic creation of Entity and DataSample objects in generic BaseProvider class.



In addition to this, **it is highly advised to create public static method within entity that returns an instance of BaseProvider class** that is related to that particular entity. This way, it is much easier and cleaner to manipulate the entity from the controller classes.

This is an example:



### Custom Entity classes

1. Entity classes are used for logical wrapping of transfer objects so for a transfer object ex. QuizDefinitionTo there can be multiple Entity wrappers:
   * EntityQuizTaker
   * EntityQuizManager
   * EntityQuizInformer
2. Entity classes should be extended from EntityApi class.
3. Each entity class should implement Entity interface template which contains:
   * Leading transfer object – for example QuizDefinitionTo named **to.** That is the only place where TransferObjects are referenced as fields
   * Side related data consisted of other entities for example List of all EntityQuestions, EntityUser (as manager etc.)
   * insert() - usually calls provider method or methods
   * update()- usually calls provider method or methods
   * delete()- usually calls provider method or methods
   * refreshRelatedData() – fetches data that is not initially needed or shown on page
   * Never put logic in any getter!
   * Always fetch only data that is needed for manipulation
   * There can be other methods in entities that perform specific actions
   * The only entity methods that can be called from xhtml page are getter methods for displaying data. All other methods **MUST** be wrapped with controller methods.

Here is a code for EntityApi class that every EntityClass should extend, and a sample EntityClass.

**API**

**package** com.mallocinc.ytm.model.entities.api;

**public** **abstract** **class** EntityApi<T> {

**private** T to;

/\*\*put related entities and entity lists here if needed\*\*/

**public** **abstract** **void** insert();

**public** **abstract** **void** update();

**public** **abstract** **void** delete();

**public** **abstract** **void** refreshRelatedData();

/\*\*put more logic methods here if needed\*/

**public** T getTo() {

**return** to;

}

**public** **void** setTo(T to) {

**this**.to = to;

}

}

**IMPLEMENTATION**

**package** com.mallocinc.ytm.model.entities.api;

**import** com.mallocinc.ytm.database.generated.to.QuizDefinitionTo;

**public** **class** EntityQuizTaker **extends** EntityApi<QuizDefinitionTo>{

**public** ArrayList<EntityTakeQuestion> questionsToTake;

**public** EntityTakeQuestion fetchNextQuestion{

//some sample method

}

@Override

**public** **void** insert() {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** update() {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** delete() {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** refreshRelatedData() {

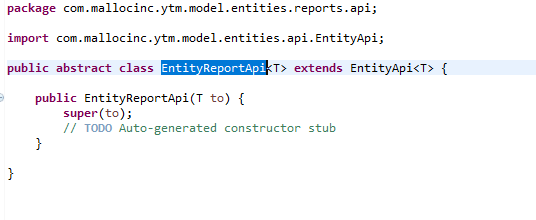
// **TODO** Auto-generated method stub

}

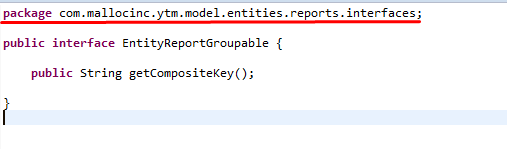
}

### Entity Report Classes

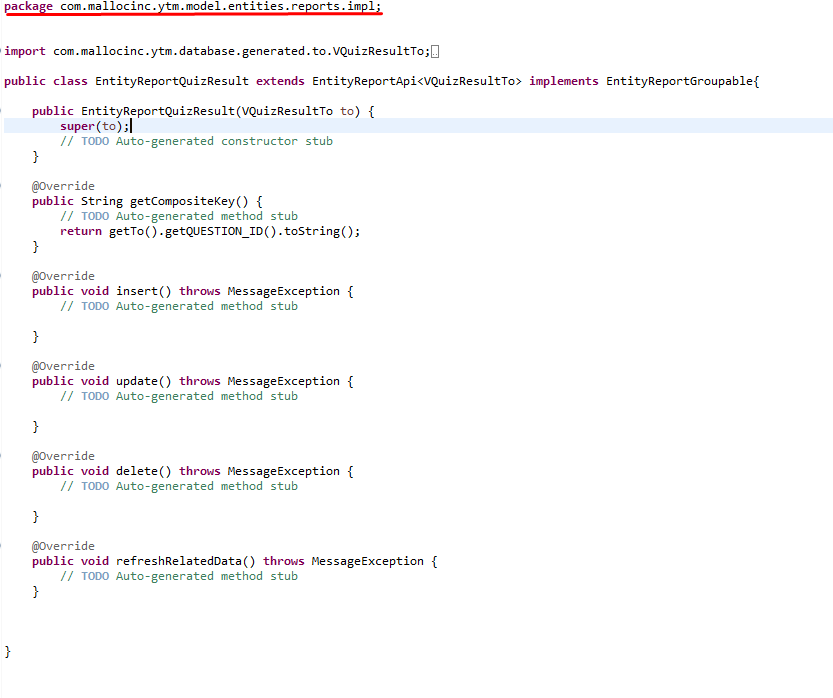
Classes for any type of report in our system have to extend EntityReportApi abstract class. EntityReportApi abstract class extends EntityApi abstract class (Section 5.6.2) which has a Transfer Object as parameter, so every implementation of EntityReportApi is linked with some row of the particular view or table in database.



