



*Take it easy,  
let your computer do the work*

*easyMda*

*Reactive cartridge guide*

*easyMda*

—

*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 Reactive cartri dge user&developers guide, version 2.0 27. Aug. 2014  
**robin.bakkerus@gmail.com**

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

## Table of Contents

1Intro.....	5
2Installation.....	6
3Quick-start, the Reactive demo.....	9
4Compile & Run the Scala backend.....	14
5Compile & Run the Dart frontend.....	16
6Notes on the Scala backend.....	19
How is data retrieved.....	20
The fd and ohc property.....	21
Generated classes.....	22
Classes that implement IEntityType.....	23
Classes that implement IServiceType.....	24
Classes that implement other interfaces.....	25
Generated project files.....	25
7Notes on the Dart frontend.....	27
Classes that implement IEntityType.....	27
Classes that implement IServiceType.....	28
Generated project files.....	28
Generated pub files.....	29
Executing Json/Rest calls.....	29
8Create your own Reactive model project.....	31
Implement the correct interface.....	31
IEntity types.....	32
IEntity methods.....	33
IEntityType limitations.....	33
IService types.....	33
IServiceType annotations.....	33
IServiceType limitations.....	33
IDtoType.....	34
Scala Compiler errors.....	34
Dart Runtime errors.....	35
9Developer's notes on the Reactive cartridge.....	36
Project structure.....	37
RegisterReactiveTemplates.....	37
The Reactive Jet engine files.....	39

The model project.....	39
Interfaces.....	39
Reflection.....	40
Annotations.....	40
JPA annotations.....	41
Crud operations.....	43
JavaDoc.....	43
10Conclusion.....	44

# 1 Intro

This document is a more compact and combined user's & developers guide that focuses on the new Reactive cartridge. For more extensive (background) information on EasyMda, please consult the 'EasyMda User's respectively the EasyMda developer's guide.

The Reactive cartridge, is called as such, because the generated backend source code is based on the principles of the Reactive Manifesto and makes use of the following Scala based frameworks:

- **Spray**  
This is used to handle the http requests, de- and encode the Json/Rest request data, and forward the request to the request handlers (below), all in an asynchronous way.
- **Akka**  
This is an actor based, concurrent and asynchronous message driven event loop system. It can run on a single machine or in cloud based environment.
- **Slick**  
This is one of the most popular Scala based persistence frameworks. It differs from well known Java /JPA frameworks like Hibernate, in the sense that it gives much more control to the developer and is much closer to plain sql.

This cartridge generates code, that makes use of Json/Rest, so any frontend client that can handle Json/Rest can be used (ex: Dojo, Flex, ZK and many more), hence I could have used one these from the Webapp cartridge. Instead I also created a brand new frontend for this Reactive cartridge, based on Google Dart. Why? because I think Dart is one of the most promising Gui frameworks that may become dominant on the Android platform. Combined with the Scala based reactive backend, this is (IMO) a superior alternative to the current JEE or Spring backend and Javascript, GWT(based), and or Flex frontend stack.

Note this version of the Reactive cart (version 2.0), is just the first version of this reactive cartridge. The "2" indicates a new major EasyMda update. Many improvements are foreseen in next releases due for Q3/Q4 2014.

## 2 Installation

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

1. At this point, I assume that you have unzipped the *easmda-reactive-2.0.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/flca.easmda.generator_2.0.zip <Eclipse>/dropins
```

Linux:

