Take it easy, let your computer do the work.

easyMda User's guide

easyMda

Fica Press

Robin Bakkerus

_

The software developer's paradox:
While many programmers are working all day
to write software that automates repetitive tasks,
They hardly bother to automate their own work.

rb.

Disclaimer
This open-source project is much inspired by an (similar) excellent project Taylor Mda by John Glibert. You may recognize a number of class- and method-names, but because the model (Java) is completely different from the model used in Taylor Mda (UML), almost all code is new.

easyMda user's guide, version 1.8 11. May. 2013
robin.bakkerus@gmail.com

The main being used is OpenDyslexis (see http://dyslexicfonts.com/)
Cover design : Niels Bakkerus (www.bakkerus.nl)

Table of Contents

1 Installation

The installation of easyMda is simple, just follow the next steps:

1. At this point, I assume that you have unzipped the easyMda-1.8.zip in some folder, that will be referenced with <easyMda-dir> from now on.
2. Copy all the files under the "**eclipse/dropins**" folder to the "**dropins**" folder of your Eclipse Juno (classic or Jee version) installation.
   Windows: `copy <easyMda-dir>/eclipse/dropins <Eclipse-Juno>/dropins`
   Linux: `cp -rf <easyMda-dir>/eclipse/dropins <Eclipse-Juno>/dropins`

Now the easyMda plug-in should be ready to use.

Notes:

- **You need Java 7!**
  That is because the plugin, cartridges and frameworks are all compiled with Jdk7. But I used Java-6 source code compatibility, so if you really want to use it under Java-6 you should recompile everything under Java-6 (see developer's guide)
  So far I only tested the plug-in under Eclipse Juno, but it probably works under older versions as well.
- For several reasons, this plug-in is expanded under the "dropins" folder, and not deployed as a single jar file. For the same reasons, it is not possible in the current version, to use a link file in the links directory.
- I tested this version under Ubuntu 10.01 with
  Eclipse : Juno Release Build id: 20120614-1722
  Jdk1.7.0_07
- Windows 7 with:
  Eclipse : Juno 4.2.1 Release Build id: M20120905-2300
  Jdk1.7.0_07

2 Quick-start, the wizard demo project

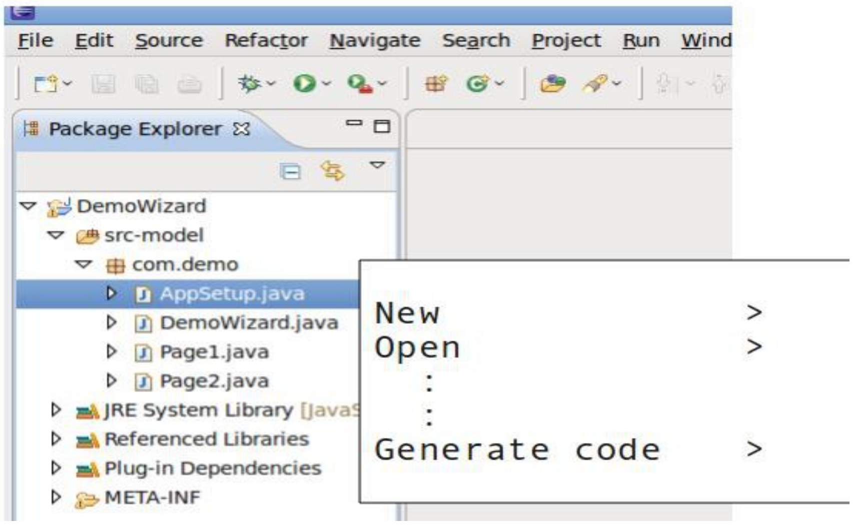
In order to see how the easyMda plug-in works, the installation contains a small demo project as a quick-start. In this chapter we will rush through the demo. Simply follow the commands, and within 5 minutes you should be able to have a working wizard.

Later on we will do the demo over again, then I describe what the hell we are doing exactly.

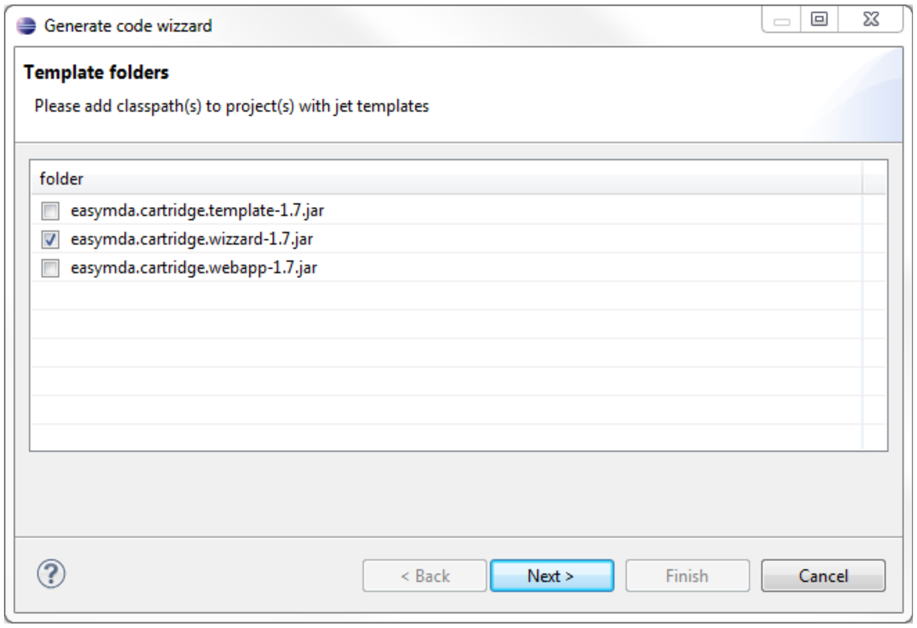
1. Import the following two projects from <easyMda-dir>/samples into Eclipse :
   DemoWizzardModel and DemoWizzardTarget

> Note for Linux, MacOs and Win64 users!!
> I tried to make the setup of this quick-start as simple as possible, so I included all necessary jar files. Unfortunately (although it is Java) the setup on Linux and MacOs is slightly different from a Windows setup. So Linux/MacOs users should change the library path from swt-windows-4.2.1.jar to the swt-xxx.jar that corresponds with your OS and Jvm!
> If you can't find the right one, you may find a suitable swt version at:
> http://archive.eclipse.org/eclipse/downloads/drops/R-3.6.1-201009090800/index.php#SWT

2. DemoWizzardModel is a small project is an example how a model in easyMda may look like. It is just a plain old simple Java project. In this case it only contains 4 java files, that describe the model for a jface wizard that we want to generate.
3. Select the in the package explorer, the package from the demo project, then right click and and select the 'generate-code' tag at the bottom

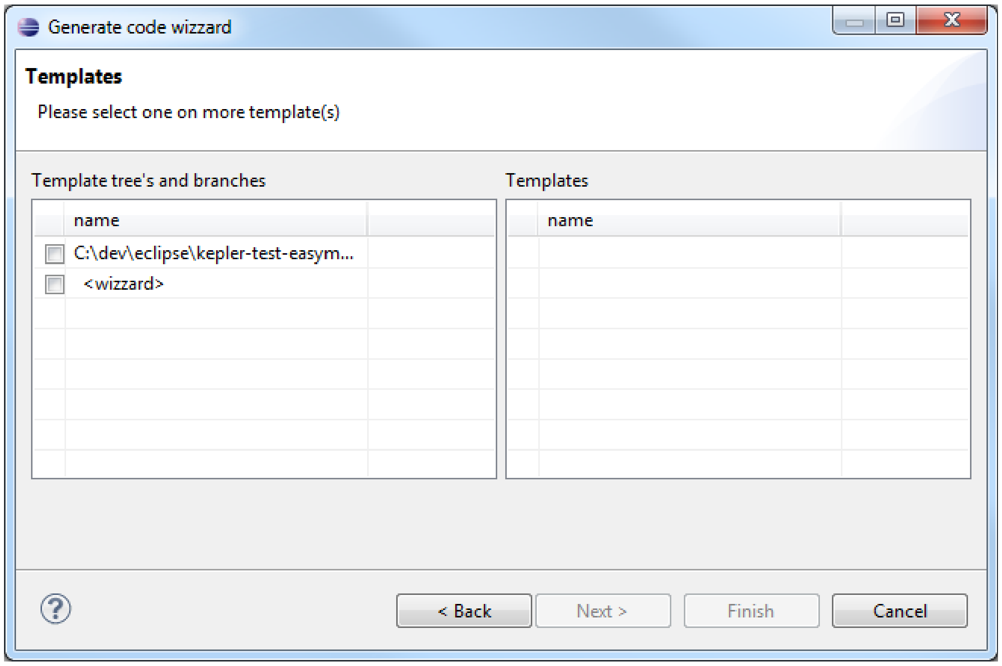


The following easyMda wizard appears:


Generate code wizzard
Template folders
Please add classpath(s) to project(s) with jet templates
folder
easymda.cartridge.template-1.7.jar
easymda.cartridge.wizzard-1.7.jar
easymda.cartridge.webapp-1.7.jar
< Back
Next >
Finish
Cancel

This page shows what cartridges are available. (See the developers guide, how to add your own templates). Note that the [Next] will only be enabled when exactly one cartridge Is selected.