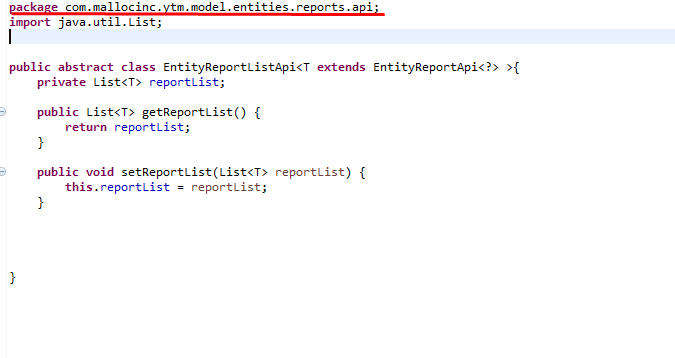
All mutual functionalities for many types of reports have to be placed in one interface with methods for every of that functionalities. All interfaces have to be placed in package from the picture below.



Implementation classes of EntityReportApi have to extend EntityReportApi and have to be placed in package from the picture below. Also, implementation classes have to implement interfaces for mutual functionalities (See the picture below). Implementation class can have some specific methods for that type of report.



For report functionalities that have to affect more than one row in specific table or view, EntityReportListApi abstract class has to be extended. EntityReportListApi class has the list of EntityReportApi objects (picture below). On this list group calculations can be done in Implementation class.



Example (Average number of points on quiz) of implementation of EntityReportListApi is shown in the picture below.



Of course, some mutual functionalities for different EntityReportList implementations should be put in one interface.

## Webview (Ready to use)

### (Important) Referencing and naming between xhtml and controller

At this point of development, the most crucial and fragile part of YTM code is the relation between xhtml files and Controller classes. Causes for the issue with xhtml usually comes up with fuzzy naming or by violation of [Law of Demeter](https://en.wikipedia.org/wiki/Law_of_Demeter), that are afterwards very hard to debug. For future development, it will be **mandatory** to have the correct approach to referencing and naming from xhtml.

1. **Referencing**

The general rule is the following: **simplier the xhtml – the better**. From a reference standpoint, in order to achieve simplicity it comes to one simple rule:

**Use Law of Demeter** and avoid this: value="#{userProfile.flightProgressController.flightBlockProgressController.blockProgress.block.codeName}" – you will have a nightmare only if you change any instance name within classes that refer to these objects. Law of demeter in our case says „**xhtml file should now only about its corresponding controller, and lazy tables within them (for now)**“. By following this rule, debugging time and potential bugs will be reduced **dramatically**.

Example: Lets say that we want to **access to email field within our EntityUser class**. Instead of going all the way through referencing within xhtml file – it should only refer to Controller class that has a reference to EntityUser. So instead of: controller.entity.email in xhtml, we should have **controller.getEntityUserEmail**(). This way, xhtml is dependant only to controller class. See pros and cons of Law of Demeter on the internet, but in the long run – it will dramatically reduce maintenance and bugs.

In addition to this, **use the benefit of private and public methods within controller classes.** If method is private, then it cannot be accessed from xhtml and therefore we dont have to check references for that method within xhtml files.

1. **Naming**

When it comes to naming, like everywhere else in the application – the best approach is to have a standard by which you call method names. This naming convetion proved to be benefitial so it should be followed initially, but it can be changed anytime for the better. These are naming rules:

* Reference to **listener** methods should be named like this „onEventDescriptionSelect“, where EventDescription is an event from the front end – for example „**onFinishQuizButtonSelect**“, „**onNextQuestionButtonSelect**“ etc.
* Reference to **render** methods should be named like this „renderEventDescription“ – for example „**renderFinishButton**“
* Reference to **disable** methods should be named like this „disableEventDescription“ – for example „**disableFinishButton**“

Having this in mind, since Controller methods are the biggest and, at this point, the most complex – there should be a standard on where each method should be in the Controller class. This is the **skeleton**:

* + Final global variables/Enums
  + Public global variables
  + Private global variables
  + Constructor(s)
  + Init method (with its private methods below it)
  + Listener methods (with their private methods below them)
  + Render methods (with their private methods below them)
  + Disable methods (with their private methods below them)
  + On the bottom of the class, put getters and setters of global variables (If necessary)

