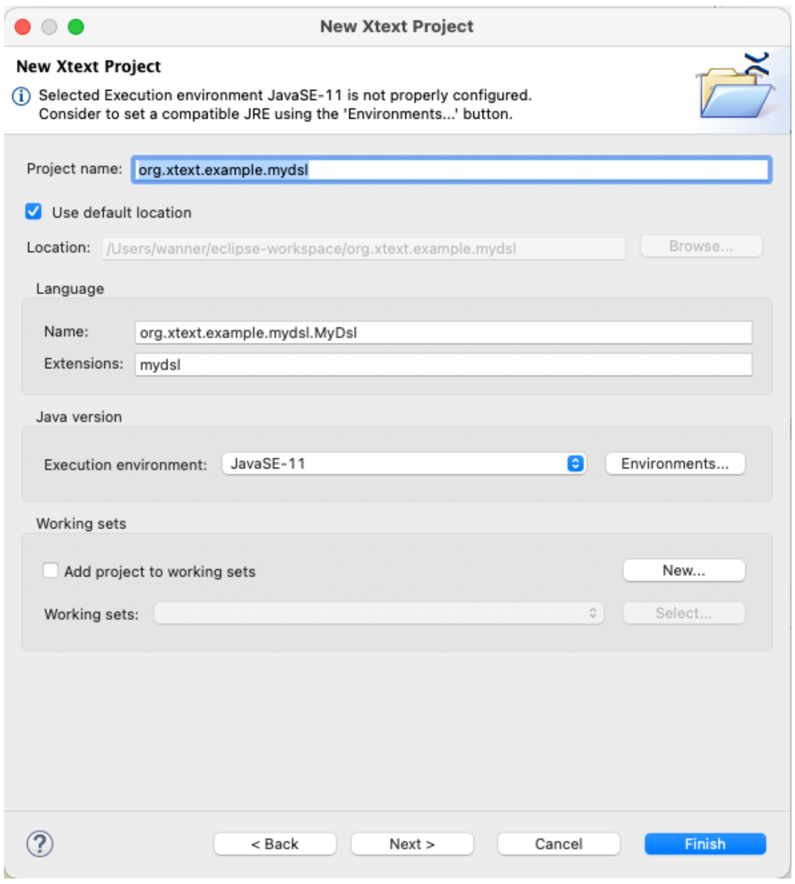
Xtext - GettingStarted

Presented below are a few steps to gain a first impression of how Xtext works. Set up your Eclipse IDE (version 4.5 or higher required) by using the plugins from the download page or install Xtext using the marketplace.

This guide is for Xtext version 2.9 and higher.

Getting started

Let us start with using the wizard to create projects which represent the example we will go through. Choose *File > New > Other...* and then *Xtext > Xtext* Project to open the wizard. Here you can enter the main project name, the name of your domain-specific language (DSL) you are about to design, and the default extension for DSL files. As a start, just leave the given values (assumed below). So do now and press *Finish*.



New Xtext Project

New Xtext Project

Selected Execution environment JavaSE-11 is not properly configured.
Consider to set a compatible JRE using the 'Environments...' button.