```
cp -rf <easymda-dir>/eclipse/dropins/flca.easmda.generator_2.0.zip <Eclipse>/dropins
```

3. TODO

4. In order to use the Jet-engine, which is used by the easyMda plugin, you may have install one extra Eclipse plugin: “EMF Eclipse Modeling Framework”. (depending on your Eclipse installation<sup>1</sup>) as follows:

Help → Install new Software

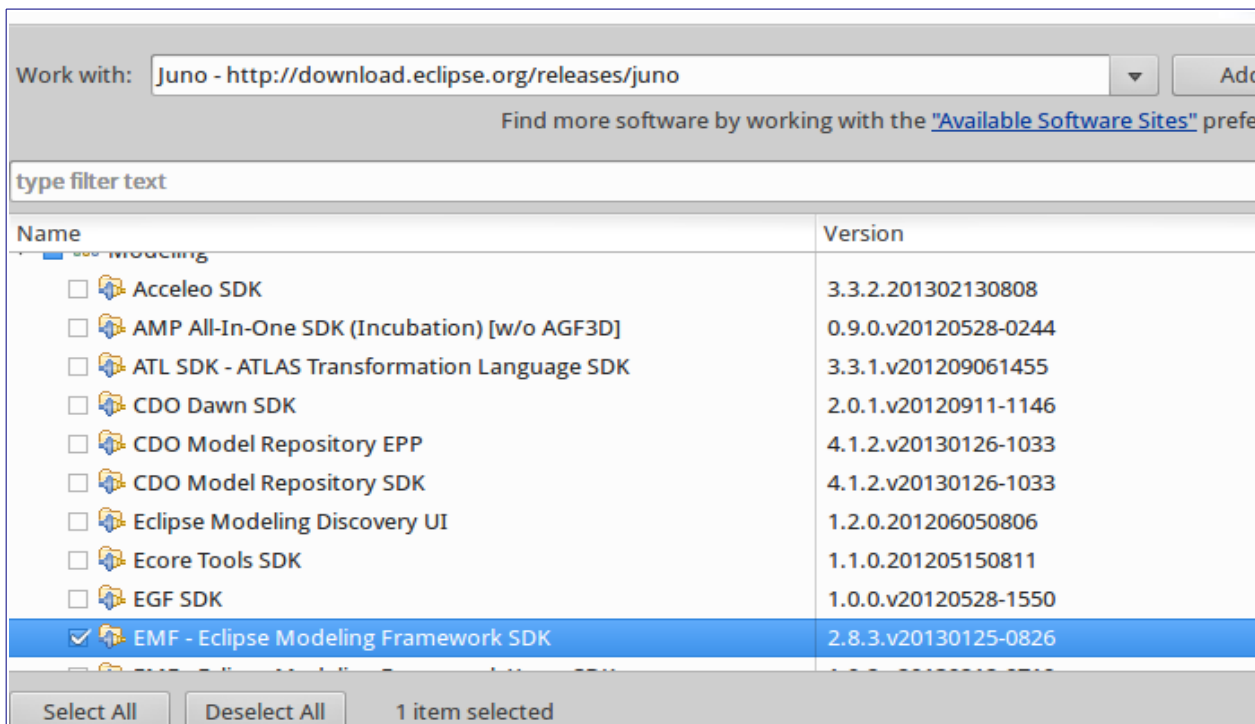
Work with: Juno - <http://download.eclipse.org/releases/juno> (or something similar)

unfold “Modeling” and select EMF Eclipse Modeling Framework.

see:

---

1 The Classic version doesn't has the EMF plugin whereas the JER version does had this plugin.



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 and  
Eclipse Luna 4.4.0  
Jdk1.7.0\_07
- Windows 7 with:

Eclipse : Juno 4.2.1 Release Build id: M20120905-2300 and  
Eclipse Luna 4.4.0  
Jdk1.7.0\_07



### 3 Quick-start, the Reactive demo

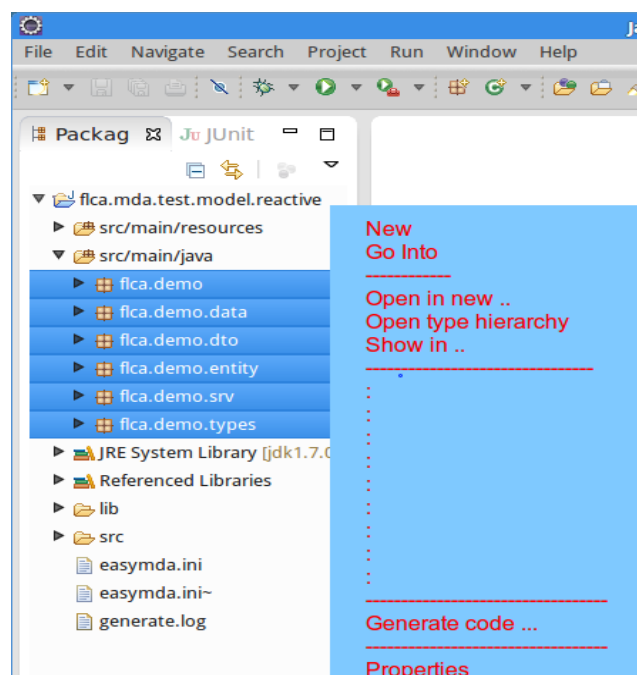
In order to see how this easyMda plug-in works, the installation contains a small demo model project as a quick-start. In this chapter we will rush through the demo. Simply follow the commands, and within a couple of minutes you should be able to see a fully functional crud application.

Later on we will do the demo over again, then I describe more I detail what is going behind the scenes.

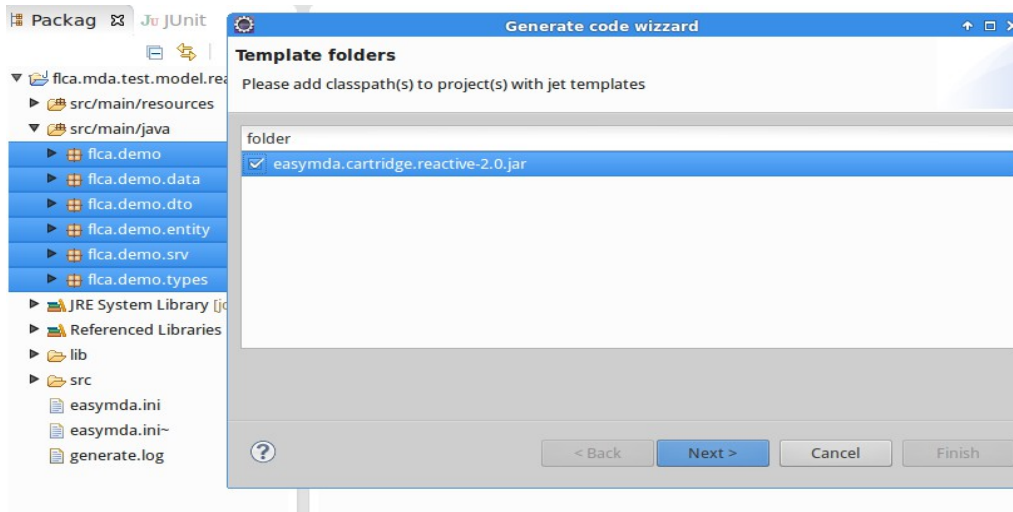
Note

If your run this demo for first time, please follow the instructions exactly as outlines. In particular don't change the prepared values for 'App name' and 'App package' in the last screen. That is because in order to run the generated application 'out-of-the-box', I prepared one file that relies on the prepared demo values. Later on I will show you how to modify this particular file, so that program also compiles with your own model and settings.

1. Import the following project from `<easymda-dir>/samples` into Eclipse :  
`flca.mda.test.model.reactive`
2. `flca.mda.test.model.reactive` is a small example project how to show how a model in easyMda may look like. It is just a plain old simple Java project. It only contains some classes, that describe the model for the reactive crud application that we want to generate.
3. Select the in the **package explorer**, all the packages from the demo project, then right click and select the 'generate-code' tag at the bottom

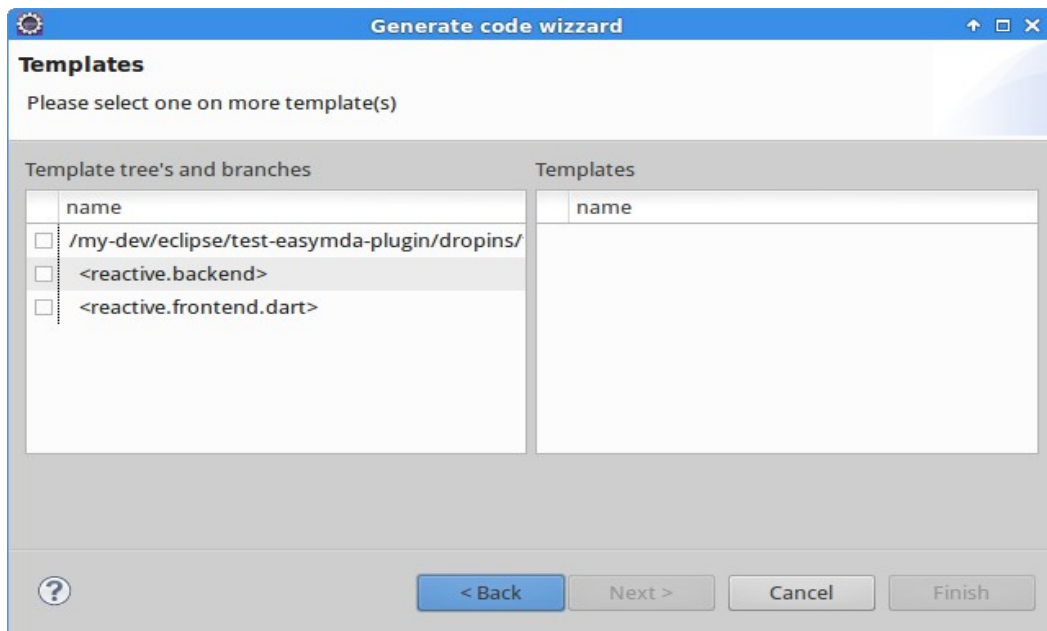


The following easyMda wizard appears:



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.

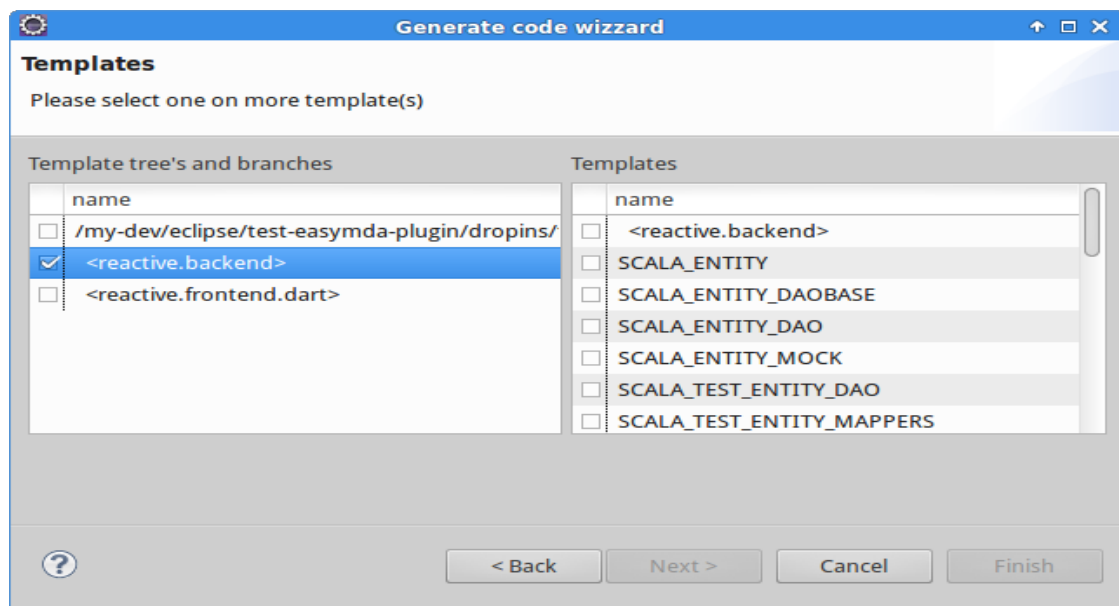
Hit [Next>] this will show the second page:



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

**<reactive.backend>**

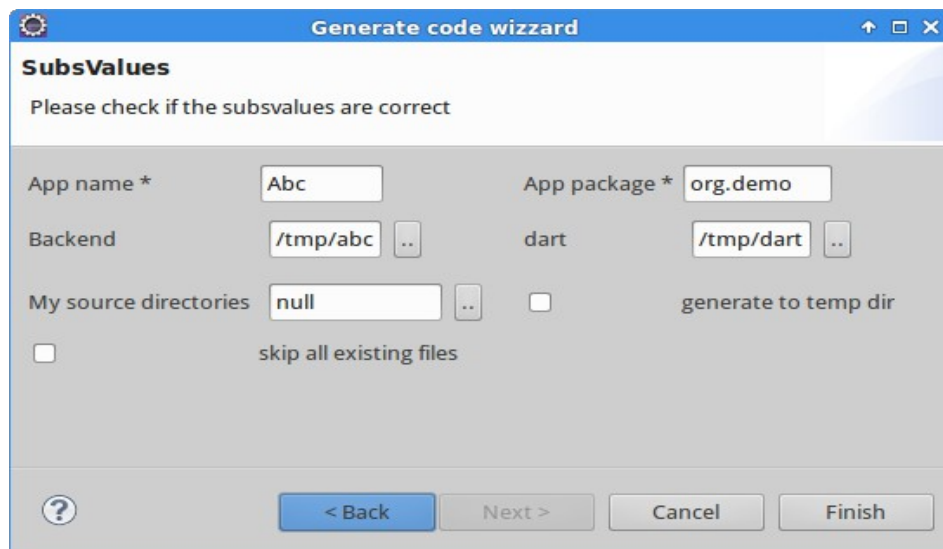
Select the `<reactive.backend>` of this template-branch. Now the right-hand side listbox will be populated with all templates that are applicable with the current selected packages and class files:



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

[ ] `<reactive.backend>` checkbox. (at the top)

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