### Controllers and xhtml rules

1. Each section (eg. Panel or navigation tree) should be controlled by single controller
2. Each controller should be able to standalone and to be a part of other controller
3. Each controller will contain render flags for inner controller
4. Each xhtml section must be within a facet and reusable
5. Pages are in layout and there are layout sections eg. ui:define name=”westContent”
6. Dialogs must be in dialog define section
7. Facets can depend from each other (updating each other) and can consist each other
8. Facets on the same layer that are connected (eg. button facet and center content facet) must have starting code of the same controller. (DONT MIX parent and child code).
9. **Never put form inside the form**
10. Methods that calls provider classes should be surrounded by try catch and throw FacesMessage**.**
11. Update only what is really necessary. Always use process and update attributes.
12. Dont use Expression language where the logic can be put in backing bean, or even worse hardcode string.
13. Put commons such as ManageUser and methods declarations such as init(), resetFlags() etc inside controllers.commons.SuperController.java and extend them.

**BAD EXAMPLE:**rendered="#{!user.internalNetwork and (quizTaker.instanceToTake.quizInstance.QUIZ\_NETWORK\_ACCESS\_CODE eq ‘NAC’)}"

**GOOD EXAMPLE:**rendered="#{quizTaker.canTakeQuiz}”

1. There is no need for more than one Application scoped bean within application

### Outer controller code

@ManagedBean**(**name**=**"programs"**)**

@ViewScoped

public class ProgramsController extends SuperController**{**

ProgramController curProgram**;**

boolean viewAllPrograms**=true;** //initial visible screen

boolean viewCurProgram **=false;** //hidden screen

AllProgramsLazy allProgramsList**=new** AllProgramsLazy**();** //list of all programs

public void openCurProgram**(**BigDecimal id**){**

curProgram**=new** ProgramController**(**id**);**

resetFlags**();** //sets all render flags to false

viewCurProgram**=true;** //renders desired section

**}**

**}**

### Inner controller code

@ManagedBean**(**name**=**"prog"**)**

@ViewScoped

public class ProgramController **{**

EntityProgram program**;**

BigDecimal progId**;**

boolean shouldInit**=true;**

boolean viewProgramGroups**;**

boolean viewProgramInfo**;**

public ProgramController **(**BigDecimal progId**)** **{** //constructor is usually called from outer controller

consImpl**(**progId**);**//implementation of constructor

**}**

public void init**()** **{** //when stands alone on program.xhtml this is a prerender method which is being called

**if(**shouldInit**){** //we need this because method is called after every action (before every render) and we need it only ones

consImpl**(**progId**);**

shouldInit**=false;**

**}**

**}**

consImpl**(**BigDecimal progId**){**

**try{**

program **=** **new** ProgramProvider**().**getProgramById**(**progId**);**//fetching program

**....**

**}**

//code continues

### Outer controller webpage

<ui:..some tags..>

<ui:define name="metadata">

<f:metadata>

<!-- programs controller doesnt need init. Since it opens all programs list. -->

</f:metadata>

</ui:define>

<ui:define name="titleContent">

<!-- we usualy put page headline or breadcrumbs here -->

</ui:define>

<ui:define name="eastContent"><!-- we usualy put boolean buttons here -->

<h:form id="butForm">

<ui:include src="/pages/programParts/progEastFacet.xhtml">

<ui:param name="selectedProgram" value="#{programs.curProgram}" />

<ui:param name="viewCurProgram" value="#{programs.viewCurProgram}" />

<ui:param name="updateOnSelect"

value=":curProgForm :classesForm :butForm" /><!-- we can always pass different stuff for update depending on page. Facet knows only about #{updateOnSelect} -->

</ui:include>

</h:form>

<ui:define>

<ui:define name="centerContent">

<ui:include src="/pages/programParts/progCenterFacet.xhtml">

<ui:param name="selectedProgram" value="#{programs.curProgram}" /><!--object of current program -->

<ui:param name="viewCurProgram" value="#{programs.viewCurProgram}" /><!-- renderer of current program -->

</ui:include>

</ui:define>

<ui:define name="dialogs">

<!-- dialogs always go here -->

</ui:define>

</ui:composition>

### Inner controller webpage (standalone webpage for single program)

<ui:..some tags..>

<ui:define name="metadata">

<f:metadata>

<f:viewParam name="itemId" value="#{prog.progId}" /><!--here we open specific program -->

<f:event type="preRenderView" listener="#{prog.init()}"/>

</f:metadata>

</ui:define>

<ui:define name="titleContent">

<!-- we usualy put page headline or breadcrumbs here -->

</ui:define>

<ui:define name="eastContent"><!-- we usualy put boolean buttons here -->

<h:form id="butForm">

<ui:include src="/pages/programParts/progEastFacet.xhtml">

<ui:param name="selectedProgram" value="#{prog}" /><!-- we pass the whole bean -->

<ui:param name="viewCurProgram" value="true" />

<ui:param name="updateOnSelect"

value=":curProgForm :classesForm" /><!-- we can always pass different stuff for update depending on page. Facet knows only about #{updateOnSelect} -->

</ui:include>

</h:form>

<ui:define>

<ui:define name="centerContent">

<ui:include src="/pages/programParts/progCenterFacet.xhtml">

<ui:param name="selectedProgram" value="#{prog}" />

<ui:param name="viewCurProgram" value="true" />

</ui:include>

</ui:define>

<ui:define name="dialogs">

<!-- dialogs always go here -->

</ui:define>

</ui:composition>

### Facet example

Always put comments about the facet.