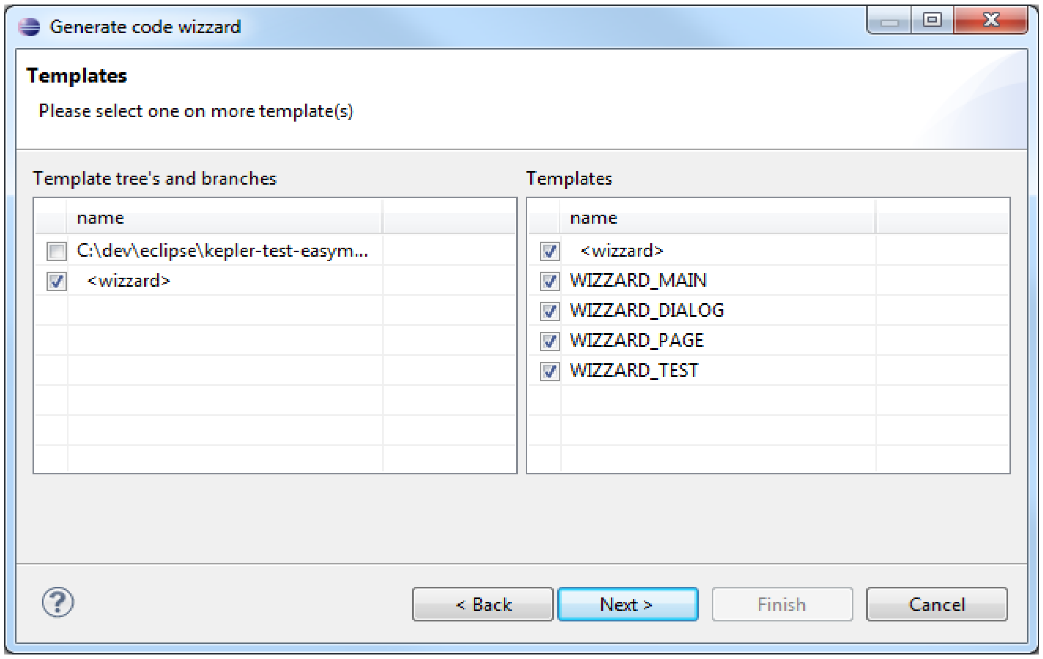
Hit [Next>] this will the second page:


Generate code wizzard
Templates
Please select one on more template(s)
Template tree's and branches
Templates
name
C:\dev\eclipse\kepler-test-easym...
<wizzard>
name
< Back
Next >
Finish
Cancel

On the left-hand side you see once more the available templates, and below that, you see all the available template-branches. The are indented, and surrounded with brackets like:

<wizard>

Select the checkbox of this template-branch. Now the right-hand side listbox will be populated with all templates that are applicable with the current selected packages and class files:


Generate code wizzard
Templates
Please select one on more template(s)
Template tree's and branches
Templates
< Back
Next >
Finish
Cancel

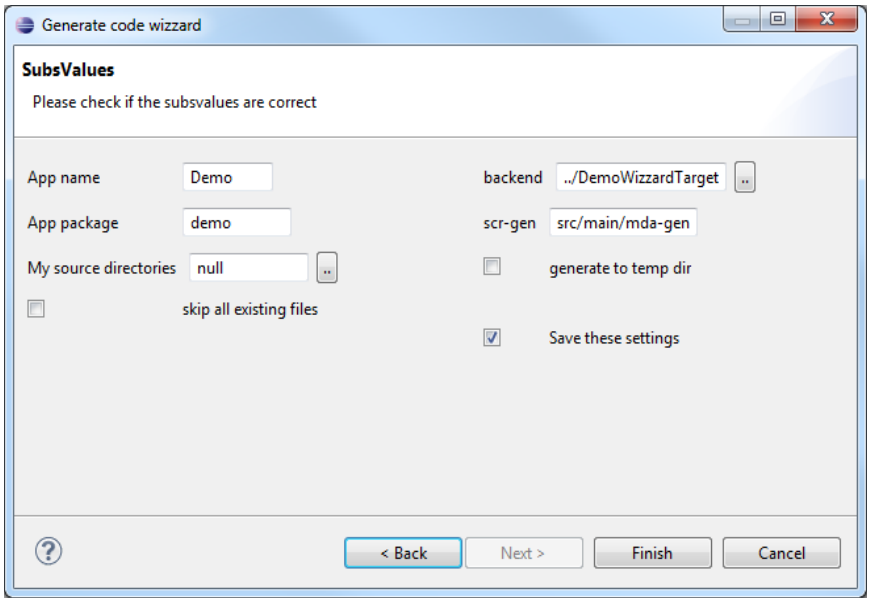
	name
☐	C:\dev\eclipse\kepler-test-easym...
☑	<wizzard>

	name
☑	<wizzard>
☑	WIZZARD_MAIN
☑	WIZZARD_DIALOG
☑	WIZZARD_PAGE
☑	WIZZARD_TEST

You can now either select individual templates or you select all at once, by the selecting the

[ ] <wizard> checkbox. (at the top)

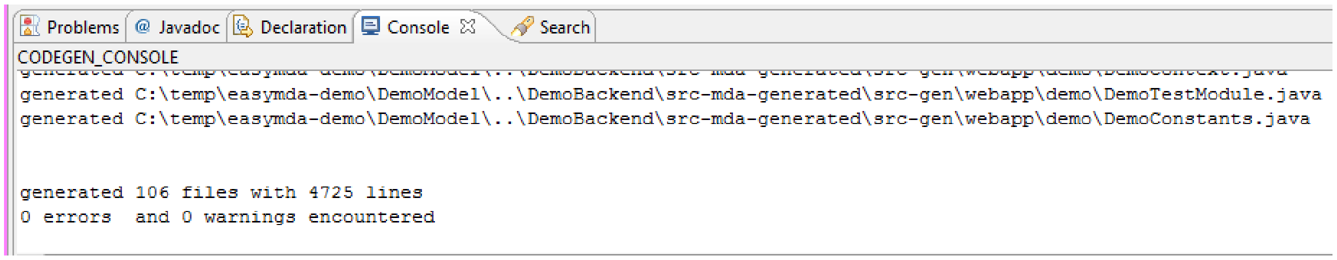
Hit [Next] this will bring us to the third and last page:



App name	Demo
App package	demo
My source directories	null
☐ skip all existing files	
backend	../DemoWizzardTarget
scr-gen	src/main/mda-gen
☐ generate to temp dir	
☑ Save these settings	

Again this page may vary depending on choices you made on the pages before, and also if you run the easyMda plug-in in this model-project for the first time. If you run it for the first time, some values will be empty and should be filled by you. The next time this value will be used.
For this demo I already prepared the correct default values, so you can simply hit the

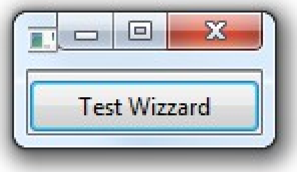
Click [Finish] button.
Now the generator will generate all the code. After a second or so, you should see a summary in the console like this:



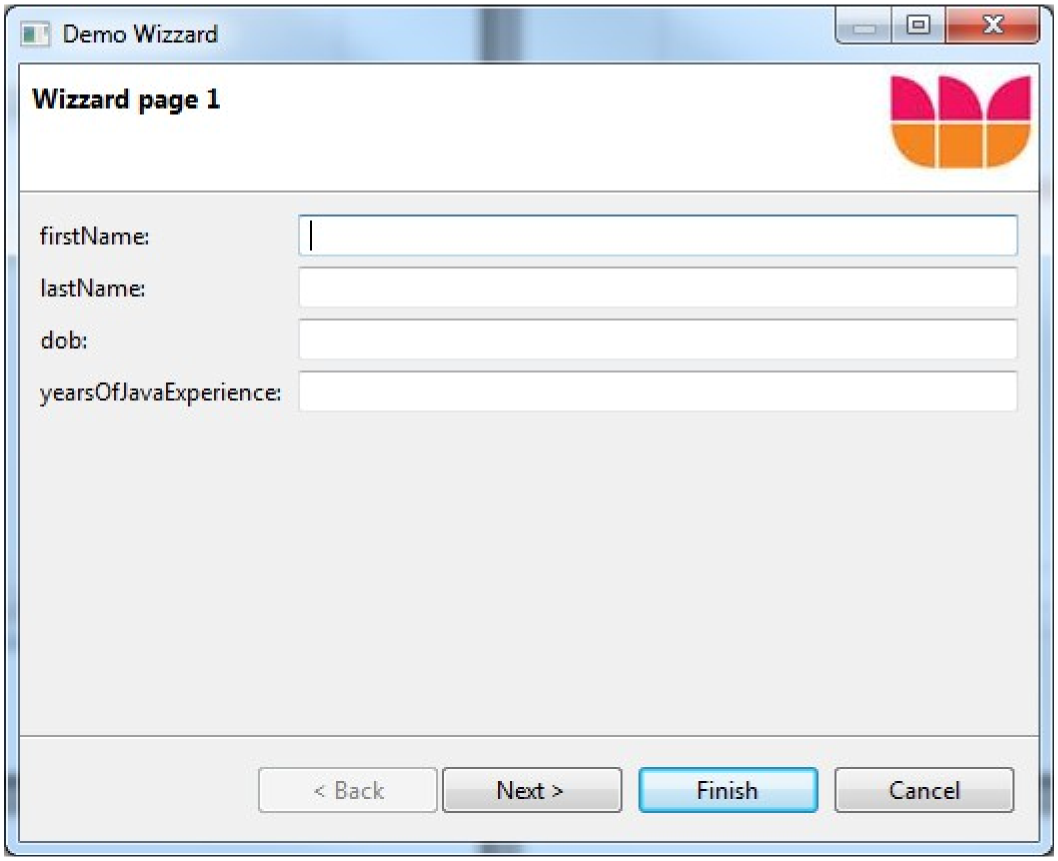
```
generated C:\temp\easymda-demo\DemoModel\..\DemoBackend\src-mda-generated\src-gen\webapp\demo\DemoTestModule.java
generated C:\temp\easymda-demo\DemoModel\..\DemoBackend\src-mda-generated\src-gen\webapp\demo\DemoConstants.java


generated 106 files with 4725 lines
0 errors  and 0 warnings encountered
```

Now refresh the demo project (Project → Refresh or F5)
Now java file should appear under the src-mda-generated folder
The new java files should now compile, without errors (or warnings)

Select the generated class: TestWizzard and
Run as → Junit test.
This start the newly created wizard.