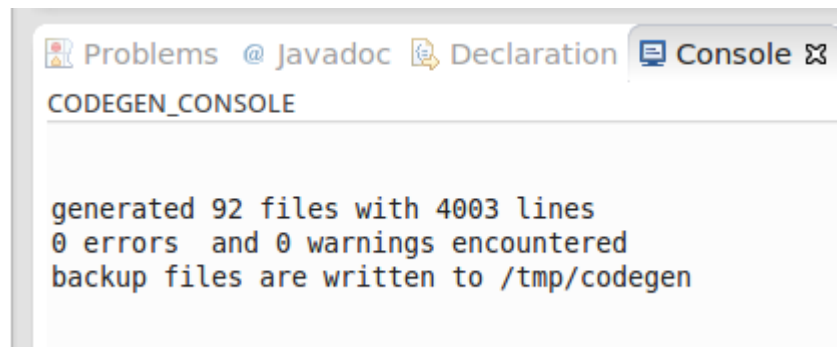
This page may vary depending on choices you made on the pages before, and also if you run the easyMda plug-in in this model-project for the first time. If you run it for the first time, some values will be empty and should be filled by you. The next time this value will be used.

For this demo I already prepared default values, so you can simply hit [Finish] button.

NOTE !

Please leave the suggested values as is for now, because you may run into problems when running the generated application. That is because for the sake of generating an out-of-the-box app, that you can run (almost) immediately without any modifications, I prepared some files that rely on the prepared values from the screen above!

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



The screenshot shows the Eclipse IDE's console window. The title bar includes tabs for 'Problems', '@ Javadoc', 'Declaration', and 'Console'. The console itself is titled 'CODEGEN\_CONSOLE' and displays the following text: 'generated 92 files with 4003 lines', '0 errors and 0 warnings encountered', and 'backup files are written to /tmp/codegen'.

Note: you may find more detailed log info, in the (logback) logfile under `/tmp/log/easymda.log`

Congratulations, you just generated the Scala backend code. Because I filled in `"/tmp/abc"` in Gui before, see:



Repeat these steps once. Again select all the packages in the package explores, from

the demo model project, but in the second screen, now select `<reactive.frontend.dart>`. Leave the values in the last screen as is, for now and hit [Finish].

This will generate all necessary code for a Google Dart project under `/tmp/dart`.

In the following chapters we will see how to build and run these generated projects.

Note:

See the EasyMda User's guide for an overall description, and the EasyMda developers guide for more detail background information.

## 4 Compile & Run the Scala backend

The Scala code is generated under `"/tmp/abc"`. The generated code is **not** restricted to 'source' code, it can be anything including project and/or classfiles, build files etc. In this case a Scala build tool file: **build.sbt** is generated. This file can be used to compile and run the generated code, and with the right sbt plugin we can also generate the necessary .project and .classpath files so that we can import this Scala project into the eclipse-based Scala IDE.

Please consult the SBT documentation for detail info. For sbt newbies here are the most import commands:

- First download and install SBT from the SBT website (<http://www.scala-sbt.org/>)
- Test the installation with: **sbt**  
This should bring the sbt console. Indicated with an `">"` prompt. You may quit the console with quit or ctrl-d
- To build this Scala project, goto the root folder that contains the build.sbt file and enter the command: `$ sbt compile`
- When no errors occur, you can run this application with `$ sbt run`  
you should see something similar like:

```
07/31 13:41:12 INFO [lt-dispatcher-2] a.e.s.Slf4jLogger - Slf4jLogger started ot
07/31 13:41:12 INFO [run-main] r.ReactiveWebApp - *** running in DEV mode ****
mode = DEV getting values from: mode.DEV.datasource
Running test against h2
mode.DEV.datasource.tables.drop
mode.DEV.datasource.tables.create
07/31 13:41:13 WARN [run-main] c.j.b.BoneCPConfig - Max Connections < 1. Setting to 20
07/31 13:41:13 WARN [run-main] c.j.b.BoneCPConfig - JDBC username was not set in config!
creating some testdata ...
Please goto: http://localhost:8000/web/index.html
07/31 13:41:17 INFO [lt-dispatcher-3] s.c.s.HttpListener - Bound to /0.0.0.0:8000
```

You can access via localhost:8000, for example localhost:8000/findTsta

Note: This backend is a standalone http(s) server, that is **not** deployed inside an application server like Tomcat or Jetty<sup>2</sup>.

---

<sup>2</sup> It is definitely possible to configure Spray so that it can run inside an application server, but not necessarily so.

If you want to examine the Scala source code, you can use any text editor, but you may want to download and install the Scala IDE (see <http://scala-ide.org/>). This is an Eclipse-based editor, hence to import a project you need a .project and .classpath file. These files can be generated with sbt if you have installed the sbt eclipse plugin. This plugin can be downloaded from: <https://github.com/typesafehub/sbteclipse>

Note

Because the generated Scala code, is based on Scala 2.10.4, you should use or download the Ide that correspond with this Scala version!

The generated source code, not only contains 'regular' code, but also a number of junit tests, in particular tests to check the persistence code. There is also a junit test, to generate json files that can be used in Google Dart to test the frontend in a completely standalone fashion. This junit test is called <AppName>JsonMocks.scala. If you run this junit test, it will generate a number of json files under "/tmp".

See the next paragraph how to use these json files within Dart.

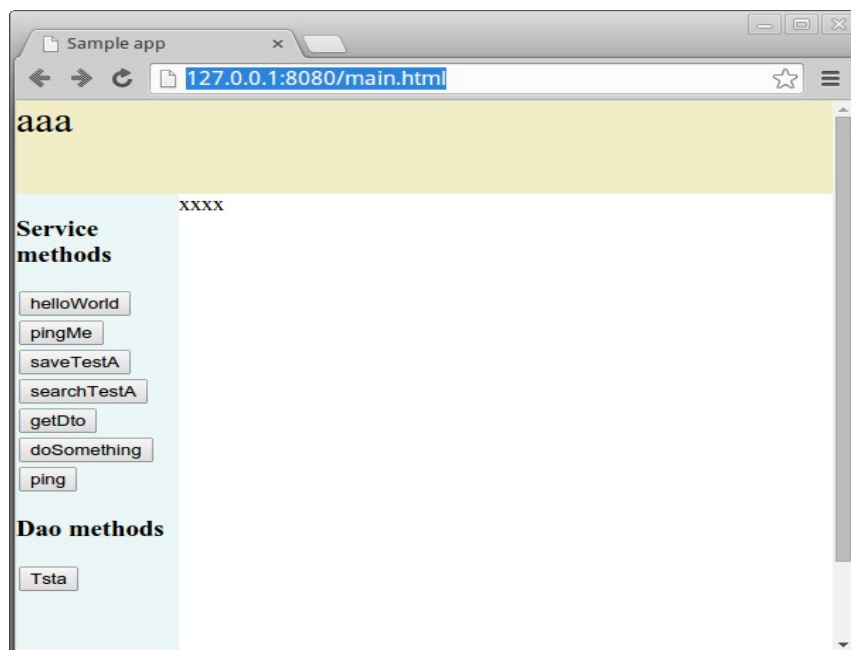
## 5 Compile & Run the Dart frontend

To run the Dart code, you obviously first have to install the latest Google Dart editor, from the website: <https://www.dartlang.org/>

If you followed the steps described in a previous chapter, then the Google Dart is generated under `/tmp/dart`. To open this Dart 'project', you simple open this folder (File → Open existing folder) (that contains the Dart the specific file: `pubspec.yaml`) You may have to execute a: Tools → pub get or Tools → pub upgrade and Tools → Reanalyze sources commands. To run the generated app:

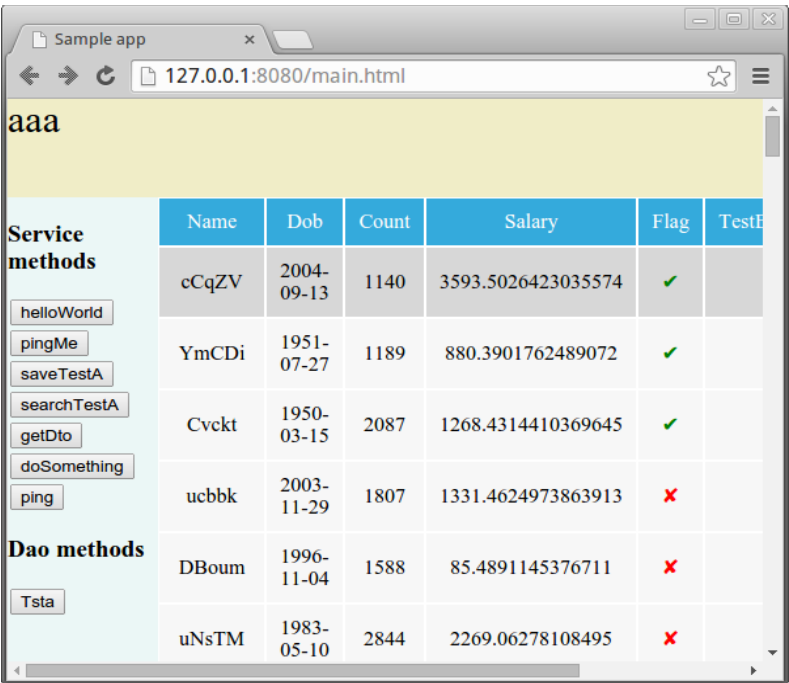
- select the file: `web/main.html`
- right click
- run in Dartium

This should pop-up a screen like this:



Click on [Tsta] to see the crud in action:





and clicking on one

the rows, shows:

The screenshot shows the same web browser window as before, but the table is replaced by a form. The form has a yellow header with the text "aaa". Below the header, there are two sections: "Service methods" and "Dao methods". The "Service methods" section contains buttons for "helloWorld", "pingMe", "saveTestA", "searchTestA", "getDto", "doSomething", and "ping". The "Dao methods" section contains a button for "Tsta". The form fields are as follows:

- id: 100
- name: LEWKI
- dob: 05/16/2014
- count: 4830
- salary: Please fill ..
- flag: ☒
- testEnum: testLiteralAnno
- testLiteralAnno: HkXxs
- Submit button

## Note

You may wonder how it is possible that the program runs, even if you did **not** start the Scala backend! That is because by default this Dart code will run in so-called standalone mode. In this mode, rather than calling an accessible Json/Rest url, it simply fetches the result Json data from a json file! These files are located under `web/resources/json`. In chapter 7 I will explain to alter this behavior.

## 6 Notes on the Scala backend

I designed the generated Scala backend, based on on a number of Scala Activator projects (see: <https://typesafe.com/activator>) in particular the projects that feature Spray, Slick and Akka (or a combination of these). Please consult the corresponding documentation and activator projects for more background information. In this chapter I want to explore the specific generated classes and its role.

But first I want to give a general overview on how and why the persistence part is generated as such.

I wanted to achieve four goals with the generated Slick code:

1. It should be as easy as with JPA to persist and retrieve a complete object graph, so it should support powerful yet easy to use full Object Relational Mapping.
2. It should be possible for any client to decide what exactly should be retrieved while obtaining an object graph. With JPA the behavior what exactly is retrieved is a very tricky process. You can annotate a relation (ex: @OneToOne @OneToMany) with a eager versus lazy. If a relation is annotated with eager, all client will always get this relation also if they don't need it. Hence it is advised to use lazy by default, but then you have to be careful to read this relation when needed and don't access it in any (tricky) way if not needed. This behavior is often the root cause of many difficult problems in Hibernate based applications.
3. Because we want to generate a reactive backend that may serve thousands of current online users, we want to be as stateless as possible. Hence we don't want objects similar to Hibernate proxies etc.
4. All last but not least, all of the above should be done in an efficient way, meaning with a small number of Sql statements.

The first goal: full ORM, is achieved by 'simply' generating all the necessary source code. So it is possible to persist an object graph with a command like, respectively retrieve the complete object graph with:

```
tstaDao.saveTsta(TstaMock.makeRandom)
val tsta = tstaDao.retrieve(123L)
```

This has the big advantage that one can see what is going on, by examining the code

inside TstaDaoBase. There is of course still some black box code inside the Slick library, but this is much less, compared to the huge black boxes when using JPA.

The second goal, that any client can specify exactly what it want to retrieve, is achieved by the 'fd' property. This fd property corresponds with FetchDepth a case class that tells exactly how much of the object graph should be retrieved. Such a case class look like this:

```
case class FdTsta(tstcs:Option[FdTstc]=Some(FdTstc()),  
                 b:Option[FdTstb]=Some(FdTstb())) extends FetchDepth
```

The signature for retrieving a Tsta object is:

```
def retrieveTsta(id:Long, fd:Option[FdTsta]=Some(FdTsta())) ...
```

This means that a command like: `tstaDao.retrieve(123L)` will retrieve the complete object graph, as if all relations were annotated with **eager**. That is because the default fd parameter is `Some(FdTsta())` and the all default parameters of the `FdTsta` constructors are `Some`'s.

If however you only need some simple Tsta property, and you not interested in any (deeply) nested object you can use a command like this:

```
tstaDao.retrieve(123L, Some(FdTsta(tstcs=None, b=None)))
```

The third goal is to be a stateless backend and don't waste any resources (memory) on proxy classes, session context etc, as for example Hibernate does. This is achieved by ... well not maintaining any additional context, proxies or whatever.

And finally the last goal is to achieve the goals above, in particular the previous goal, in an efficient and fast way. That means that in particular the number of Sql statements should be minimized, because it is mostly inefficient Sql statements that make performance suffer. I hope to achieve this goal by the chosen algorithm and making use of a combination of the fd property ohc property.

### ***How is data retrieved.***

First the algorithm. There are different ways possible to populate an object graph from several Sql tables using Sql queries. One easy to generate and elegant way, is to query each table from the top model, and for each relation, recursively loop through all instances until the last (deeply) nested model. Again, this could result in elegant code

that due to its recursive nature, fits very well in Scala. Unfortunately this could potentially lead to a (huge) number of SQL statements in order to populate the entire model.

On the other hand it is possible to generate a (possibly huge) SQL join query that fetches all data with one statement. After that it is up to the program to map each result set row into the corresponding objects. This is the approach most JPA implementations try to execute. If it works out, then indeed 1 SQL statement is sufficient. Unfortunately, for more complex situations, it happens regularly that the JPA implementation is not able to generate 1 SQL statement, and you are still facing the 1/n query problem. There is another issue with this approach, the result set coming from a complex SQL join query, may very well contain a lot of waste because all columns from the top model are copied over and over again for all nested 1/n relations. And one last issue (but that's me to blame), is that this approach is very complex to generate into solid source code.

The two observations from above directed me into another direction, that is something between very many individual SQL statements and one huge SQL join statement. Instead this cartridge generates one SQL statement for each (nested) class. So far the demo class `TstA` this means that in total 5 queries will be executed (if the entire graph needs to be retrieved). The difference with the first approach (the individual queries) is that each table is queried only once, with all the necessary where clauses. Once all tables are queried the program must map these results accordingly.

But isn't this approach potentially slower than the one SQL join query approach? I don't think so. Rather than one huge complex SQL join query, this ends up in max N (where N is the total number of objects involved) very simple SQL queries like `"select <props> from <table> where <clauses>"` that all can run in parallel.<sup>3</sup> And (again) this has the advantage that (much) less data needs to be transported over the wire.

### ***The fd and ohc property.***

Retrieving data is probably the most important factor that impacts the overall performance, but updating data may also occur frequently and hence impact the performance. Especially to reduce the number of SQL statements during updates, the `fd` and `ohc` properties are introduced.

Each generated entity (a model class that implements `IEntityType`) contains these two

---

<sup>3</sup> Note: In this version these table queries do not run in parallel yet, I will improve that in the next version.

val's:

```
val fd:Option[FdTsta]=None
val ohc:List[Long]=List(0)
```

Where `fd` is a shortcut for `FetchDepth` and `ohc` is a shortcut `Original-HashCode`. I already explained before the reason I introduced the `fd` property. It gives the client the possibility to specify exactly what needs to be fetched. But it also plays a role to minimize the number of required `Sql` statements when updating an object graph. The main role however in this respect is for the `ohc` property. Every time a record is retrieved from the database, in order to populate a corresponding object, the hashcode of `XxxRow` case-class is calculated. The value of this hashcode is saved in the `ohc` property and returned (as a `val`) to the client. When updating an object graph, this `ohc` value is first compared with the new actual value, and only when these values differ, a `Sql` update query will be performed. This is all fine for the simple properties of the object itself, but what about the nested 1/n relations? How do you know if the client added, removed and/or updated a nested relation? To know that, the `ohc` property is not a single value but instead a list see: `ohc:List[Long]=List(0)`

The first value is old hashcode of the object itself, and the following correspond with the 1/n relations inside this object. Its value is calculated by the generated `def` that looks like:

```
def ohcTstcs:Long = this.tstcs.foldLeft(0L)((acc,item) =>
  acc + (if (item.id.isDefined) 31 * item.id.get else 0))
```

Hence it is a hashcode that is calculated from the primary keys of the nested objects.

Finally we also need the original `fd` value, because otherwise we might delete all nested relations because they were not retrieved in the first place!

Well this may all look very complex (and it is), but imagine a typical use-case where (gui) client retrieved a large object graph. For example a customer with all its orders, addresses, contact information etc etc. If only item is changed, say the mobile number, then executing this simple command: `customerDao.saveCustomer(customer)` eventually only needs to perform one simple update query!

## Generated classes

If you examine the classes in the `model-project` (`flca.mda.test.model.reactive`), you can see that the two most important classes, are the ones that implement `IEntityType` (for example `Tsta`, `Tstb` etc and `IServiceType` (`TstService`). There are classes like `TstDto` to en

TstEnum, but these there to complement the two types mentioned before. I will explain these more in detail.

### ***Classes that implement IEntityType.***

These are classes that can be persisted. From the webapp cartridge, I borrowed the JPA annotation's although the persistence mechanisms itself is definitely **not** based on a JPA implementation (like Hibernate or TopLink), but on the Slick framework instead. With the JPA (look alike) annotation's like @mda.annotation.OneToOne or @mda.annotation.Table one can model the classes in the same manner as in the webapp cartridge, and the effect will be identical, that is that a class instance plus its 1/n and/or n/1 relations will be persisted eventually via SQL commands in Jdbc compatible database(s), using radically different techniques compared to JPA you are used to.

For each model entity (ex <pck>.Tsta) a number of Scala classes are generated that make use of Slick to persist the data, namely:

- <pck>.Tsta.scala

This case class corresponds to the model class, but this file also contains one extra case class: FdTsta. The 'Fd' prefix stands FetchDepth and the Tsta case class contains a **val** property fd:Option[FdTsta]=None. In addition Tsta contains another cryptic val property: val ohc:List[Long]=List(0)

The purpose of these two val properties fd and ohc will be explained more in detailed below.

Simple properties are generated as **var** properties, unless you specifically annotate (inside you model class), a property with @mda.annotation.Val <sup>4</sup>

OneToOne properties are generated as two var's, one property that corresponds with the target Class and one property that corresponds with the primary key of the target class. For example: var b : Option[Tstb]=None and var \_bId :

Option[Long]=None

The latter is important if you asked to not to retrieve the Tstb instance.

OneToMany properties are generated as Set's for example:

---

4 This decision to make var properties by default, is of course very questionable! All Scala books teach you to use val's by default and only var's if you really have to, and rightly so. Nevertheless I opted for var properties in this case because in my experience for these type of 'data' classes you often to manipulate the individual properties.