<ui:composition xmlns=*"http://www.w3.org/1999/xhtml"*

xmlns:ui=*"http://java.sun.com/jsf/facelets"*

xmlns:h=*"http://java.sun.com/jsf/html"*

xmlns:p=*"http://primefaces.org/ui"*

xmlns:f=*"http://java.sun.com/jsf/core"*>

<!— selectedProgram: ProgramController passed as a reference 🡪

<h:form id=*"curProgForm"*>

<p:panel id=*"curProgram"*

styleClass=*"centerPanel oneThirdPanel SHOWAUSERSParent leftSideHint ui-content-padding-none dataListUnits"*

rendered=*"#{selectedProgram.viewInfo}"* toggleable=*"true"*

closable=*"false"* widgetVar=*"curProgVar"*>

.... and so on...

### GDAO Statement

In GdaoStatement class you can see all methods for creating sql query like method to put AND, OR... or some methods to reset query or put parentheses.

We use the hash map to preserve the values we want to we want to access several times.

*String pera = "pera";*

*map.put("p", pera);*

- termConst (for const value) and termVal (variable value from hashmap) we use for creating some part of query.

E.g, if we want to write *" NAME LIKE '%P%' ",*

we will use " *termConst(GdaoTerm.****LIKE****, "NAME", "P")"* but if we want some value from hash map then we have to use *" termVal(GdaoTerm.****LIKE****, "NAME", "p")".*

There is some way to creating statement:

Example 1:



Now, our query look like "WHERE (EXERCISE\_ID::text LIKE '%vezba%' OR EXERCISE\_NAME::text LIKE '%vezba%')"

Example 2:

HashMap<String, Object> map = **new** HashMap<>();

String adis = "adis";

map.put("a", adis);

GdaoStatement stm = **new** GdaoPagingStatement( "COURSE\_NAME", 0, 10, GdaoPagingStatement.***DESC***);

stm.setOperatorsAndValuesMap(map);

stm.termConst(GdaoTerm.***EQUALS***, "COURSE\_ID", **new** BigDecimal(4002)).and().not().leftParen()

.termConst(GdaoTerm.***LIKE***, "USERNAME", "ADM").or().termVal(GdaoTerm.***LIKE***, "FIRST\_NAME", "a").or().termConst(GdaoTerm.***LIKE***, "LAST\_NAME", "ADM").rightParen();

stm.not(stm);

Query:

NOT (COURSE\_ID = 4002 AND NOT (USERNAME::text LIKE '%ADM%' OR FIRST\_NAME::text LIKE '%adis%' OR LAST\_NAME::text LIKE '%ADM%'))

### Lazy example

We can use lazy loading because we can “lazy-load” the children, which means that it does not actually load all the children when loading the parent. Instead, It loads then when requested to do so.

**public** **class** SelectLessonUsersLazy **extends** LazyDataModel<EntityUserLesson> **implements** SelectableDataModel<EntityUserLesson>, SelectLazyApi<EntityUserLesson> {

**private** List<EntityUserLesson> datasource = **new** ArrayList<>();

**private** **int** tableHeight = **new** Integer(UtilProperites

.*getSettingsProperty*("tableHeight"));

**private** ArrayList<EntityUserLesson> filteredItems;

**…**

**public** SelectLessonUsersLazy(BigDecimal lessonId, String userLessonRole) {

**this**.lessonId = lessonId;

**this**.userLessonRole=userLessonRole;

}

@Override

**public** List<EntityUserLesson> load(**int** first,

**int** pageSize,

String sortField,

SortOrder sortOrder,

Map<String, Object> filters

) { …

filters.put("LESSON\_ID", lessonId);

filters.put("USER\_LESSON\_ROLE", userLessonRole);