Project name: org.xtext.example.mydsl

[✓] Use default location

Location: /Users/wanner/eclipse-workspace/org.xtext.example.mydsl

Language

Name: org.xtext.example.mydsl.MyDsl

Extensions: mydsl

Java version

Execution environment: JavaSE-11

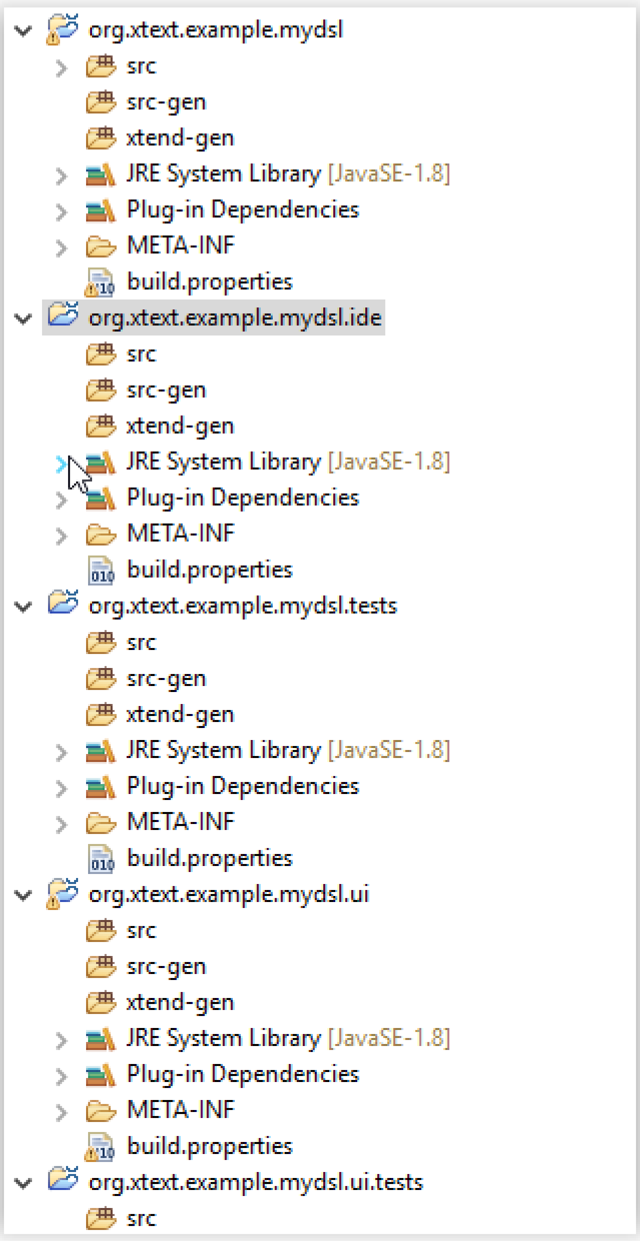
Working sets

[ ] Add project to working sets

Working sets:

Project Overview

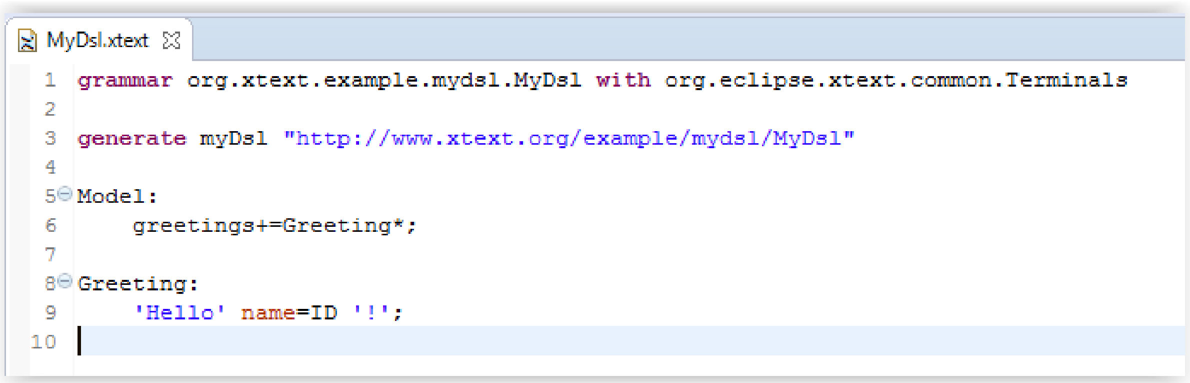
In the Package Explorer you can see five new projects. In *org.xtext.example.mydsl* you can define the grammar of your DSL, in *org.xtext.example.mydsl.test* you will be able to create tests, and into *org.xtext.example.mydsl.ui* editor classes for your DSL will be generated. It is good to be clear and unambiguous whether the code is generated or is to be manipulated by the developer. Thus, the generated code should be separated from the manual code. We follow this pattern by giving a folder *src* and a folder *src-gen* in each project. Keep in mind not to make changes in the *src-gen* folder. They would be overwritten in the next generation process.



```
org.xtext.example.mydsl
    src
    src-gen
    xtend-gen
    JRE System Library [JavaSE-1.8]
    Plug-in Dependencies
    META-INF
    build.properties
org.xtext.example.mydsl.ide
    src
    src-gen
    xtend-gen
    JRE System Library [JavaSE-1.8]
    Plug-in Dependencies
    META-INF
    build.properties
org.xtext.example.mydsl.tests
    src
    src-gen
    xtend-gen
    JRE System Library [JavaSE-1.8]
    Plug-in Dependencies
    META-INF
    build.properties
org.xtext.example.mydsl.ui
    src
    src-gen
    xtend-gen
    JRE System Library [JavaSE-1.8]
    Plug-in Dependencies
    META-INF
    build.properties
org.xtext.example.mydsl.ui.tests
    src
```

Grammar

If everything went well, the example grammar file *MyDsl.xtext* from the first project is opened with the Xtext Editor. This grammar has two purposes. First, it describes the concrete syntax of text files that can be processed by the tools of your new DSL. Second, it equates the meta level, i.e. the model which describes your model.



```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
    greetings+=Greeting*;

Greeting:
    'Hello' name=ID '!';