And hitting the [Test Wizzard] button, will pop-up the actual swt based wizard:

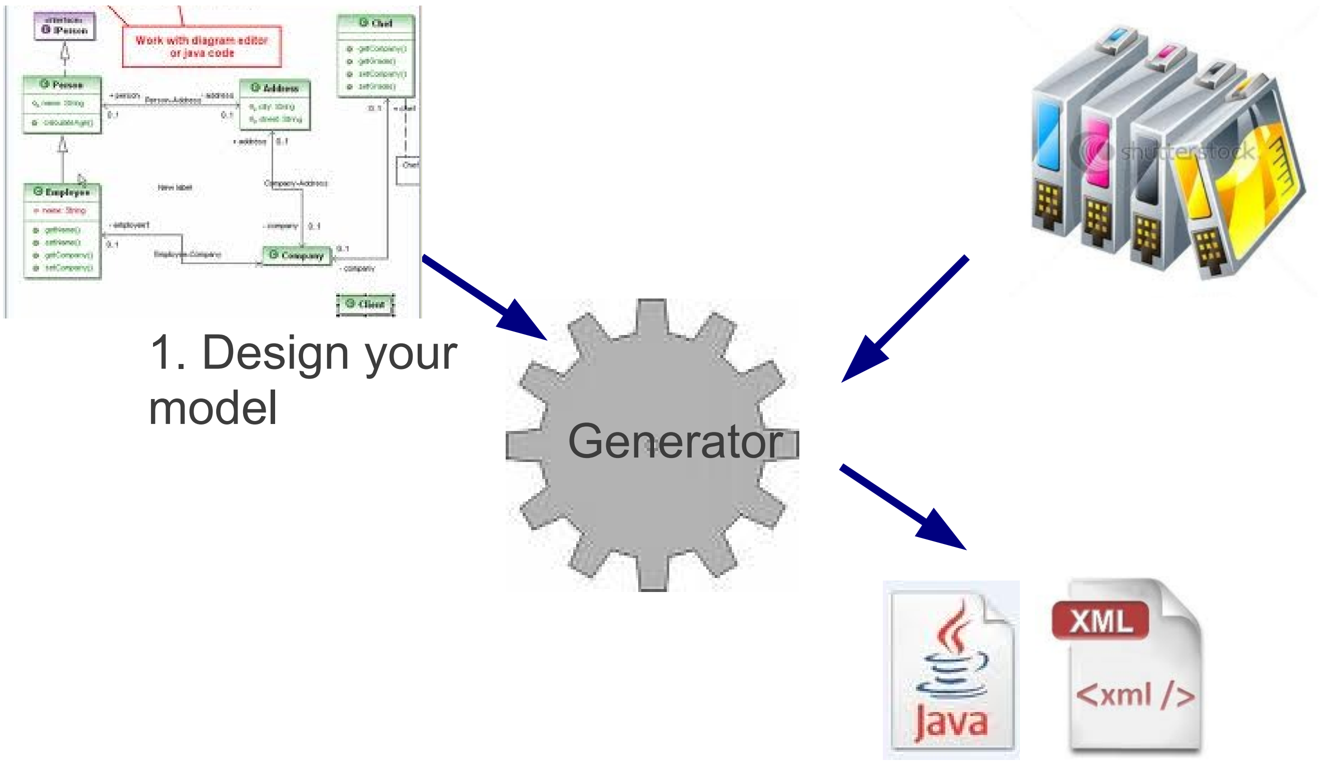


Congratulations, you just generated your first set of files using easyMda. I hope you taste for more. In the next chapters I will go much more in detail through the demo's, but that will not take much time either.

3 What is easyMda

easMda is a general purpose MDA/code-generator tool. It is similar to products like: AndroMDA, OptimalJ, Taylor MDA, Acceleo, Mod4J, Spring Roo and many others.

All these MDA tools, including easyMda, share the following main architecture:



Also the process flow for all these tools is basically the same:

1. You start by designing a software model.
2. Select a module, cartridge or template to tell the generator what you want to generate.
3. Generate !

And in all cases source code is generated.

The differences between all these tools are, how all these parts are implemented.

The model

All these tools mentioned above can be divided in two categories when it comes to the model it uses as the basic to generate-code.

You have the :

- UML based league (AndoMDA, OptimalJ, Taylor, Acceleo and others)
- And DSL based league (Spring Roo and Mod4j and also Microsoft's Oslo)

The generator engine

Each tool comes with its own generator engine. Some are standard engines. So makes AndroMDA use of Velocity, Taylor uses Eclipses (EMF) Jet engine. OptimalJ has it's own engine, and DSL is more or less a generator engine in itself.
easyMda uses Jet , that is part of Eclipse modeling framework (EMF), as its generator engine.

Generated code

The generated code is uh ... well 'just' generated code as if you would have written it yourself. How 'ready-to-use' the generate code really is, depends on how good the cartridge or module is (or wants to be).

What makes easyMda different, from the others. easyMda differs on one significant point. easyMda is not based on an UML or DSL model, but instead, it uses Java itself as the model! Using Java as the model has the following advantages:

- It is easy to create the model: it is just Java.
- To explore the model, you get a lot of power for free: java reflection.
- It is easy to create your own (type-safe) java-based 'DSL's, with inheritance and annotations.
- Via Ide's like Eclipse you get the features like intellisense and auto-completion for free.

The assets of easyMda

easyMda has three main component's :

- The Eclipse plug-in itself + the User's guide.
  With this part, an end-user can create a model-project, and generate source code out of it.
- The overall framework to create and deploy your own templates cartridges, plus the Developer's guide.
- Ready-to-use cartridges, that can be used either out-of-the-box. Or as a starting point to build your cartridge.

The focus for the short term, is to finalize the plug-in and framework, including the corresponding user and developer's guide. After that the focus will be on developing and publishing ready-to-use template cartridges.

In the current first version two cartridges are supplied:

- com.flca.cartridge.wizzard

This is merely a demonstration project to show how the plug-in works, how jet template files look like etc.

• com.flca.cartridge.webapp
This is to become a ready-to-use cartridge to generate a web based application with a Jee backend and a choice of different frontends: Dojo, Flex, and ZK

• com.flca.cartridge.template
This is a simple cartridge that I use when developing (new) cartridges !

Future plans

The following topics are on my wish list for future releases:

- Extented webapp wizard. with template branches for a backend based on GAE (Google App Engine)
- Support for Perst database
- Support for at least one No-SQL file-based databases (Neo4J ?)
- Security
- Scala backend

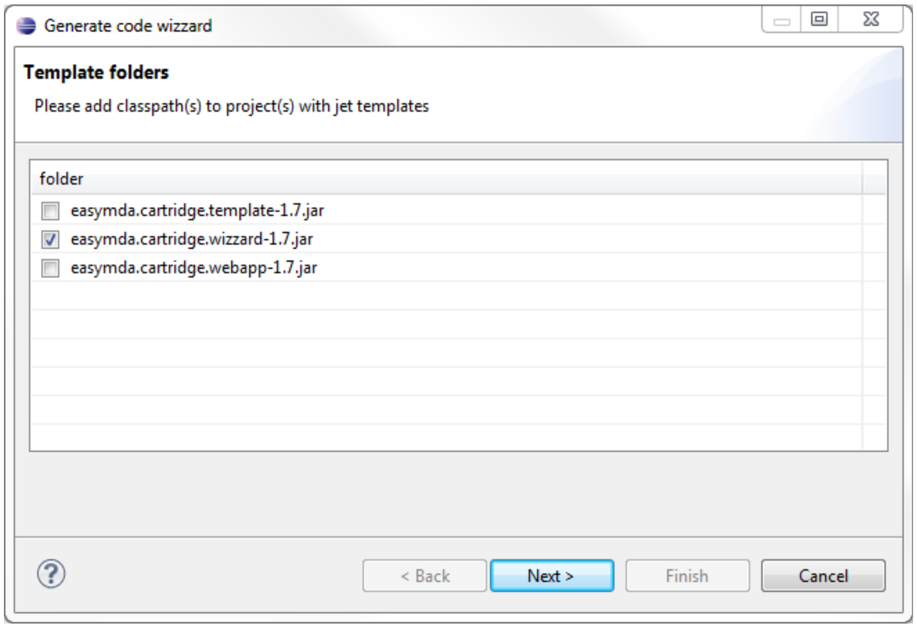
But I am particularly interested in any request coming from you!

4 The wizard demo project, revisited.

In this chapter, we will more or less follow the same steps from the previous quick-start, but this time I explain what you do exactly, and we will also make some sidesteps.

Our starting point in this chapter, is the same as the previous quick start, that is the wizard demo project that we imported earlier (in the next chapter, we will create our own model project).

1. Select the in the **package** explorer, the package from the demo project like this:
2. Right click and and select the **generate-code** button at the bottom.
3. The following easyMda wizard appears:


Generate code wizzard

Template folders

Please add classpath(s) to project(s) with jet templates

folder
- [ ] easymda.cartridge.template-1.7.jar
- [✓] easymda.cartridge.wizzard-1.7.jar
- [ ] easymda.cartridge.webapp-1.7.jar

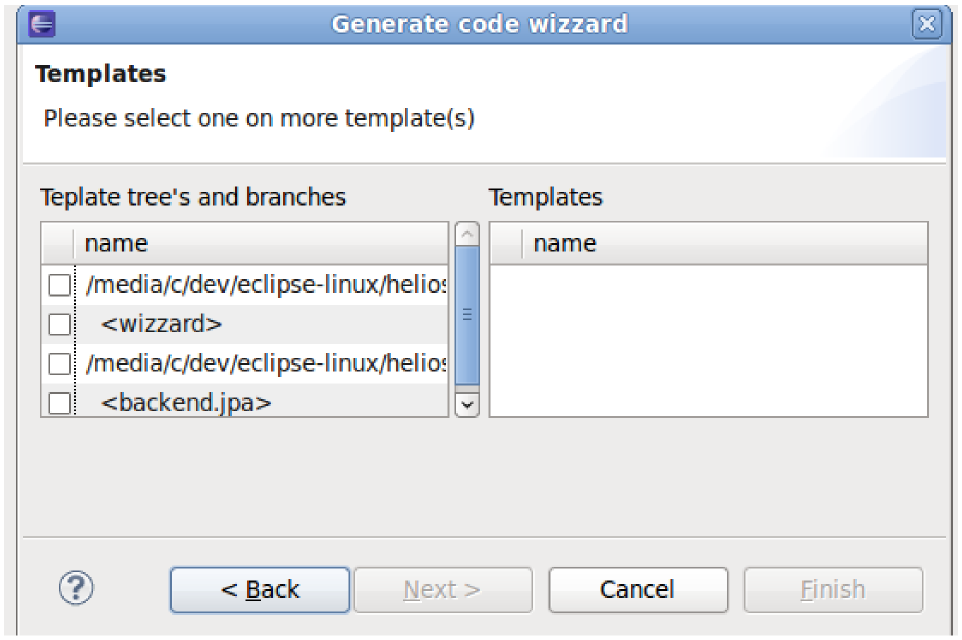
You now may see zero or more names of classpath's that point to a Eclipse java project that contain so called Jet templates. This classpath can be either a java jar file or the location of the bin directory. These are classpath's are set in:

```
<eclipse-dir>/plugins/easyMda-1.8/cartridges
```

That can be either a jar file, containing a templates project, or a link file, that contains a path to a bin directory of templates project.
For muck more info on this topic, see the developer's guide.

Only one cartridge may be selected in order to enable the [Next] button.