}

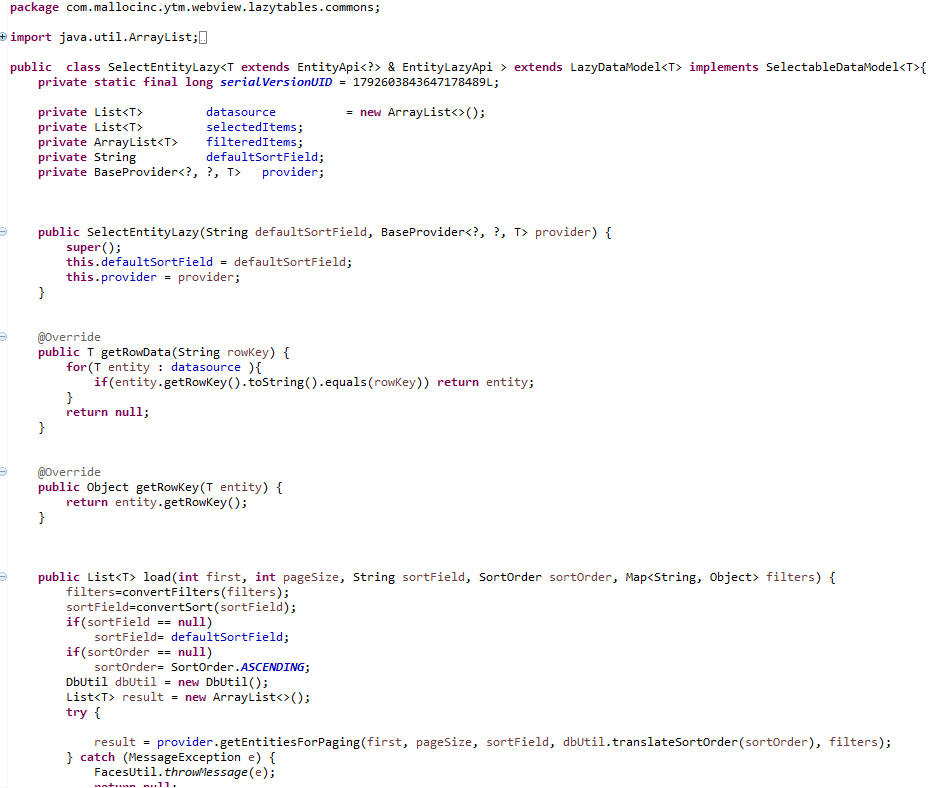
### Lazy Example – upgraded

In every project we should have SelectEntityLazy class which is the common class for all Lazy classes and they should extend it.

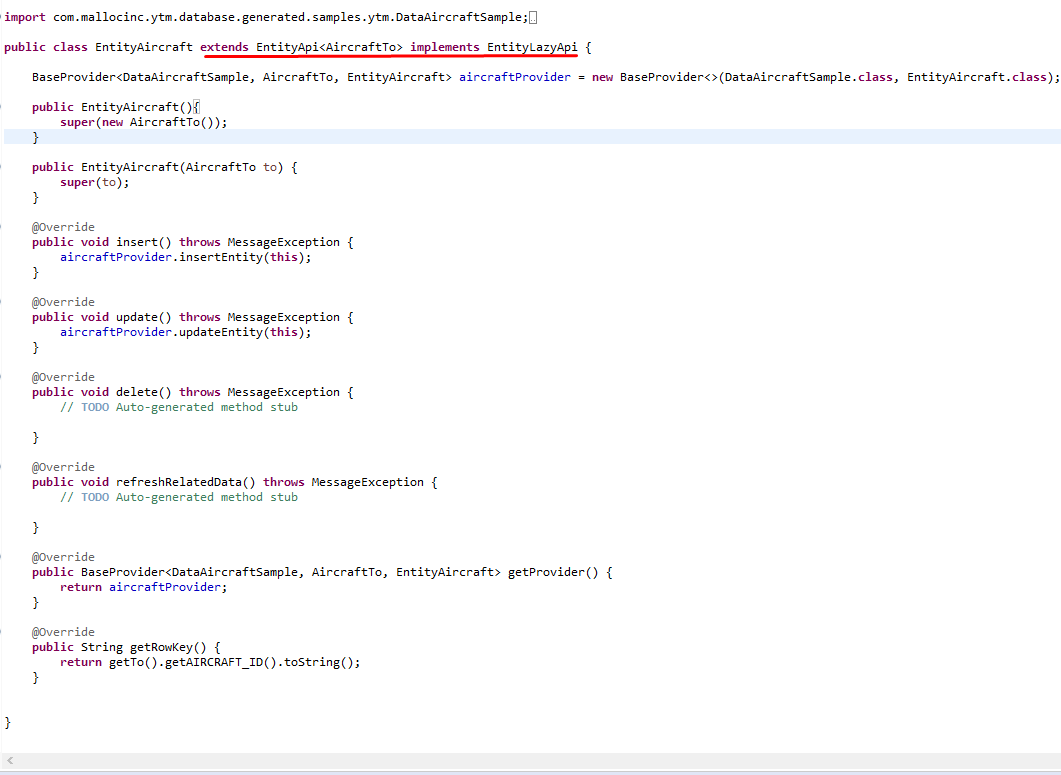
### Select entity lazy

SelectEntityLazy is a generic class that wraps LazyDataModel and it is used to load entities from database in primefaces’ DataTable.

The picture below shows the part of the SelectEntityLazy class definition. It is parametrized with Entity class that implements EntityLazyApi , which represents the entities that are displayed in the DataTable.



The picture below shows how the Entity class definition should in order to instances of that entity can be selected lazy.