```
var tstcs : Set[Tstc] = Set()
```

- `<pck>.dao.TstaDao`

This is an almost empty class that extends `TstDaoBase` (see below)

The idea is that inside `TstaDao` you can put your own (special) queries, apart the generated ones.

- `<pck>.dao.TstaDaoBase`

This is workhorse responsible for creating, reading, updating and deleting instances of `Tsta` objects. It contains one class classes: `TstaRow` and one trait `TstaDaoBase`.

The `TstaRow` case class is the one actually corresponds with a Sql table row (hence the name).

The `TstaDaoBase` trait contains the class `TstaRows` (mind the last 's' ) that extends the Slick Table class. And a number of method's (def's) depending of the nested object's inside this class.

- `<pck>.test.dao.TestTstaDao`

This is a junit test to test the generated Slick crud code.

- `<pck>.mock.TstaMock`

This is a helper class to generate instance(s) of `Tsta`, in order to test insert functionality.

For the middleware (Akka) the following class is generated:

- `<pck>.actor.TstaDaoSrvActor`

This is Akka actor that will execute the Dao services described above

For the request handling via Spray the following classes are generated:

- `<pck>.route.TstaDaoRoute`

This Spray route is responsible for creating and executing the Akka actor from above.

### ***Classes that implement IServiceType***

Compared with the code being generated for entities, this is much less code and more straightforward. There is no Slick code generated.

The following examples are based on the model class `<pck>.TstService`

For the middleware (Akka) the following classes are generated:



- `<pck>.TstService`  
This is a plain Scala trait with the service interface.
- `<pck>.TstServiceImpl`  
This is a Scala class that contains dummy bodies for all the service methods. Unfortunately (: you have to implement all these methods yourself.
- `<pck>.mock,TstServiceMock`  
This is an empty mock implementation that can be used for testing purposes.
- `<pck>.TstServiceFact`  
This is plain old factory to return the correct service implementation. I may want to replace this with a CDI implementation, but I think it is okay for this purpose. Note you may have to manually modify this implementation to return the mock implementation instead of real implementation.
- `<pck>.TstServiceActor`  
And this is the actual Akka actor that execute the service implementation.

For the request handling via Spray the following classes are generated:

- `<pck>.TstServiceRoute`  
This Spray route is responsible for creating and executing the Akka actor from above.

### ***Classes that implement other interfaces.***

In the demo model project, there are some other classes that implement some other interfaces than the ones describe above. But these are more supporting classes. For example the model that implements IDtoType, will only generate the corresponding DTO class. These can be used in service interfaces.

### ***Generated project files***

The generated files from above all correspond directly with a class from the demo model project. In addition this cartridge also generates a number of files that are not specific to particular class but are project or application specific files.

Note: I assume that did not change any value in the generator Gui screens, so the project name is called “Abc”, and the project package is called “org.demo” These names will

appear in a number of project files.

Most of these project files can be found under: `/tmp/abc/src-gen/org/demo` see:

- **AbcConfig.scala**  
This is Scala configuration file. It is used for example to obtain the datasource. The datasources itself are stored in the companion file:  
`src/main/resources/application.conf`  
you may want to modify this file to provide other datasources. By changing the value:  
`run.mode = XXX` you can work with other datasources.
- **AbcConstants.scala**  
This is a Scala object with a few constants
- **AbcDataSource.scala**  
This Scala object maintains the actual datasource.
- **AbcDataStores.scala**  
This is class that extends a Slick profile that is used create and/or drop tables if needed,
- **AbcMain.scala**  
This is the main that startups the Akka system and prepares the database, it also calls `AbcStartup.initialize()`
- **AbcStartup.scala**  
This is class in which you can out your own initialization code. The generated code inside this class is obtained from this ini file:  
`easymda-reactive-2.0/samples/flca.mda.test.model.reactive/src/main/resources/user-config/bootstrap.ini`  
Hence it is possible to alter this generated code without changing the cartridge itself, on ad-hoc basis.
- **AbcReception.scala**  
This is Spray file were all http(s) request enter
- **AbcStaticRoute.scala**  
Another Spray with some static routes.

## 7 Notes on the Dart frontend

Like the Scala backend, the purpose of the code generator for the frontend, is to very quickly generate a lot of the common repetitive boiler plate source code. In particular the mapping from a backend object graph to the corresponding frontend object graph is otherwise cumbersome and error prone. A lot of this code frontend generator deals with the Crud part. I am aware that in practice a pure Crud application like the generated one, is very seldom. But what is very common are applications in which the crud functionality is somewhat hidden, but is nevertheless still there in the background. A very common pattern is to retrieve data from the backend, present it in whatever form in a Gui, and then update this data based after the user submitted some form or clicked some button. Hence, although you will not often see a pure Crud app like the generated one, a lot of the underlying functionality remains valid. A big difference with the generated frontend classes, is that all of the Gui code is defined as such that by default it will **not** overwrite existing code, based on the idea that you want to change the generated Gui code anyway. With this in mind let's explore the generated Dart classes.

### ***Classes that implement IEntityType.***

These classes are in subfolders under `/web` that correspond with the same package structure as in the backend. I followed the Dart naming convention, hence all filenames are in lowercase and underscore instead of camelcase. So using the model entity class `org.demo.entity.Tsta` as an example, we have:

- `web/org/demo/entity/tsta.dart`  
This is the corresponding frontend Dart class. In addition it contains code to clone and map from/to Json representation.
- `web/org/demo/entity/srv/tsta_dao_service.dart`  
This is a class that extends a (generated) service baseclass, and contains the supported Dao services, like save, retrieve and findAll
- `web/org/demo/entity/ctrl/tsta_ctrl.dart`  
This is controller that triggers the service from above and presents the right form.
- `web/org/demo/entity/view/tsta_browse.dart` & `tsta_browse.html`  
This is a Polymer component that displays the values from the findAll service in a grid.

- `web/org/demo/entity/view/tsta_form.dart` & `tsta_form.html`

This is a Polymer component that displays all the fields from the selected object.

These fields itself are also Polymer components defined in `/web/pub/form_fields`

#### Note

The latter four gui files, are generated only if the corresponding model class is annotated with `@Gui`.

### ***Classes that implement IServiceType.***

Using the model service class `org.demo.srv.TstService` as an example, we have:

- `web/org/demo/srv/tst_service.dart`

This is class that extends the (generated) base service and contains the call to corresponding backend service methods.

(Note this version generates a wrong implementation for service methods with more than one parameter, because the `getData()` does not support this. I will fix this in a next version)

- `web/org/demo/srv/tst_service_impl.dart`

TODO

### ***Generated project files***

Like the Scala backend, the Dart generator also generates a number of project related files. A number of these files are regular files that may provide a good starting point for any project, and there are some files that are only generated for the sake to create from scratch a fully functional working Crud demo. You can probably dispose these files in your own projects.

First the regular project files, that may be a good starting point:

Note: I assume that did not change any value in the generator Gui screens, so the project name is called "Abc". This name will appear in a number of project files.

- `pubspec.yaml`

Every Dart project needs this file.

- `web/abc_constants.dart`

This is an abstract class with a lot of constants that are used in several files.

- `web/abc_library.dart`

To minimize import statements in other files, it is always good to have such a library file.

- `web/abc_service_base.dart`

This is an abstract base class that contains the actual code to execute the Json/Rest to the backend.

How this is done, is explained in the “executing Json/Rest calls” section below.

Generated files for the sake of this demo are:

- `web/main.html`
- All files under `web/view/`

These are files that you probably want to replace.

### ***Generated pub files***

And finally the generator also created a (large) number of files under `we/pub`.

One can argue that files these are not generated at all. In a sense that is true because these are generated via Jet engine files, but instead simply copied from the zip file: `dart_project.zip` that is located in `<eclipse>/dropins/easymda-xxx/cartridges` folder.

I deliberately put these files under the “/pub” subfolder these files may be replaced in a future version by a pub library that provides similar (and more) functionality. The files under `pub/form_fields` are Polymer components that provide functionality like formatting and validation.

### ***Executing Json/Rest calls.***

If you examine `abc_service_base.dart` file, you can see that can execute a Json/Rest call in three different modes, that correspond with the constants define in `abc_constants.dart` :

```
static const RUN_MODE_PROD = 0; //use the backend url
static const RUN_MODE_LOCAL = 1; //use generated json files under resources/json
static const RUN_MODE_TEST = 2; //use a proxy server to overcome cors headers
```

The generated current value is `RUN_MODEL_LOCAL`, that means the service base is not actually calling a Json/Rest via an url, but instead get the response via a json file. It tries to obtain this json file from this location: `web/resources/json/<servicename>.json`

While developing the frontend this mode is particular handy because you can run the frontend in completely standalone fashion, you don't need a backend at all!

Ultimately you want to deploy the Dart app somewhere and use the url of the 'real' backend for all the Json/Rest calls. This is done when `RUN_MODUS = RUN_MODE_PROD = 0`.

The problem here is that you quickly run into 'INVALID CORS HEADERS' issues. Apparently in the current Dartium version, you run into this problem more frequently, even if the backend can handle Cors headers.

To overcome the problem above, there is third modus: `RUN_MODE_TEST`. In this modus a proxy server is used so that execute call to the Scala backend under construction, but don't run into Cors headers issues.

There are many proxy servers out there. I personally like the Corsa proxyserver, and this generator, generates code for this particular proxy server.

You can download this Python based proxy server here: <https://github.com/olt/corsa>

And start it like this:

```
corsa --allow-proxy ALL --allow-origin ALL
```

In this mode the json/rest url is something like:

```
http://localhost:<PROXY_PORT>/proxy/http://localhost:<HOST_PORT>/findTsta
```

## 8 Create your own Reactive model project

This demo so far, was based on a demo model project with not very useful abstract classes like “Tsta” Tstb” etc, only to proof the technical functionality. If you want to create your own model with more useful classes like Customer, Order, Movie whatever, than probably the easiest way, is start from the demo model project from above, and simply rename and refactor the technical classes accordingly. Besides it is a fast path, you can also peek in the provided classes how to define your own models. Luckily there are not many rules to obey in order to create valid model classes.

I will explain these rules below, but first you to need to understand what is the purpose of this model-project. This Reactive cartridge, is all about generating all of the plumbing source code (for both the front- and backend) that deals with persisting objects into Jdbc compatible databases and Json/Rest based services. So those are the two main classes that we want to model: Entities and Services. Because a Json/Rest service is very limited if it would only support simple parameters, we can also model data transfer object (Dto's) and because these are heavily used as well: Enum objects.

### ***Implement the correct interface***

If you examine the classes inside the demo model project, you can see that all of the classes are plain Java classes, (or an interface in the case of a `IServiceType`), that implement a specific interface.

**Note:** There is one exception, and that is the class: `flca.demo.types.TstEnum`, this is just a regular enum being used in `Tsta`.

This interface is important because it tells the generator how to deal with this model class. Currently the following interfaces are supported by the reactive cartridge:

- `IEntityType`  
This will generate a class that can be persisted, plus all supporting code, including junit tests.
- `IServiceType`  
This will generate all code for a Json/Rest service.
- `IDtoType`  
This will generate the corresponding Dto class. This one is not persisted.
- `IApplicationType`

This is a strange beast. It is not used to generate some specific code, but instead it is used by the generator to 'detect' the model project. The generator scans the model project for exactly one class instance that implements this `IApplicationType`.

Depending on the interface, the body of the class is modeled differently.

### ***Entity types***

This class contains simple properties and/or relations. These properties may be primitives like (int, float) etc or standard Java properties like String, Integer, BigDecimal, Date etc. It is also possible to define a property that points to an Enum.

Because this class must be persisted into a jdbc compatible database :

- it is **not** possible to define arrays or collections!
- It is **not** possible to define a property to non standard Java object (except for an Enum class).

It is however possible to define a 1/1 relation to another entity class. This is achieved with the `@mda.annotation.OneToOne` for example:

```
@OneToOne
@JoinColumn(name="TSTA_TSTB_FK")
Tstb b;
```

And it possible to a 1/n relation with the `@OneToMany` resp `@ManyToOne`, see:

```
in owner class Tsta:
    @OneToMany(mappedBy="tsta")
    Set<Tstc> tstcs;
```

in the child class Tstc:

```
@ManyToOne()
Tsta tsta;
```

#### **Tip**

All the cartridge annotations can be found under: "mda.annotation" and below. So to find out annotations, you can use Eclipse's intellisense.

The annotations above are all so called field annotations. For `IentityType`'s there are also the following class annotations available:

- `@RestService (generateService=true)`  
For each entity a number of data access services will be generated: `findAllXxx`, `retrieveXxx`, `saveXxx` and `deleteXxx`. which can be used internally on the backed. By default, **no** corresponding json/rest services will generated for these internal services. If you want to expose these data access (dao) service, you have this `@RestService(generateService=true)` annotation.
- `@Gui`



By default **no** Google Dart screens (`xxx_browse` and `xxx_form`) will be generated for each entity. You have to use this `@Gui` annotation<sup>5</sup>, if you do want to generate these screens.

### ***!Entity methods***

Currently the Reactive cartridges, does not support any method within a `IEntityType`. In a next version I want to support methods that are annotated to define additional finder methods, something like:

```
@DaoFind
Tsta findTstaByCriteria(TstCriteriaDto aCriteria)
```

### ***!EntityType limitations***

This (first) version of the Reactive cartridge has the following limitations :

- The `@ManyToMany` annotation, that should generate the corresponding join table's, is not yet supported.
- The `IEntityType` class, may not extend another `IEntityType` class.

To fix these limitations have the highest priority after the first release.

### ***!Service types***

Only an interface may extend an `IServiceType`. The body contains any number of methods. The method return type and method arguments may be any simple Java attribute. Also arrays or collection's etc are supported. And the return and/or argument types may be entities or dto's. See `TstService` as an example.

### ***!ServiceType annotations***

TODO

### ***!ServiceType limitations***

It is possible to define methods with more than 1 parameter, but currently these are not yet handled correctly inside the service base inside Google Dart.

---

5 I may want to change the `IEntityType` interface with methods like: `boolean generateGui()` and `boolean generateRestService()`, so one is forced to set this value, rather than using these optional annotations.

## IDtoType

This class is much simpler than the IEntityType. It can be used for return type and/or argument type in service methods from above.

Because a Dto is not persisted (at least the generator does not generate any code for that), it is less restrictive to the property types. Hence any standard Java type is supported including arrays collection's etc. In addition a Dto can contain other Dto's, but rather than annotate these, you simply define the relation, for example:

```
class TestDto implements IDtoType {
    MyDto mydto;
    List<OtherDto> otherDtos; //any Collection or array is allowed
    Tsta tsta; // you may also embed entity objects
    :
}
```

## Scala Compiler errors

When you make up your own model, you most likely will encounter a Scala compiler error after you generated the new Scala backend. This is because the generated XxxStartup.scala file will not compile. This file is generated as a placeholder in which you can put afterward any initialization code you want. So it should be more or less empty. But in order to see immediately some results when accessing for example:

localhost:8080/findTsta

I put in some initialization code, to populate the Tsta and all tables with some test data. I did not put this initialization code directly inside the corresponding Jet engine file:

templates/frontend/dart/appbase/AppStartup.jet

Instead this initialization code is picked up this ini file:

flca.mda.test.model.reactive/src/main/resources/user-config/bootstrap.ini

see:

```
[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("error start H2 " + ex);
    }
}
}}
```

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

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

syntax.

### ***Dart Runtime errors***

Running your newly generated Dart application in standalone mode, will most likely generate errors, because it can find the corresponding json file under `web/resources/json`.

You can generate these files by running the (generated) Scala junit test:

`<AppName>JsonMocks.scala6`

---

<sup>6</sup> In a future version I may want to generate these json files directly by the generator.

## 9 Developer's notes on the Reactive cartridge

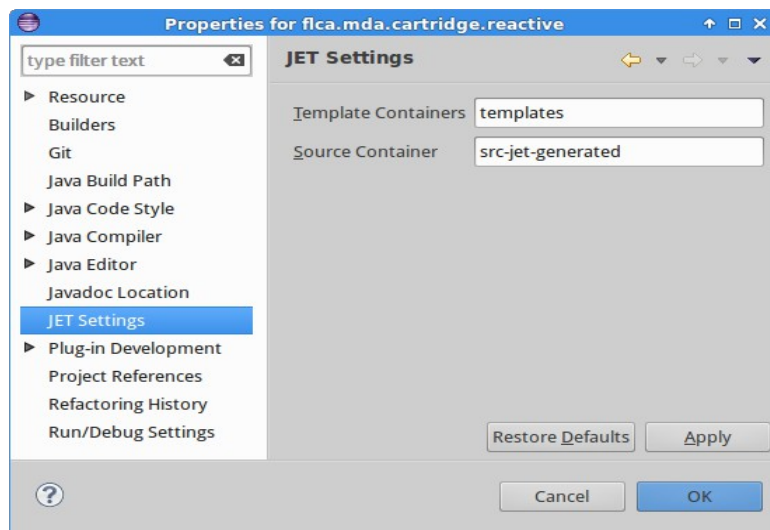
Please consult the EasyMda developer's guide, for more overall and background information on how the EasyMda generator, and how modify a cartridges or create your own cartridge.

In this chapter I zoom more into this Reactive cartridge.

The source code of the Reactive cartridge, can be found in the Eclipse project `flca.mda.cartridge.reactive`. It is standard Eclipse plugin project, that heavily uses the EMF (Eclipse Modeling Framework). In particular it relies on the Jet engine. The Jet engine is defined in the `.project` file., see:

```
<nature>org.eclipse.emf.codegen.jet.IJETNature</nature>
<nature>org.eclipse.pde.PluginNature</nature>
```

The Jet engine requires the correct setup of the JET Setting, this is setup as follows:



The `flca.mda.cartridge.reactive` project also heavily depends on the older project `flca.mda.common.api`. This common project contains many of the general api used in jet files.

## Project structure

In the `flca.mda.cartridge.reactive` project, the Java classes are organized as follows:

- Under `mda.annotation` the annotations that are specific for this cartridge, can be found. All the other are (also under `mda.annotation`) in the common project.
- Under `mda.type` is `IApplicationType` interface that is specific for this cartridge.
- Under `flca.mda.api.util` are the helper api classes that can be used by the jet files. Several of these extend classes from the common project. For example the `ScalaTypeUtils` extends the `ReactiveTypeUtils` that extends `TypeUtils`.
- And under `mda.reactive` are the classes that actually define the Reactive cartridge. Some of these I will explain more in detail:

## RegisterReactiveTemplates

This class actually defines the cartridge, because this is the class that implements the `IRegisterTemplates` interface. The generator scans the cartridge for the one (and only one) instance that implements this interface. Once the generator located this instance it then calls the `IregisterTemplates` methods to register all the templates etc. In particular it calls this method: `List<TemplatesBranch> getTemplateBranches()`.

An EasyMda cartridge can be seen as tree, that contains one or more branches and each branch contains one or more `ITemplate`'s. In this Reactive cartridge this method like:

```
public List<TemplatesBranch> getTemplateBranches() {  
    List<TemplatesBranch> result = new ArrayList<TemplatesBranch>();  
    result.add(makeBranch(sScalaTemplates, "reactive.backend", "Scala ..."));  
    result.add(makeBranch(sDartTemplates, "reactive.frontend.dart", "Frontend ..."));  
    // result.add(makeBranch(sReactiveDojoTemplates, "reactive.frontend.dojo", "Dojo.."));  
    return result;  
}
```

Hence two branches are defined, the Scala based backend and Google Dart frontend. Note that most code for a third branch, a frontend based on Dojo, is also ready, but I left it out of this release. It will be published soon with a next update.

You may recall that all available branches appear in the second screen, after hitting "Generate code ...".

Each branch in its turn, contains one or more `ITemplate` instances. These are registered in `ReactiveScalaTemplates` and similar classes. An `ITemplate` defines which Jet engine is used, what will be the target Class, package and file name. It has a name, which is defined in

the Tid (shortcut for Template Id) class. This Tid is very frequently used in the Jet files, to access other instances, via the TemplateUtils and ImportUtils <sup>7</sup>api. For example:

```
<% impu.addTemplateImport(Tid.SCALA_SLICK_PROFILE.name());%> or
<% String mock = tplu.getClassName(Tid.SCALA_ENTITY MOCK.name());%>
```

The Itemplate method: `Class[] getApplyToClasses()` defines if the class(es) that is selected, is applicable for this template. Note: maybe the name `getApplyToInterfaces()` would have been more appropriate, because the generator checks if the selected class implements on the interfaces returned by this method.

Note: You may wonder, what's the point of setting the empty value:

`r.getApplyToClasses(new Class[] {});` inside the `makeDummy()` method in `ReactiveScalaTemplates`. This looks pointless because apparently the generator will never use this template to generate any code. Well that is true, but nevertheless this template is still registered, and can therefore be looked up via the TemplateUtils and ImportUtils api from above. Which is also used frequently for these 'dummy' templates, because the corresponding classes are indeed 'generated', not via the Jet engine but by simply copying all the files from the `cartridges/scala_project.zip` file to the target folder, and after that execute some very simple search/replace commands.

And there is one other use-case where an empty `getApplyToClasses()` makes sense. That is when you simply force this template to be used. The point is, how to deal with situations where depending on whatever input, you don't want to generate code or instead want to generate some other file(s). For example for an `IEntityType` class, that is not annotated with `@Gui` annotation, we don't want to generate Dart gui screens. This is achieved by simply returning "" in the Jet file like this:

```
<% if (!tu.hasAnnotation(element.getClass(), mda.annotation.Gui.class)) return ""; %>
```

On the other hand, instead of returning an empty string (or null), a Jet file can trigger the execution of other template(s), via this method in `TypeUtils` :

```
public void generate(Class<?> aClass, ITemplate aTemplate, Object ... args)8
```

For no particular reason, this latter method is hardly used, whereas returning an empty string when you don't want generate code, is used quite frequently.

---

7 To use these api, these shortcut are defined: `impu = new ImportUtils(); tplu = new TemplateUtils();`

8 Note that the return type of this `generate()` method is void. Because this method itself does not return a value, instead it will generate the target file, if everything goes well.

ITemplate also allows to (optionally) add one or more hook(s) in the form of ITemplateHooks. Via an ITemplateHooks it is possible for example to modify the generated source code before it is written to the target file. Examples of such a hook is the ReplaceImportWildcardPostprocessor class that replaces the Java import “.\*” clause for the Scala syntax “.\_”;

### ***The Reactive Jet engine files***

As we have seen before, the source- and target location of the Jet engine files are defined in .project and point respectively to the templates and src-jet-generated folder. One nice and powerful feature of the Jet engine, is that at any time you can delete all the files under

src-jet-generated and one sec later to be re-generated again! Any Jet syntax error is spotted immediately and not only run-time as with many other template engines.

A Jet engine file is similar to a jsp file, hence you (almost) have the full Java stack at your disposal. But because this Jet code is cluttered with all these “<%...%>” tags, you try keep this file as simple as possible, by delegating lot of the work helper classes.

These helper classes can be found under the package flca.mda.api.util

### ***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.

### ***Interfaces***

The Reactive cartridge (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
<code>scala.Val</code> <code>scala.Var</code>	To indicate if this property should be a <b>val</b> or <b>var</b> . Default is <b>var</b> !
<code>JavaDoc</code>	Add javadoc



	Note it is much easier however to simply put Javadoc above the property or method, this will be copied as-is
jpa.Id jpa.OneToMany jpa.ManyToOne jpa.OneToOne	These are annotations that are similar to the equivalent javax.persistence.xxx annotations. I have copied these annotations to the reactive cartridge under mda.annotation
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
Gui	Used for IEntityTypes to indicate if frontend screens should be generated.

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 reactive 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:

- OneToMany relation
- ManyToOne relation
- OneToOne relation

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

```
class Tsta {
```

```
@OneToOne ()
Tstb b;
```

```
@OneToMany()
Set<Tstc> 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 Tstc {
    @ManyToOne()
    Tsta 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 IEntity type interface. So for example relations like:

List<Integer> numbers;

are invalid<sup>9</sup>. If you need this relation, you should create a class for example like this

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

and use this class like this:

```
@OneToMany
List<MyNumber> numbers;
```

## Crud operations

In many applications you will have some standard crud operations for entity instances.

<sup>9</sup> At least for jdbc based persistence. For OO database such relations are ofter valid.

The reactive templates make it is easy to generate these standard crud operations. The internal crud operations are generated anyway. To expose these crud operations into Json/Rest services, you have to annotate the IEntityType class with `@RestService (generateService=true)`

It will generate the following crud methods:

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

Note

I a future version it will be possible to model finder method(s) as well, via a dedicated annotation like `@DaoFind`

### **JavaDoc**

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

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

## 10 Conclusion

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

Try experimenting with your 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 send 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](mailto:robin.bakkerus@gmail.com)