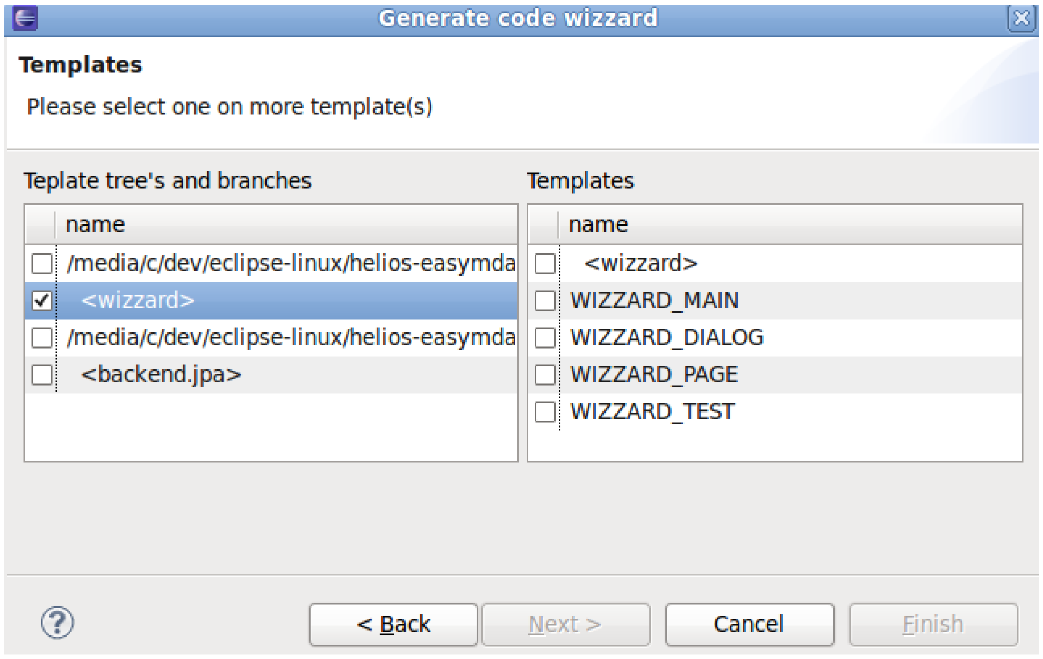
when you click [Next] the following wizard page appears:


Generate code wizzard
Templates
Please select one on more template(s)
Teplate tree's and branches
Templates
name
/media/c/dev/eclipse-linux/helios
<wizzard>
/media/c/dev/eclipse-linux/helios
<backend.jpa>
name
< Back
Next >
Cancel
Finish

This page show two checkbox-lists. On the left hand side all the so called template tree branches that belong to the templates projects from the first page are displayed. A template project contains exactly one **template-tree.** A template tree contain one or more **template-branch(es)** and each branch contain one or more **template(s).** This setup allows to group certain templates that logically belong to each other but can still be applied individually. A good example (that is also provided as an example in the zipfile), is a template project called **webapp.** This project hosts a template tree with template branches for generating the:

- the Java backend
- a Dojo based frontend
- or a Flex based frontend
- or a ZK based frontend etc.

When you click one of the (template-tree) checkboxes on the left hand side, the corresponding template branches will appear on the right hand side. See:



Templates

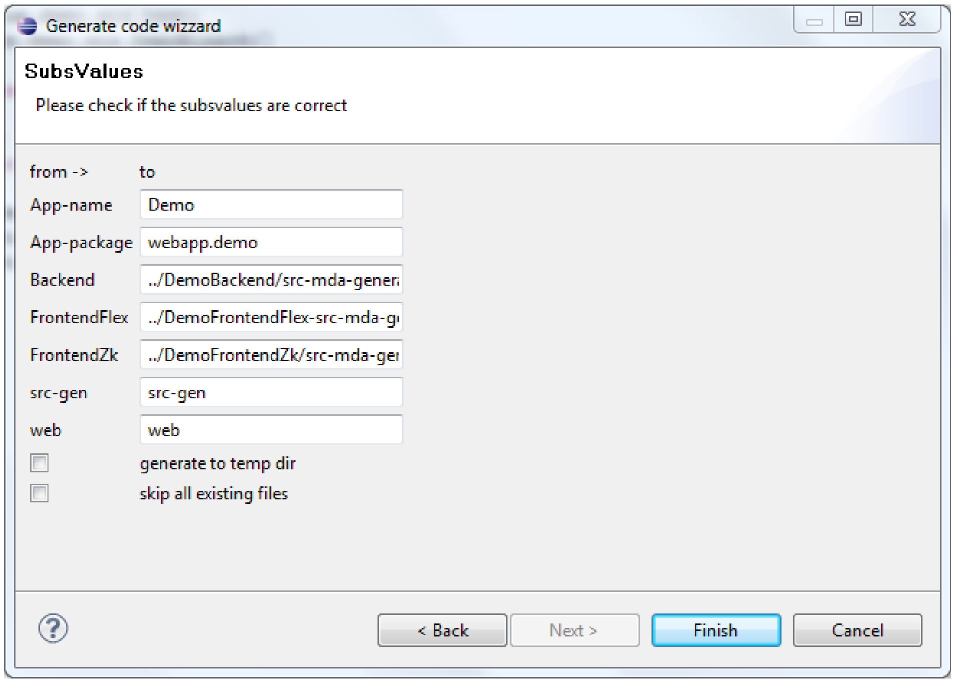
Please select one on more template(s)

Teplate tree's and branches	
	name
☐	/media/c/dev/eclipse-linux/helios-easymda
✓	<wizzard>
☐	/media/c/dev/eclipse-linux/helios-easymda
☐	<backend.jpa>

Templates	
	name
☐	<wizzard>
☐	WIZZARD_MAIN
☐	WIZZARD_DIALOG
☐	WIZZARD_PAGE
☐	WIZZARD_TEST

What templates exactly will pop-up on the right side, depends on the classes or packages that were selected earlier. For example, try to experiment by canceling the wizard and then select just one particular class file in the model project. Depending on what you selected, different template(s) will be shown on the right side.

Initially all templates are unselected. You can select a branch, this will automatically select all child templates, or you can select one or more templates individually. At least one template should be selected before the [Next] button gets enabled. After [Next] the last wizard page appears:



SubsValues

Please check if the subsvalues are correct

from ->	to
App-name	Demo
App-package	webapp.demo
Backend	../DemoBackend/src-mda-gener
FrontendFlex	../DemoFrontendFlex-src-mda-g
FrontendZk	../DemoFrontendZk/src-mda-ge
src-gen	src-gen
web	web
☐	generate to temp dir
☐	skip all existing files

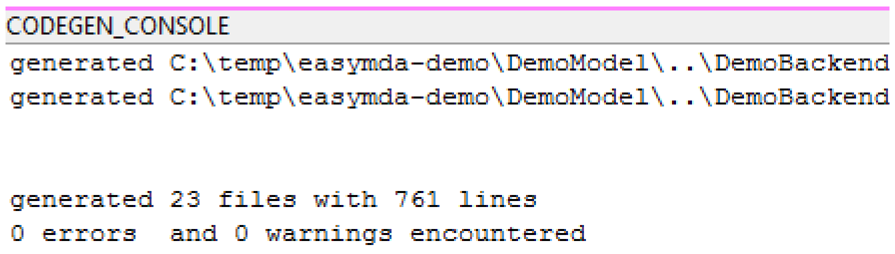
The last page wizard, shows all the variables that are used in the templates that you selected just before. So depending on the templates that you selected, different key/value pairs may appear, but always, the key/value pairs that are used to generate the output filename(s) for the generated code will be presented. The first time that you generate code for a particular model project some values may empty, and must be filled before the [finish] button will be enabled.

In the example above a complete folder name was provided: \home\project\mytarget
or in Windows, something like c:\project\mytarget
Alternatively, it is also possible to give a relative path, (like we did in the demo before) for example: ..\mytarget . In this case the folder-name will be relative from the selected Java model-project.

> Note:
> All values from the last wizard page, are read from, and written to the file:
> **<model-dir>\easymda.ini**.
> This is a special ini-file. . You could edit the file before you start the easyMda plug-in, so that the wizard pages has the correct values immediately.

Click [Finish] button.

When the finish button is clicked the generator will loop over all selected classes or packages and then loop over all the selected templates and generate the corresponding files. And show summary information in the console:



```
generated C:\temp\easymda-demo\DemoModel\..\DemoBackend
generated C:\temp\easymda-demo\DemoModel\..\DemoBackend


generated 23 files with 761 lines
0 errors  and 0 warnings encountered
```

More detailed information can be found in a logfile under the selected model project i.e. <model-project>\easymda-codegen.log

Now refresh the demo project (Project → Refresh or F5)
Now java file should appear under the src-mda-gen folder

Select the generated class: TestWizzard and Run as → Junit test.
This start the newly created wizard.

5 Create your own wizard model project

In the previous quick-start we used an already prepared model project. In this chapter I show you how easy it is to create you own model project, that can be used to generate a new wizard. We will also create a new output project that will contain the generated wizard.

1. Create a new Java project:
   File - new - Java project
   Enter a location and name keep all default values.
2. Add the libraries that you may need
   Right-click project - Properties - Java build path – Libraries
   now add all the jar files under: <easyMda-dir>/samples/DemoLibs/lib/

The project that you created above will be the model project, hence in a role similar to UML diagrams or DSL files in other Mda tools. A model project in easyMda is very 'relaxed' there are hardly no specific requirement, you only need to:

- Add the necessary libraries depending on cartridge that you intend to use. (not really an easymda requirement)
- Create exactly one class that implements the interface: IApplicationType.
- And in order to have a meaningful Model, you should create some class(es) that make up the actual model!

Required IApplicationType class

The required class that implements the IApplicationType interface, is used by the generator to collect some overall model data. Depending on the wizard different values may be collected, but because all IApplicationType interfaces extend from ApplicationBaseType, it will always collects the so called 'base-package' name see for example:

```
package flca.demo;
  public class MobileApp implements IApplicationType {
  @Override
  public String getBasePackage() {
    return this.getClass().getPackage().getName(); //hence: flca.demo
  }
}
```

In this example the value: "flca.demo" will be returned. This will initially be the default value on the Gui text field labeled with "App-package". If you change this value to fo example:

```
com.myapp.wizard
```

than the code will be generated under:

```
<target-project>/src/main/mda-gen/com/myapp/wizard/xxx
```

Wizard classes

A wizard is some kind of container that contains one ore more wizard page(s). Let's create some page(s) first.

1. Create a new class, that implements IWizzardPage
   Note: pick a package and classname carefully, because these will be used in the generated wizard as well.
   Eclipse will create a class, that look's similar to this:

```
public class X implements IWizzardPage {
  @Override
  public String getName() {
    return null;
  }

  @Override
  public String getTitle() {
    return null;
  }
}
```

Replace the "return null" with some valid values. These will be used by the generator.
But what about the fields that should appear on this page, why is there no method like:

```
  @Override
  public List<Field> getFields() {
    return null;
  }
```

Well that is indeed one of the possibilities that could be used to provide the generator with information about what fields should be added on a page.
But here we use another way to provide the fields, simply by adding properties to class. The final result may look like this:

```
public class Page1 implements IWizzardPage {
  @Override
  public String getName() {
    return "CustomerInfo";
  }

  @Override
  public String getTitle() {
    return "Enter customer information";
  }

  String firstname;
  String lastname;
  Date dob;
}
```

The generator will use the java reflection api to create the fields on this page, and create the necessary:

```
Label   TextInput
  :
```

code. With the supplied class, the generator can only use the name of the property to define the label, so the result will look like:

```
firstname :     [                    ]
lastname :      [                    ]
dob:            [                    ]
```

That may be okay for first, and lastname, but obviously “dob” is rather vague, so how can we tell the generator to supply a proper prompt?