The picture below shows how the SelectEntityLazy class should be instantiated. The constructor takes two parameters, the first represents the default sort column and the second one is the BaseProvider for entities.



If you need different SelectEntityLazy functionalities, SelectEntityLazy class should be extended and specific methods that represent that functionalities should be overridden.

### Select entity lazy (GDAO statement)

**private** GdaoStatement additionalStatement = **new** GdaoStatement();

**public** SelectEntityLazy(BaseProvider<?, ?, T> provider) {

**super**();

**this**.defaultSortField = **null**;

**this**.provider = provider;

additionalStatement.setOperatorsAndValuesMap(additionalFilters);}

...

**public** List<T> load(**int** first, **int** pageSize, String sortField, SortOrder sortOrder, Map<String, Object> filters) {

**if**(!additionalFilters.isEmpty()) addFilters(filters);

filters=convertFilters(filters);

sortField=convertSort(sortField);

**if**(sortField == **null**)

sortField= defaultSortField;

**if**(sortOrder == **null**)

sortOrder= SortOrder.***ASCENDING***;

DbUtil dbUtil = **new** DbUtil();

List<T> result = **new** ArrayList<>();

**try** {

GdaoStatement gdaoStatement = prepareStatement(filters, first, pageSize, sortField, dbUtil.translateSortOrder(sortOrder));

**if**(gdaoStatement.getStatement().isEmpty()) gdaoStatement.setStatement(additionalStatement.getStatement());

**else** gdaoStatement.and(additionalStatement);

***log***.info(gdaoStatement.getQuery());

result = provider.selectUsingGdaoStatement(gdaoStatement);

***log***.info(gdaoStatement.getQuery());

GdaoStatement countStatement = **new** GdaoStatement();

countStatement.setStatement(gdaoStatement.getStatement());

***log***.info(gdaoStatement.getQuery());

**int** rowCnt = (**int**) provider.selectCountGdaoStatement(countStatement);

**this**.setRowCount(rowCnt);

} **catch** (MessageException e) {

FacesUtil.*throwMessage*(e);

**return** **null**;

}

datasource.addAll(result);

**return** result;

}

**private** GdaoStatement prepareStatement(Map<String, Object> filter, **int** firstPage, **int** pageSize, String sortField, String sortOrder){

GdaoStatement statement = **new** GdaoPagingStatement(sortField, firstPage, pageSize, sortOrder);

**boolean** first = **true**;

**for**(String key: filter.keySet()){

Object value = filter.get(key);

**if** (first) first = **false**;

**else** statement.and();

statement.termConst((value **instanceof** String)?GdaoTerm.***LIKE***:GdaoTerm.***EQUALS***, key, value);

}

**return** statement;

}

## Reusable search components

Each search component consists of a Controller (which implements the Callback interface), a Facet (the controller is forwarded to it) and a Face converter . The surrounding bean implements the Callback method.

First of all, we have to create interface:

**public** **interface** CallbackSelectExercise {

**public** **void** onExerciseSelect(SelectEvent event);

}

Then, next step is creating Controller:

**public** **class** CourseExerciseSearchController {

CallbackSelectExercise parentController;

**private** **int** minQueryLength = 0;

**private** **int** queryDelay = 1000;

**private** EntityExercise curSearch = **null**;

BigDecimal courseId;

BaseProvider<DataExerciseSample, ExerciseTo , EntityExercise> provider =

**new** BaseProvider<>(DataExerciseSample.**class**, EntityExercise.**class**);

**public** CourseExerciseSearchController(CallbackSelectExercise parentController, BigDecimal courseId) {

**super**();

**this**.parentController = parentController;

**this**.courseId = courseId;

}

**public** List<EntityExercise> completeExercises(String str) {

String[] splited = str.split("\\s+");

GdaoStatement stmt=**new** GdaoPagingStatement(Columns.***EXERCISE\_NAME***.toString(), 0, 10,GdaoPagingStatement.***ASC***);

stmt.leftParen();

**boolean** first=**true**;

**for**(String query: splited) {

**if**(!first){

stmt.or();

}

stmt.termConst(GdaoTerm.***LIKE***, Columns.***EXERCISE\_ID***.toString(), query)

.or()

.termConst(GdaoTerm.***LIKE***, Columns.***EXERCISE\_NAME***.toString(), query);

first=**false**;

}

stmt.rightParen();

List<EntityExercise> results = **null**;

**try** {

results = provider.selectUsingGdaoStatement(stmt);

} **catch** (MessageException e) {

FacesUtil.*throwMessage*(e);

}

**return** results;

}

getters and setters...