```

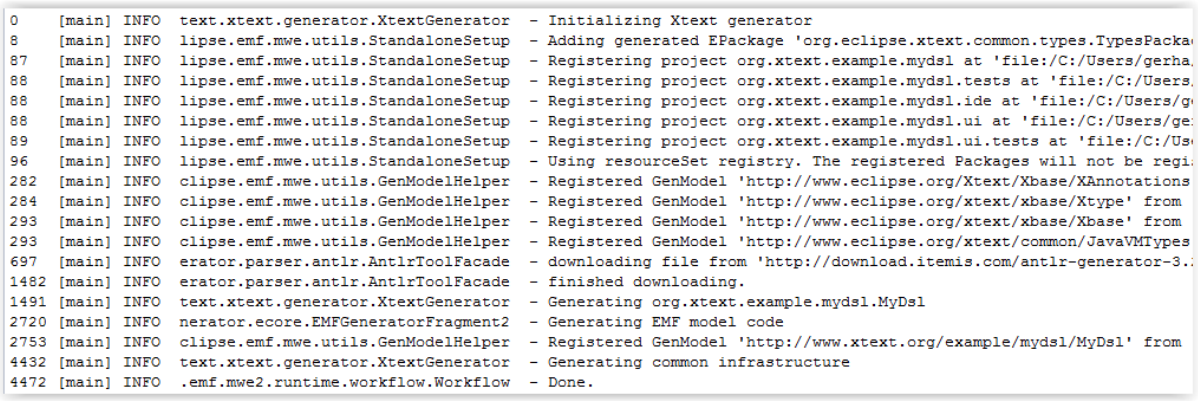
As you can see, a working example is already given. Besides the grammar's name declaration (*grammar...*) and the declaration of the further generation process (*generate...*) you find parser rules for *Model* and *Greeting*.

Please exchange the code of that getting started example with the following code:

```
grammar org.xtext.example.mydsl.MyDsl with
org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
	(imports+=Import)*
	(elements+=Type)*;

Import:
	'import' importURI=STRING;

Type:
	SimpleType | Entity;

SimpleType:
	'type' name=ID;

Entity :
	'entity' name=ID ('extends' extends=[Entity])? '{'
		properties+=Property*
	'}';

Property:
	'property' name=ID ':' type=[Type] (many?='[]')?;
```

DSL Generation

In the same directory you find *GenerateMyDsl.mwe2* which is a workflow configuration for the Modeling Workflow Engine. Right-click this file in the Package Explorer and choose *Run As → MWE2 Workflow*. You can observe the generation process so triggered in the console view. After it, files are added in all of your three DSL projects. You are almost ready to design your model that follows the DSL.



```
0    [main] INFO  text.xtext.generator.XtextGenerator  - Initializing Xtext generator
8    [main] INFO  lipse.emf.mwe.utils.StandaloneSetup  - Adding generated EPackage 'org.eclipse.xtext.common.types.TypesPacka
87   [main] INFO  lipse.emf.mwe.utils.StandaloneSetup  - Registering project org.xtext.example.mydsl at 'file:/C:/Users/gerha
88   [main] INFO  lipse.emf.mwe.utils.StandaloneSetup  - Registering project org.xtext.example.mydsl.tests at 'file:/C:/Users
88   [main] INFO  lipse.emf.mwe.utils.StandaloneSetup  - Registering project org.xtext.example.mydsl.ide at 'file:/C:/Users/g
88   [main] INFO  lipse.emf.mwe.utils.StandaloneSetup  - Registering project org.xtext.example.mydsl.ui at 'file:/C:/Users/ge
89   [main] INFO  lipse.emf.mwe.utils.StandaloneSetup  - Registering project org.xtext.example.mydsl.ui.tests at 'file:/C:/Us
96   [main] INFO  lipse.emf.mwe.utils.StandaloneSetup  - Using resourceSet registry. The registered Packages will not be regi
282  [main] INFO  clipse.emf.mwe.utils.GenModelHelper  - Registered GenModel 'http://www.eclipse.org/Xtext/Xbase/XAnnotations
284  [main] INFO  clipse.emf.mwe.utils.GenModelHelper  - Registered GenModel 'http://www.eclipse.org/xtext/xbase/Xtype' from
293  [main] INFO  clipse.emf.mwe.utils.GenModelHelper  - Registered GenModel 'http://www.eclipse.org/xtext/xbase/Xbase' from
293  [main] INFO  clipse.emf.mwe.utils.GenModelHelper  - Registered GenModel 'http://www.eclipse.org/xtext/common/JavaVMTypes
697  [main] INFO  erator.parser.antlr.AntlrToolFacade  - downloading file from 'http://download.itemis.com/antlr-generator-3.
1482 [main] INFO  erator.parser.antlr.AntlrToolFacade  - finished downloading.
1491 [main] INFO  text.xtext.generator.XtextGenerator  - Generating org.xtext.example.mydsl.MyDsl
2720 [main] INFO  nerator.ecore.EMFGeneratorFragment2  - Generating EMF model code
2753 [main] INFO  clipse.emf.mwe.utils.GenModelHelper  - Registered GenModel 'http://www.xtext.org/example/mydsl/MyDsl' from
4432 [main] INFO  text.xtext.generator.XtextGenerator  - Generating common infrastructure
4472 [main] INFO  .emf.mwe2.runtime.workflow.Workflow  - Done.
```

Try the Editor

Let's give the editor a try. If you select Run As → Eclipse Application from the project's context menu, you can create a new Eclipse Application. A new Eclipse instance will be launched and allow to test drive the editor.