For this requirement (and many many others), the power of java annotations come to help.

Simply add the correct annotation, and fill in the details. How?
Put the cursor above for example the “dob” property. Insert the “@” character and then “mda.” . Eclipse build-in intellisense will popup all the annotations that start with “mda.” then obviously for this example you should pick the “@mda.annotation.wizard.PageField” annotation.
Next insert “()” and then the intellisense will show all the annotation fields. In this case there is just one required field: “label” , enter a value, and the result may look like:

```
        @mda.annotation.wizard.PageField (label="date of birth")
        Date dob;
```

You may repeat these steps for the other properties.
After you have created this class, you have seen the three cornerstones of the *easyMda* code generator:

- Inheritance (more specifically interfaces)
  By implementing methods that are provided the interface(s), the generator can be supplied with a lot of information.
- Annotations
  This is more or less similar to implementing interfaces, but on much fine grained level. In this example we used just one annotation with just one (simple) field, but in fact many different annotation(s), with much complex field(s) may be applied to a property.
- Reflection
  Using the ability to deduct (detailed) information about properties and/or methods from a class or interface, gives the generator the possibility, to generate whatever can be imagined.

As always, there is no single way to achieve something, and the same is true for *easyMda.* Different techniques could be used to archive the same result. We already saw just one example, how to provide the generator with info about the fields that it should generate. Both inheritance and reflection are good candidates, and even annotation(s) could be used, but the latter is probably not the best approach to model the page fields.

We have now create one wizard page. Repeat this step for a number of other wizard page(s)
Next create a class that implements IWizzard, that will look like:

```
public class MyWizzrd implements IWizzard {

  public String getTitle() {
    return null;
  }

  @Override
  public String getImage() {
    return null;
  }

  @Override
  public List<IWizzardPage> getPages() {
    return null;
  }
}
```

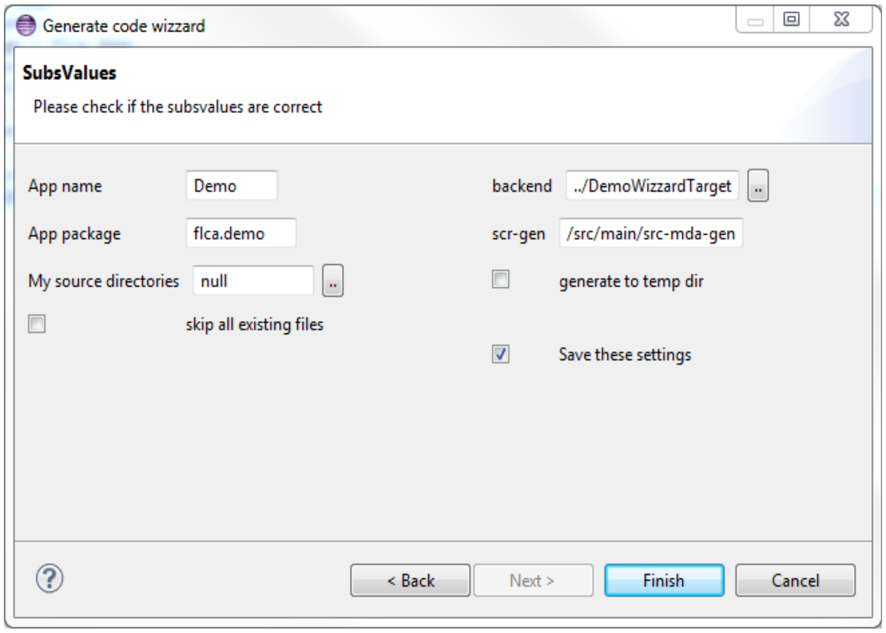
The first two properties are self explanatory. The last one is a little bit different, but easy enough as well. The getPages() should return an (ordered) list of pages that we created earlier, so something like this:

```
  public List<IWizzardPage> getPages() {
    List<IWizzardPage> result = new ArrayList<IWizzardPage>();
    result.add(Page1);
    :
    return result;
  }
```

Okay you model is now ready!

We can now generate the source code, like we did before, but this time in a new java project.
So first create a new java project.
File → new → Java project ... and give any name and location that you want.

Now we are ready to generate source into this new project. Select in the package explore the package from the model project you just created before, then right-click and hit ***"generate code".*** Select the same templates in the first two wizard page like in the previous example. But in the last wizard page, now enter the location of the **src** directory from the new java project you just created. Again, the easiest way is to copy/paste this name using the properties of this folder. So you get something like this:



SubsValues

Please check if the subsvalues are correct

App name	Demo	backend	../DemoWizzardTarget
App package	flca.demo	scr-gen	/src/main/src-mda-gen
My source directories	null	☐	generate to temp dir
☐	skip all existing files	✓	Save these settings

Now click the [Finish] button, you can see the progress in the console window. Refresh the new java project in order to see the generated code. You will now probably get a lot of compile errors, that is because we have to change some settings in this project before we end up with valid java code. To fix the errors we have to two things, correct the source location and the necessary libraries:

- Right click the project → Properties → Java build path → Source (tab)
  remove the **src** dir and instead add the **src/src-gen** folder.
- Right click the project → Properties → Java build path → Libraries (tab)
  Add the libraries <todo>
  junit-4.8.jar (or other compatible version)

Now all compile error should be gone, and you able to run newly created wizard!

6 Cartridge to help building cartridges

The project flca.mda.cartridge.template is very simple, but useful, cartridge that helps enhancing other cartridges. The goal of this cartridge is not generate a complete, or even partial, application. Instead it only generates one template plus a text file that contains code you can cut/paste into existing source code files.

Every time when you create a new template in an existing cartridge project, you basically have to prepare the following:

- Create the new template itself
- Create the code to register this new template.

Hence a perfect candidate to automate with easyMda!

The process is identical to what we have done before:

- Import the project DemoTemplateGeneratorModel from <easymda-dir>samples folder into Eclipse.
- Within the DemoTemplateGeneratorModel project, via the package explorer select all the flca.cartrdige.* pacakges.
- Right click → generate-code
- In the Gui use all default values and hit [Finish]

In this case, the output files are not generated in a separate project, but in this model itself under the “src-gen” folder. Here you may find:

- A template jet file.
- A .txt file with source code that you can cur/paste to the actual java files.

Note

Once you have a working example of code snippets that you have to create over and over again, with easyMda it is very easy and fast to convert this in a functional cartridge. So rather than trying to create a (complex) cartridge that 'does it all', it is maybe

You may wonder why, with this cartridge, the last page does not show any fields, like the other cartridges. Rather than using the interface to collect all values one could also collect this data interactively with the Gui. Using an interface has the advantage that it can collect more complex data in a type-safe way. But in this case the reason I opted for an interface was because of the way I am using this cartridge myself. Rather than starting the plugin, I always edit the same model file: GenerateTemplate.java run a junit test and I'am done!

7 Generate, deploy & run a crud application

In this chapter we are going to explore a much more ambitious cartridge, the 'webapp' cartridge. This cartridge is able to generate a fully functional crud application, that consists of a (simple) Java backend and a choice of frontends. The backend has the following characteristics :

- Simple Java based application not using a specific framework like Spring or (the full) JEE stack.
- It makes use of the JPA spec using Batoo
- Rest based services using Jersey.
- Runs under Tomcat or Jetty (embedded)

Currently the following frontends are supported:

- Html/Javascript based on Dojo
- Flex
- ZK

I want to stress once more that my objective for easyMda was no to generate the best application, but to make the code generator itself as easy, flexible and powerful as possible. During that process I was happy to use frameworks that a came across in my daily work as test cases for my easyMda project.

> Note
> I would very much welcome any complete fully functional crud application, in order to make a new cartridge that can generate it. I am particularly interested in a Scala based backend application.

For this quick-start I have prepared a number of projects in order to jump into the interesting bits a quickly as possible. The overall process is more or less identical to the process we have seen before when generating the wizard, but this example is more elaborate because:

- more projects are involved
- this time we have to prepare the libraries as well
- there are much more generator options in this example.

The goal in this example is to create a fully functional crud application running under Jetty embedded using the H2 database with a HTML/Javascript based frontend using Dojo.

1 Preparing the Eclipse projects

First thing to do is open all relevant projects under Eclipse and prepare the workspace so that we get rid all project errors.

Import the following projects (from the easymda-1.8.zip file) into Eclipse (file - import - general - existing application ..) :

DemoWebappModel, DemoWebappBackend, DemoWebappDojo

Note
In the developer's guide is explained how you can setup a similar project structure very quickly from scratch

After you imported the projects above, you probably get a lot of errors because of missing libraries. In order to fix that we need get all relevant jar files. This can be achieved by, downloading the following zipfiles:

ext_m3_repo-1.7.zip and flca_m3_repo-1.8.zip [1]

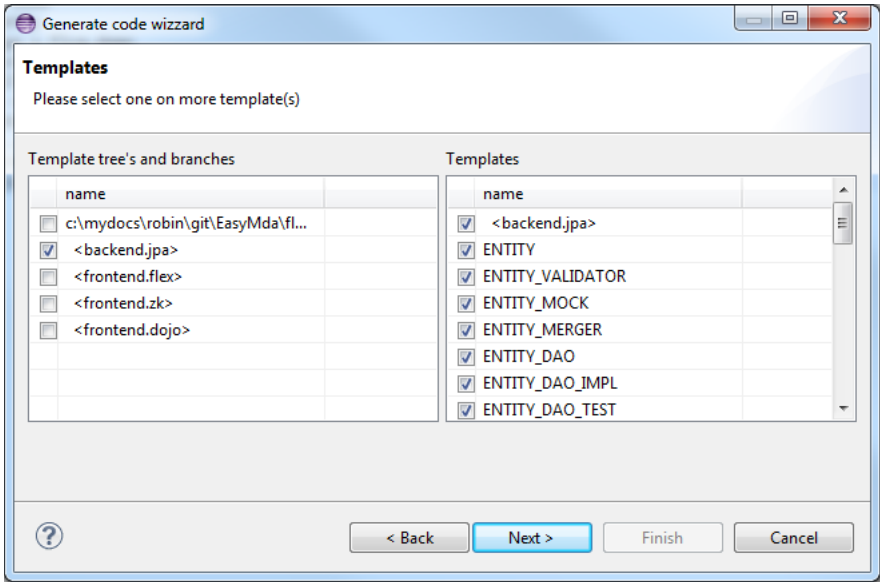
and unzip these to any (**same!**) location you want. You can see that these jar files are organized following the Maven convention.