Creating converter and facet

@FacesConverter("exerciseSearchConverter")

**public** **class** CourseExerciseSearchConvert **implements** Converter {

BaseProvider<DataExerciseSample, ExerciseTo , EntityExercise> provider =

**new** BaseProvider<>(DataExerciseSample.**class**, EntityExercise.**class**);

**public** Object getAsObject(FacesContext fc, UIComponent uic, String value) {

**if** (value != **null** && value.trim().length() > 0) {

**try** {

**return** findExercise(value);

} **catch** (NumberFormatException e) {

**throw** **new** ConverterException(**new** FacesMessage(FacesMessage.***SEVERITY\_ERROR***, "Conversion Error", "Not a valid test."));

}

} **else** {

**return** **null**;

}

}

**public** String getAsString(FacesContext fc, UIComponent uic, Object object) {

**if** (object != **null**) {

**return** String.*valueOf*(((EntityExercise) object).getTo().getEXERCISE\_ID());

} **else** {

**return** **null**;

}

}

**private** EntityExercise findExercise(String value) {

BigDecimal id = **new** BigDecimal(value);

**try** {

HashMap<Object, Object> filters = **new** HashMap<Object, Object>();

filters.put("EXERCISE\_ID", id);

**return** provider.getSingleRecordWithParameters(filters);

} **catch** (MessageException e) {

FacesUtil.*throwMessage*(e);

}

**return** **null**;

} }

****

After all, we have to instance our controller and implements method for onSelect

**public** **class** PlanOrderRecordController **extends** SuperController **implements** CallbackSelectCourseMember,CallbackSelectAirCraft,**CallbackSelectExercise**

{

EntityVPlanOrderRecord planOrderRecord;

CourseMemberSearchController memberSearchController;

AirCraftSearchController airCraftSearchController;

**CourseExerciseSearchController courseExerciseSearchController;**

**public** PlanOrderRecordController(EntityVPlanOrderRecord planOrderRecord) {

memberSearchController=**new** CourseMemberSearchController(**this**, planOrderRecord.getTo().getCOURSE\_ID());

airCraftSearchController=**new** AirCraftSearchController(**this**);

**courseExerciseSearchController=new CourseExerciseSearchController(this, planOrderRecord.getTo().getCOURSE\_ID());**

**this**.planOrderRecord=planOrderRecord;

}

...

**public** CourseExerciseSearchController getCourseExerciseSearchController() {

**return** courseExerciseSearchController;

}

**public** **void** setCourseExerciseSearchController(CourseExerciseSearchController courseExerciseSearchController) {

**this**.courseExerciseSearchController = courseExerciseSearchController;

}

@Override

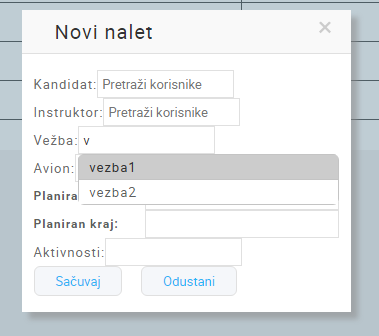
**public** **void** onExerciseSelect(SelectEvent event) {

EntityExercise exercise=(EntityExercise) event.getObject();

planOrderRecord.getTo().setEXERCISE\_ID(exercise.getTo().getEXERCISE\_ID());

planOrderRecord.getTo().setEXERCISE\_NAME(exercise.getTo().getEXERCISE\_NAME());

}



# System (application level) parameters management

The class responsible for system parameters management is **SystemParametersManager** from com.mallocinc.ytm.model.providers.system package. This class is a global singleton, which means that only instance of this class exists in the runtime. **SystemParametersManager** on instantiationloads all system parameters stored in database and stores them in the memory. So, when we need a value of particular system parameter, we don’t need to read it from database every time, we **should** read it from the instance of **SystemParametersManager.**

## How to use **SystemParametersManager**

### Getting the instance

SystemParametersManager systemParameterManager = SystemParametersManager.*getInstance*();

### Getting value of system paramter

communityFlag = systemParameterManager.getBooleanParameterValue(UtilSystemParameter.***CODE\_COMMUNITY***);

Types of system parameters can be String, boolean, Blob and BigDecimal, and there is a method getting value for each type. You just need to pass the code of the system parameter.

### Changing value of system paramter

systemParameterManager.changeParameterValue(UtilSystemParameter.***MODULE\_CODE\_AD***, **true**);

You need to pass code of system paratemer and new value. The value of system parameter will be changed both in memory and the database.

### Good practices

For every system parameter, two methods should be introduced in **SystemParametersManager.**

1. One for getting system parameter value. For example:

**public** **static** **boolean** shouldDisplayLanguage() {

**return** *getInstance*().

getBooleanParameterValue(UtilSystemParameter.***CODE\_DISPLAY\_LANGUAGE***);

}

So, we can check the value of “display language” system parameter by:

SystemParametersManager.*shouldDisplayLanguage*();

1. One for changing system parameter value. For example:

**public** **static** **void** changeDisplayLanguageValue(**boolean** value) **throws** MessageException {

*getInstance*().changeParameterValue(UtilSystemParameter.***CODE\_DISPLAY\_LANGUAGE***, value);

}

These two methods can simplify the code when you work with specific system parameter.