Finally we have to create a Java symbol in Eclipse to reference the jar files from above. This is done via:

Window - Preferences - Java - Build path - Classpath variable and then create a new variable called: **M3_REPO** that points to the root of the maven repo we created above.

At this points all errors should be resolved.

In the next chapter we will explore the model much more in detail. For now let's generate and run the code first. Select via the package explorer all the packages from the DemoModel project, right-click and click on "generate-code", like we did with the before in the WizzardModel in chapter 3. We first start the backend code:
From the first wizard page, enable the flca.mda.cartridge.webapp-1.8.jar checkbox, hit [Next], then select the <backend> checkbox on both the left and right-side listbox, like this:


Generate code wizzard
Templates
Please select one on more template(s)
Template tree's and branches
Templates
< Back
Next >
Finish
Cancel

Template tree's and branches:

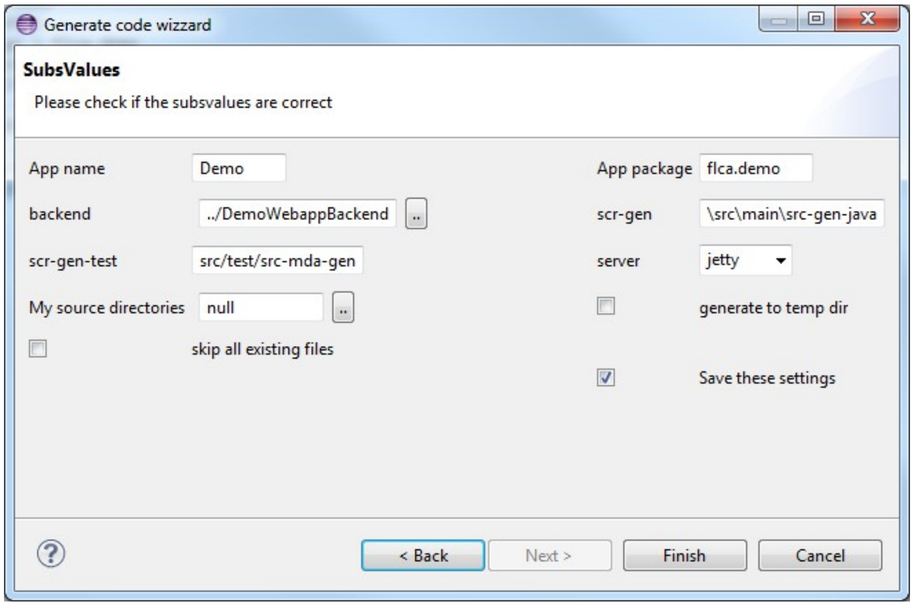
	name
☐	c:\mydocs\robin\git\EasyMda\fl...
☑	<backend.jpa>
☐	<frontend.flex>
☐	<frontend.zk>
☐	<frontend.dojo>

Templates:

	name
☑	<backend.jpa>
☑	ENTITY
☑	ENTITY_VALIDATOR
☑	ENTITY_MOCK
☑	ENTITY_MERGER
☑	ENTITY_DAO
☑	ENTITY_DAO_IMPL
☑	ENTITY_DAO_TEST

1 I split the external jar files into two separate files, because the flca_m3_repo-xx.zip will change much more frequently than the ext_m3_repo-xx.zip file.

Hit [Next] to show the last page to fill in the last variables:
Leave all values as is for now. So you get something like this:



SubsValues

Please check if the subsvalues are correct

App name	Demo	App package	flca.demo
backend	../DemoWebappBackend	scr-gen	\src\main\src-gen-java
scr-gen-test	src/test/src-mda-gen	server	jetty
My source directories	null	☐	generate to temp dir
☐	skip all existing files		
		☑	Save these settings

Click [Finish] to generate the code and refresh the target-project (F5)
Make sure that the target-project contains the source directories that you defined above.
Normally all generate-code should not contain any errors and very few warning if any. If you still see a lot of compile errors than you probably forget to create the Java symbol M3_REPO pointing to the m3_repo from all-jars.zip file .

So let's examine quickly what we just generated.
For each entity a number of classes is generated including a junit test to test the (also generated) corresponding Dao layer class for this entity. For example for the entity class flca.demo.data.C the class flca.demo.data.dao.TestCDao is generated. You can run this test immediately and should execute successfully. In the next chapter I will examine in detail how the database is configured. This demo is prepared for the H2 database.
Also for each modeled service a junit test is generated, for example: flca.demo.srv.TestTstService.
You can run this test now, but will result in (many) errors because we did not implement the actual code yet.

We leave the backend for now and concentrate on the frontend.
From the model project, once more select all the packages and (via right-click) generate code. But on the second page, we now select <dojo-frontend> on both the left and right hand listbox. In the third page leave all values as-is for now and hit the [Finish] button. After you refresh the WebappDemoDojo project, you will see a large number of generated files under the '/js' folder.

Notes

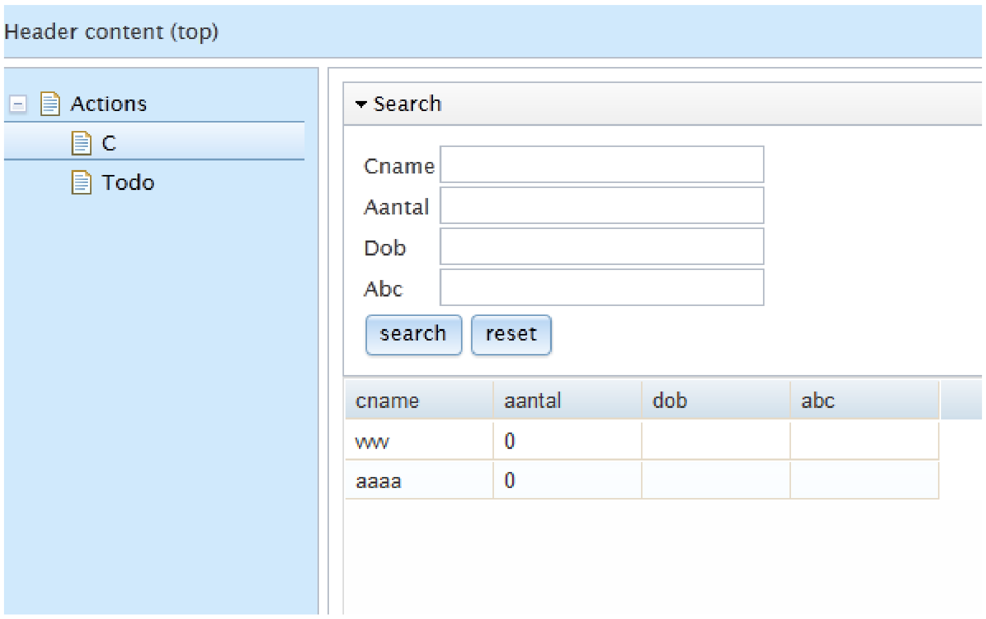
1. To develop Javascript code, I am currently using the AptanaStudio ide, but I may switch to Ideal Webstorm if this has good support for TypeScript.
2. In this demo (unlike the backend), not all files have been generated. There are a few 'manual' files, that give access to the actual crud functionality. In the next chapter I will describe ithese manual files in detail.

We are now able to run the generated crud application by starting the main class: flca.demo.DemoLauncher

This class will start the embedded Jetty browser under port 8070. You can access this application via:

http:/localhost:8070

this will display:



cname	aantal	dob	abc
ww	0		
aaaa	0		

8 The webapp cartridge, in detail

In this chapter we will walk through the WebappDemo project once more, but this I will explain what is the role of all the classes in the model project, what classes are being generated, how to configure the target application to use other database(s) and the 'manual' javascript files.

The easyMda distribution contains the 'webapp' cartridge with currently three branches:

- java.backend
- dojo.frontend
- flex.frontend
- zk.frontend

The model project

We have seen that to model your target code, we basically have the following tools:

- interfaces
- reflection
- annotations
- special api, like substituting values.

In the developer' s guide is explained how to define your own set of tools. Here I describe how these 'tools' are implemented in the webapp cartridge.

Interfaces

The webapp (currently) only has the following four interfaces:

- IEntityType
  This one is used for classes that should be persisted.
- IDtoType
  For classes like data transfer objects or value object that will not be persisted.
- IServiceType
  For services.
- IApplicationType
  This is the one required model class that provides model specif data. Here we need to implement two methods: getBasePackage() and getRestBaseUrl()

You may also find some java classes in the example, that do not implement any interface at all!, yet these will be 'generated' as well. This is true for a for some 'special' classes:

- Enums
- Exceptions
- Classes or interfaces that are annotated with @CopyFile <todo>

These files are copied as-is to the target location. All other files should implement one the interfaces mentioned above.

Reflection

Java reflection is used in IEntity and IDto classes to process all properties and in IService classes to process all methods.

Annotations

All of the class properties and/or methods may be annotated for fine-grain control. Hence the largest variety is in this area.

Note

All annotations inside the templates, start with: **mda.annotation**
This makes it easy to find the right annotation by entering @mda.anno and then Eclipse's intellisense will make live easier.

Here is list of the current supported annotations (note the prefix mda.annotation. is ommitted)

Annotation	purpose
AnnotationLiteral	Copy the annotation as is.
JavaDoc	Add javadoc Note it is much easier however to simply put Javadoc above the property or method, this will be copied as-is
jpa.Basic jpa.CascadeType jpa.Column jpa.Entity jpa.FetchType jpa.Id jpa.Inheritance jpa.InheritanceType jpa.JoinColumn jpa.JoinColumns jpa.JoinTable jpa.ManyToMany jpa.ManyToOne jpa.OneToMany jpa.Table jpa.Transient jpa.UniqueConstraint jpa.Version	These are annotations that are similar to the equivalent javax.persistence.xxx annotations. I have copied these annotations to the webapp cartridge under mda.annotation because I the end-user to be able to find all supported annotations via @mda.annotaion ...
crud.Crud crud.CrudOperation	Special annotation to generate standard crud operations for entities.

crud.Search	
RestService RestMethod	Class annotation to define the context name for this Rest service. Method annotation, to indicate if this method should be a http GET or POST
DatabaseUnit	Optional annotation used in IEntity types, to define the datasource other than the 'default'.

Most of the annotations are, in combination with the example, self explanatory. In the section below I will therefore only describe some particular items.

JPA annotations

The majority of annotations mimic the standard JPA specification, hence you should consult this JPA spec for a complete overview. Below I will outline the most important aspects.

The example model-project to create a crud applications, is also the reference model project to enhance and test the webapp api. The classes do not represent a typical business application, with example customers, orders etc. Instead these are 'abstract' (not the Java abstract) classes that contain all kind constructs that can be generated. So class A contains all supported primitives and relations. The related class resp B, C, D and E represent different kind of relations:

- One2Many relation
- Many2One relation
- Join table

Inside an entity (indicated with the interface IentityType), a relation can be represented with either an explicit annotation (@mda.annotation.jpa.ManyToOne or OneToMany) :

```
class A {
  @ManyToOne ()
  B b;

  @OneToMany()
  Set<C> sList;
}
```

Note

The ManyToOne results in a table A with foreign key field pointing the primary key of B. Hence many A instances may contain the same (1) B instance.

The OneToMany results in a table C that contain a foreign key that point to the primary key of A. Hence one instance of A may contain may different instances of C. For this relation it is therefore mandatory that inside class C also an relation exists that is the other way around like this:

```
class C {
  @ManyToOne()
  A a;
}
```

Note
see the appendix Jpa annotations for a more detailed info and examples.

Note
A relation is only valid if the related class is also a class that can be persisted, hence implements the IEntityType interface. So for example relations like:
List<Integer> numbers;
are invalid[2]. If you need this relation, you should create a class for example like this

```
class MyNumber {
  Integer number;
  }
```

and use this class like this:

```
List<MyNumber> numbers;
```

Crud operations

In many applications you will have some standard crud operations for entity instances. The webapp templates make it is easy to generate these standard crud operations. This can be achieved with the class annotation:

```
  @Crud(service=MyService.class)
```

this is the simplest variant, it will generate the following crud methods:

- <IEntityType> retrieve(Object aPk)
- <IEntityType> save(EntityType aSource)
- Set<IEntityType> findAll()
- void remove(Object aPk)

You have control over what methods are generated with 'operations' property, see:

```
@Crud(service=MyService.class, operations={CrudOperation.FindAll,
CrudOperation.Retrieve})
```

A special 'crud' operation is the @Search annotation. This annotation, also requires a class (Dto or Entity) that will be used to pass the search argument. Example:

```
@Search(entity=A.class, searchArguments="SearchA.class)
```

You can also indicate for both the findAll() and search() method, if the result should be wrapped inside a paging object.

2 At least for jdbc based persistence. For OO database such relations are ofter valid.

Note

The current the webapp does not generate paging code yet.

JavaDoc

There is a special annotation to generate javadoc in the output code:

@JavaDoc (doc="your documentation ...")

Alternatively you can simply put java above the property and/or methods and this will copied as is.

The target backend project.

The following source code files will be generated, depending on the interface it implements, **note** that the output filename will be based on the following assumptions:

entity classname	: flca.demo.entity.A
dto classname	: flca.demo.dto.TstDto
service interfacename	: flca.demo.srv.TstService
Application name	: flca.demo.Demo
Base-package	: flca.demo

Interface	Output filename
IEntityType	flca/demo/A.java flca/demo/AValidator.java flca/demo/AModel.java flca/demo/test/AMock.java flca/demo/AMerger.java flca/demo/dao/ADao.java flca/demo/dao/ADaoImpl.java flca/demo/test/TestADao.java flca/demo/test/ADao.java
IDtoType	flca/demo/dto/TstDto.java flca/demo/test/Dto1Mock.java
IServiceType	flca/demo/Srv.java flca/demo/test/SrvMock.java flca/demo/test/TestSrv.java flca/demo/SrvIntf.java flca/demo/SrvImpl.java flca/demo/ExtSrvIntf.java
IApplicationType	flca/demo/DemoBinder.java flca/demo/DemoBootstrapper.java

	flca/demo/DemoConfig.java flca/demo/DemoConfigProviders.java flca/demo/DemoConstants.java flca/demo/DemoContext.java flca/demo/DemoDbsProvider.java flca/demo/DemoExtServiceBase.java flca/demo/DemoLauncher.java flca/demo/DemoModule.java flca/demo/DemoServletConfig.java flca/demo/DemoTestServiceBase.java

So in general for all IEntity classes, the entity itself, the DAO layer, validator class, a mock object and a jUnit test, to test the Dao layer are generated. One important topic I did not mention so far is how to define the target database(s). This will be explained in the following paragraph along some other application configuration topics.

Application configuration and database setup.

In the list above you can see that most of the generated classes are 'application-wide' classes in root of the provided application package. But whereas for each new entity dto and/or service new classes are generated, the number of these application classes remains the same. The purposes for these classes is facilitate the following:

- CDI pattern
- environment specific properties
- database configuration
- bootstrapping the application.

This will be explored more in detail below.

CDI pattern

This application makes use of the CDI pattern based on Google Guice. A number of generated classes implement this Guice code, see:

DemoModule
This is the class that extends the AbstractModule, but the actual bindings are done in the class DemoBinder because doing so we can reuse this class in DemoServletConfig.

DemoBinder
Here the actual binding are defined. For some bindings DemoBinder relies on two other (also generated) classes: DemoConfigProviders and DemoDbsProvider. These will be explained below.

DemoServletConfig
This class extends GuiceServletContextListener which a placeholder where you can put your binidngs, but instead the DemoBinder from above is used here.

Environment specific properties

DemoConfig

This class extends ConfigBase defined in the flca.frw.common project. This base class is a wrapper for Commons Configuration. With Commons Configuration you can easily access values from properties, ini and xml files. ConfigBase makes this even easier because rather than defining each file individually, here you only provide the root path under the classpath and/or the root path on the file-system. These are than scanned in a specific order and all properties, ini and xml it finds are than cached and from that point accessible via one api. DemoConfig is injected via Guice (via bindings in DemoBinder) but the actual values are defined in this class:

DemoConfigProviders

This class implements DemoConfigProviders that has the following methods:

- Provider<List<String>> getFileSystemDirs()
- Provider<List<String>> getClasspathDirs()
- Provider<List<String>> getScanJars()

You may modify this (generated) class yourself if you want to make use of environment specific variables, by implementing the methods above, You may provide one or more root-path in either the classpath or file-system. These path(s) are scanned in a particular order and more specific files may overwrite values it found earlier. The scan order is like this:

- First the classpath(s) from right to left.
- Last the file-system paths(s) again from right to left.

Hence the most specific file is the first (or left) file-system path. The path(s) that you provide may contain system variables like $HOSTNAME.
A typical example how this can be used is the following pseudocode:

```
getClasspathDirs() = { "/config" }
getFileSystemDirs() = { "/$HOSTNAME/demo/config", "/var/demo/config" }
```

What you achieve with this setup, is that you put common config setting under the classpath somewhere under the "/config" folder. These may be overwritten by files found under /var/demo/config which may overwritten again by files found in folder that is specific for each deployment. It is no problem if a given path does not exists, although you get a warning message during startup. Hence a configuration like:
{ "C:/data", "/home/data", ... } is perfectly valid on any environment.

One method I ignored so far is this one: getScanJars(). So far when I talked about a configuration file it implied a file on the file-system. It is however also possible to store configuration file(s) inside a jar file that is in the classpath. Via the getScanJars() you can indicate which jar file(s) to scan. A null or empty list indicates that you don't store values in a jar files. When you supply "*.jar" you scan all jar files, including all framework jar files, or you provide more specific jar file(s).

Database configuration

In the previous Crud demo, magically the entities were stored in some database, without any configuration from our side. How is this achieved and can we setup the App to use another database?

First of all it is important to know that this Crud app makes use of the standard JPA spec implemented by Batoo (see batoo.org). Batoo in its turn makes use of connection pooling framework from BoneCP.

Secondly this Crud app is able to work with more than one datasource(s), and I also started with backend functionality to also support not only JPA databases but other datasources as well. I am planning to support (on of my favorites) the Perst OO database in the near future. And after that one of the popular No-SQL (file-based) datasources like Cassandra.

When you model an entity class, you may annotate this class to indicate what datasource should be used to persist this entity. In the demo examples we have seen so far, we did not use this annotation, which falls back to the default annotation like this:

```
@mda.annotation.DatabaseUnit (name="default" type=PersistenceType.JPA)
```

So if we don't specify any DatabaseUnit in an IEntityType model, we tell the generator to generate-code for this class using JPA persistence with datasource "default". This name should correspond with a unit name in the standard JPA **persistence.xml** file. For example:

```
<persistence-unit name="default">
  <provider>org.batoo.jpa.core.BatooPersistenceProvider</provider>
<class>flca.demo.entity.A</class>
<class>flca.demo.entity.B</class>
<class>flca.demo.data.C</class>
<properties>
        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
        <property name="javax.persistence.jdbc.url"
                value="jdbc:h2:tcp://localhost/~/home/robin/Work/h2-test2" />
        <property name="javax.persistence.jdbc.user" value="root" />
        <property name="javax.persistence.jdbc.password" value="" />
</properties>
<exclude-unlisted-classes>true</exclude-unlisted-classes>
</persistence-unit>
```

If you do annotate your model entities with different datasources, the generator will generate the file DemoDbsProvider accordingly.
So how comes that the entities were stored in the database?
Easy! I simply put the (not generated) file:
src/main/resources/META-INF/persistence.xml
in the DemoWebappBackend project!
If you want to experiment with other database simple modify this file accordingly.

Note
On of the files being generated, are: Demo-Readme.txt and persistence,xml. The readme file tells about some manual steps that you may want to apply after the code has been generated. The latter contains generated <persistence-unit> sections, that you may want to in-cooperate with the actual META-INF\persistemce.xml file.

Bootstrapping the application

Depending on what server type you select on the generator gui page:

- Tomcat
- Jetty embedded

Different files are generated in root application package. If you selected the embedded Jetty server, the files:

DemoJettyLauncher and DemoServletConfig

are generated.