### Important rules

1. **Don’t read system parameters directly from database. Use SystemParametersManager for that.**
2. **Update the values of system parameters through SystemParametersManager.**

# Session level parameters

When the user logs in the application, HTTP session associated with that user is created, and all significant data for the management of the user session is stored in the session map. The main session parameter is ID of logged in user. Almost all other session parameters can be loaded from database, but because those parameters are frequently used, it’s more efficient to load them in the memory (session parameters) only ones on the session initialization and not to query the database every time to get them. Class responsible for the management of the session data currently logged in user is **ApplicationUser** located in com.mallocinc.ytm.webview.session package. This class has granular structure which consist of a few classes each responsible for management of some independent parts of user session data:

1. ApplicationUserData – responsible for elementary session data like user account information, user preferences, IP address, device, session ID
2. ApplicationUserHints – responsible for management of user hints on session level
3. ApplicationUserRoles – responsible for management of user roles on session level

## Getting ApplicationUser object for currently logged in user

Wherever you need to use session parameters you can get the ApplicationUser object like this:

**private** ApplicationUser applicationUser = HttpSessionUtil.*getCurrentUser*();

See the Javadoc for more information about the available data in the ApplicationUser object.

# Useful YouTestMe Tutorials

Up to date list of tutorials can be found on YouTestMe Wiki:

<https://wiki2.youtestme.com/index.php/Useful_Links>

Dev account:

Username: Ytmdev

Password: 1Dev#@!

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Category** | **Create date/ Creator** | **Description** |
| [**Model** and **logic** layer](https://www.youtube.com/watch?v=bgwaYONDKTY&index=1&list=PLNr69myqMnd8b_36ZX-NP34q2_FmP5Sun) | Backend | 29.11.2016.  Danilo Sretenovic | Tutorial about ytm.db and ytm.model  (Sligthly outdated, you can watch it informately) |
| [**Webview** layer and example](https://www.youtube.com/watch?v=snAY1K7fLM8&index=2&list=PLNr69myqMnd8b_36ZX-NP34q2_FmP5Sun) | Backend  Frontend | 29.11.2016.  Danilo Sretenovic | Tutorial about ytm.webview  (Sligthly outdated, you can watch it informately) |
| [2018-Gdao standardized web development](https://www.youtube.com/playlist?list=PLNr69myqMnd-GynnGY2TiyEZvVxywGfjd) | Backend  Frontend | 10.09.2018.  Danilo Sretenovic  Adis Dijab | Architecture description with examples of coding standard |
| [1. Introduction - Importing project into Eclipse and setting environment](https://www.youtube.com/watch?v=RwicBEB0Df8&feature=youtu.be) | Backend  Frontend | 31.07.2019.  Adis Dijab | Setting eclipse enironment and importing GetCertified project  (Most recent) |
| [2. Power designer - modeling database](https://www.youtube.com/watch?v=rGdcCAQTcq4&feature=youtu.be) | Backend | 31.07.2019.  Adis Dijab | PowerDesigner tutorial, Modeling database, generating sql code, writing database patch  (Most recent) |
| [3. GDAO - generating db layer code](https://www.youtube.com/watch?v=j45CR4Zm9lQ&feature=youtu.be) | Backend | 31.07.2019.  Adis Dijab | Generating database layer with Database Analyzer  (Most recent) |
| [4. Entity Provider Refactoring Concepts](https://www.youtube.com/watch?v=CMIkbN-8nEE&feature=youtu.be) | Backend | 31.07.2019.  Adis Dijab | Coding standard for entites and providers, refactoring code  (Most recent) |

# Logging in Java

Logging is essential part of all Java code and it has purpose to give information about:

* what is going on so we can ensure that system is running as expected and problem free
* how long it takes to execute partucular function (essential for coding for good performances)
* errors on the system with relevant information required to quckly solve the problem

Below are guidelines about log4j Java logging.

| **Logging Level** | **Description** | **Example** |
| --- | --- | --- |
| ERROR | This is event that support team has to be immediately notified about.  It is always turned on in all environments.  Email should be sent to support team immediately. | * Could not read or write into a database * No disk space left |
| WARNING | Events that requre attention but do not immediately affect availability of the system.  It is always turned on in all environments. | * User XYZ is locked due to 5 unsuccesfull log in attempts * Supperuser logged in * One of the Tomcat servers in the cluster is shut down |
| INFO | Events that administrator of the system is interested in.  It is always turned on in all environments. | * Application started on 2021-06-21 10:18:32 * System parameters are sucessfully loaded in memory * Application received shut down signal * Loaded 1,439 questions from file “questions.xlsx” * User X logged in to the system |
| DEBUG | Events that are used for debugging and performance testing.  It is turned on in development and QA. environment. It is tuerned on in production only on demand. | * Function XYZ started * Function XYZ finished * Starting SQL query XYZ * Finished SQL query XYZ * Inserted 4 records in database table XYZ (this type of logging should be automatically covered by GDAO framnework) |