You may want to modify the DemoJettyLauncher to use another port add other listeners and filters etc. When you open up this file, you will find the following code snippets:

```
public static final String JETTY_PORT = "jetty.port";
public static final String JETTY_BASE = "jetty.base";
:
private static int getPort() {
    if (System.getProperty(JETTY_PORT) != null) {
        return new Integer(System.getProperty(JETTY_PORT)).intValue();
    } else {
        return 8070;
    }
  }
  private static String getResourceBase() {
    if (System.getProperty(JETTY_BASE) != null) {
        return System.getProperty(JETTY_BASE);
    } else {
        return "C:/mydocs/robin/git/EasyMda-target/easyMda-1.8/samples/DemoWebappDojo";
    }
  }
```

So you may either change this source code, or (preferable) modify the run-configuration with the following program arguments:

- -Djetty.port=xxxx
- -Djetty.base=<folder-contains-the-dojo-code>

The launchers also kicks the generated class : DemoBootstrapper

The DemoBootstrapper triggers Guice that triggers in its turn the DemoConfig and also the database conection pool(s) are initialized. In addition other initialization code can be executed like starting-up the database.
This code is generated as well, by substituting the value if it is found in a an ini-file under the model or target project. The webapp cartridge specifically looks for a section with the name “bootstrap” and the key “startup.commands” . And because I put the following file in the model-project, the generator was able to generate the code starts up the H2 database. See:

DemoModeProject /src/main/resources/user-config/bootstrap.ini :

```
[bootstrap]
startup.commands = {{
  protected void preInitializations() {
    logInfo("yourPreInitializations");
    startDatabase();
  }

  private org.h2.tools.Server h2Server;
  private void startDatabase() {
    System.out.println("starting H2 database" );
    try {
          String args[] = new String[] {"-tcpAllowOthers"};
          h2Server = org.h2.tools.Server.createTcpServer(args);
          h2Server.start();
          System.out.println("H2 database succesfully started" );
    } catch (Exception ex) {
          System.out.println("eror start H2 " + ex);
    }
  }
}}
```

As you can see, this ini-file slighly differs from a normal ini file. This one support a values that spans over multiple lines using the :

```
key = {{
  value1
  :
}}
```

syntax.

> **Note**
> You should not use such an ini-file with this special syntax in site specific configuration, that we discussed before. That one is implemented by Common Configuration that only supports 'normal' ini-files.

Not generated Javascript files.

I mentioned that in contrast to the backend, not all files on the Dojo frontend target were generated. If you look at the DemoWebappDojo project in the easymda-1.8.zip, you will the following files:

- index.html
- under js/base/:
  the files: main.js, BaseCtrl.js, mainLayout.js and LeftTree.js
- style/style.css
- The file: /resources/tree.json
- Under /resources/json-mock :
  some json files.

Index.xml triggers main.js that in its turn triggers BaseCtrl.js. This script setup's the simple surrounding 'environment' for the generated crud functionality. This a standard window layout with a tree view on the left hand. This tree is populated with the file: resources/tree.js This file contains a number of records under the 'children' tag like this:

```
{
  "name": "A",
  "id": "id_A",
},
```

The 'name' will be displayed in the tree, and when you click this, it generates an event that ends up in the BaseCtrl script in one the actions defined like this.

```
actions["id_A"] = {
    exec : function() {
         Acontroller.showGrid();
    }
  }
```

So if you make a new model class like Customer, you should update both the tree.json and BaseCtrl.js accordingly.

Besides tree.json, there are also a number of json files under resources/json-mock. These are test responses that you can use instead call a json Rest service. Hence you could test the frontend with starting the Webserver!

This can be achieved by setting the constant MOCK_SERVICES in DemoConstants.js to **true.**

This value is used in for example AControler.js like this:

```
getServices = function() {
  return AppConst.MOCK_SERVICES ? TstServiceMock : TstService ;
}
```

Note
I did not bother to generate these files as well (it would have been easy enough), because in my experience, most applications do have crud functionality, but presented in a more hidden way. You seldom encounter a pure crud app like this Demo.

The (also) generated script TstServiceMock.js can function without a running Tomcat or Jetty server, because it get it's response from the corresponding json file from the resources/json-mock folder.

But how did these json file's got here !?

Well for this purpose, yet another file was generated in Java backend project :

DemoJsonMockGenerator.java (under src/test/src-mda-gen)

This is a jUnit test that will generate the necessary json files. Note that when this jUnit test the first time, it may give error messages indicating that you no need to implement a particular method. It does however generate all json files that belong to crud methods, using the (also) generated Mock objects.

9 What about already generated code

One complaint about MDA is that it is difficult to synchronize the model and source code. Ideally the model and source code can by synchronized both ways. There are a few MDA tools that are capable to synchronize both ways to some extend, Roo for example. But like most other tools, easyMda is also a one-way code generator. Luckily (because the model is also plain java) to synchronize the model once the source code has been modified, is very easy. Nevertheless, how do you manage the (common) scenario when you are long way along with development and weeks after the last time that you generated code from a large model, you need to implement some extra use-case(s) that are perfect candidates to generate, by enhancing the current model. What will happen with initially generated code, but meanwhile is potentially out-of-sync with the model.

First thing I want to say about this s the following. This scenario is indeed very likely to happen at large projects. Is that a good reason to say upfront that you can **not** use MDA in that project !?
I don't think so. If the MDA tool (like easyMda) you use, makes you very productive at the beginning of the projects, by generating thousands lines of high quality source code, use it! And accept that at a certain point during the development, the source code becomes king, and say farewell to the model that helped you so much in the beginning.

That is not to say, you should generate only once and then forget about it. In fact *easyMda* has a number of techniques that makes it possible to keep the model alive for a long time, or even all the way. That will be explained below.

Only generate what you need

The easiest way of course, is to make use of an important feature of easyMda, that is that it allows to generate:

- either for all packages all templates possibly generating hundreds of files.
- Or for one particular dto, entity or service, one particular template, that results in exactly one file.

Note
The source code of the last generated file is also put on the clipboard.

My experience is that at the start of a project you frequently generate all (or large portions) often, and soon you only (re)generate one class at a time.

Java merge

The template developer has a number of options, when he/she registers a new template for the so-called “mergeStrategy”. The default is **Merge** meaning that the code generator will eventually try merge the generated code with an already existing file. This merge tool does a very decent job so in most cases it will leave extra code that you added op updated intact.

Skip particular files

You can indicate inside a generate source file, that the generator should leave this intact, by inserting the keyword:

@KeepThisFile <TODO>

Note

This is not really an annotation, and can be inserted anywhere also inside a **comment**. Hence your generated code does **not** depends on the cartridge.

Another possibility, is to move a generated file(s) from that target folders to your 'manual' folder. So once you are happy with TstServiceImpl class, you could move it for example from the src/main/mda-src-gen to the src/main/java folder. You must than also tell the generator that it should look inside this folder to find out if it should skip generating the TstServiceImpl again. This can be achieved in the last gui page by populating the “user dirs” field with all the 'manual' folders. See:

<img>

Skip all existing files

You can tell the generator to leave all existing files as-is, by enabling the checkbox on the last wizard page. So only new files will be generated.
<screenshot> <TODO>

Organize your source code files well

I found it very helpful to organize the source files in such a way, that it is apparent what files are generated and what files are manual. See for example:
You have service CustomerService.
The generated source code with the implementation (CustomerServiceImpl) could be stored under the : src/main/mda-gen folder. Unfortunately the current version of easyMda is not capable to generate the implementation as well :), so obviously this generated file will eventually contain lots of manual code. But instead of creating this manual code inside this generated file, I prefer to manual create another implementation file (with a more or less similar name), under for example:

scr/main/java

And inside the generated implementation file, I only put one-liners to my own implementation file.

Use diff tool & Temp dir.

Another possibility to 'protect' your source code from mistakes while regenerating the code, is to make use of the feature of *easyMda* that it generates in fact many more files, then it shows in the summary log. In total the following files be created for each java model file / template combination:

- <tmp>/**merged**/<generated-package>/<filename>
- <tmp>/**unmerged**/<generated-package>/<filename>
- <tmp>/**backup**/<generated-package>/<filename>

The first folder (merged) contains the final generated source, the same it will output to for example the ${Backend} location.
The second folder contains the source before it is merged with the original file.
The last folder contains the original file.

If you select the check box on the last wizard page:

<img>

You force the generator to output all the files to the temp dir only. Using a Diff tool (like Meld or BeyondCompare) one can visual the differences between the original and generated files and synchronize that's what's needed.

10 Conclusion

I hope that you had fun while creating a number of applications in record-time.

Try experimenting with you own models, to get comfortable with the plug-in. If you taste for more, and want to alter the generated code, consult the developer's guide, this explains in depth how to do this.

Please let me know what you think of this plug-in, and sent me ideas what could be enhanced, and last but (unfortunately) not least, please let me know if you encountered any problem.

robin.bakkerus@gmail.com

11 Appendix Jpa annotations

To express relations between two entities the following annotations can be used:

```
mda.annotation.ManyToOne
mda.annotation.OneToMany
```

Below the most scenario's will be reviewed using a Person class that may have zero on more Address(es). The relation can be:

- bi-directional (so from the Address class we can directly retrieve it Person owner)
- uni-directional
- with or without a Join table (with a Join table the same address can be used by more persons)
- and the Person can have just 1 address of zero or more addresses.

Uni-directional, just 1 address

```
Person
@ManyToOne
Address homeAddress
Address
// nothing here
```

This results in the following tables:

Person		Addres	
ID		ID	
HOMEADDRESS_ID	Property name + _ID	...	
...			

Uni-directional, just 1 address, explicit column name

```
Person
@ManyToOne
@JoinColumn(name="address_id")
Address homeAddress
Address
// nothing here
```

This results in the following tables:

Person		Addres	
ID		ID	
ADDRESS_ID	Explicit name	...	
...			

Uni-directional, zero or more addresses

Person
@OneToMany
Set<Address> addresses
Address
// nothing here
This results in the following tables:

PERSON	ADDRESS	PERSON_ADDRESS (join table)
ID	ID	ADDRESSES_ORDER
..	...	PERSON_ID
		ADRESS_ID

This is identical with:
@OneToMany
@JoinTable(name="PERSON_ADDRESSES")
Set<Address> addresses

Uni-directional, zero or more addresses

Person
@OneToMany
@JoinColumn(name = "person") // this will not create a join-table
Set<Address> addresses
Address
// nothing here
This results in the following tables:

Person	Addres
ID	ID
..	PERSON

Bi-directional, zero or more addresses

Person
@OneToMany(mappedBy="person")
Set<Address> addresses

Address
@ManyToOne
Person person
This results in the following tables:

Person	Addres
ID	ID
..	PERSON_ID

Bi-directional, zero or more addresses with join-table

Person
@OneToMany(mappedBy="person")
Set<Address> addresses

Address
@ManyToOne
@JoinTable
Person person
This results in the following tables:

Person	Addres	PERSON_ADDRESS
ID	ID	PERSON_ID
..	PERSON_ID	ADDRESS_ID

Bi-directional, many-to-many with join-table

Person
@ManyToMany
Set<Address> addresses

Address
@ManyToMany(mappedBy="addresses")
Set<Person> person
This results in the following tables:

Person	Addres	PERSON_ADDRESS
ID	ID	PERSON_ID
..	PERSON_ID	ADDRESS_ID

Conclusions

- to create a bi-directional relation use the mappedBy
- A join-table is created when using a OneToMany (without an explicit JoinColumn) or whem using a MenyToMany