Before you can create a file for the sample language, you'll have to create a sample project. Select File → New → Project... and choose a project type of your choice, e.g. Java Project, and name it Sample. Please uncheck the option "Create module-info.java file" on the second page of the dialog.

Create a new file in the src folder of the project: From the context menu of the folder choose New → File, name it Sample.mydsl hit Finish. The newly created editor will open for you language and ask you in a dialog, whether you want to add the Xtext nature to your project, which is a good idea. You can now give the editor a try, e.g. use content assist (Ctrl+Space) to insert the keyword entity and see how the input is validated immediately.

Enter the following code in this editor:

```
/*
 * This is an example model
 */
type String

entity Leaf extends Composable {
  property name: String
}

entity Composite extends Composable {
  property content: Composable[]
}

entity Composable {
}
```

If all works correctly you should see the syntax-highlighting as in the code snippet above.

Deployment

There are (at least) two kinds of developers that use Xtext to create and work with DSLs. The tool smith designs the language (and generator templates) before he delivers his work to the modeler. The "work" such a tool smith produces are Eclipse plug-ins that can be installed on an Eclipse installation of the modeler.

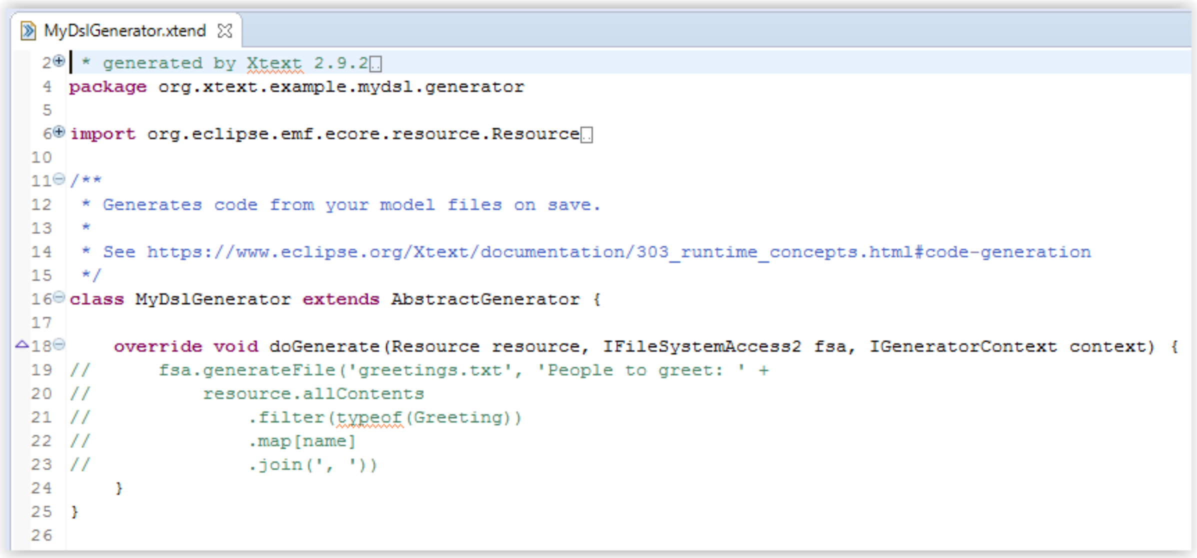
For the moment, we are both tool smith as well as modeler. Therefore, we have to deliver our work to ourself in order to model with our DSL in the next step. Choose *File* → *Export...* and *Plug-in Development > Deployable plug-ins and fragments*, select all plug-ins and choose the Eclipse directory as destination. Press *Finish* and restart Eclipse. You are now acting as modeler.

And, tool smith, please remember to deploy your plug-ins again each time you change the grammar.

Writing a Code Generator with Xtend

As soon as you generate the Xtext artifacts for a grammar, a code generator stub will be put into the runtime project of your language. Let's dive into Xtend and see how you can integrate your own code generator with Eclipse.

Close the second instance of Eclipse and locate the file MyDslGenerator.xtend in the package org.xtext.example.mydsl.generator (in folder src). This Xtend class is used to generate code for your models in the standalone scenario and in the interactive Eclipse environment.



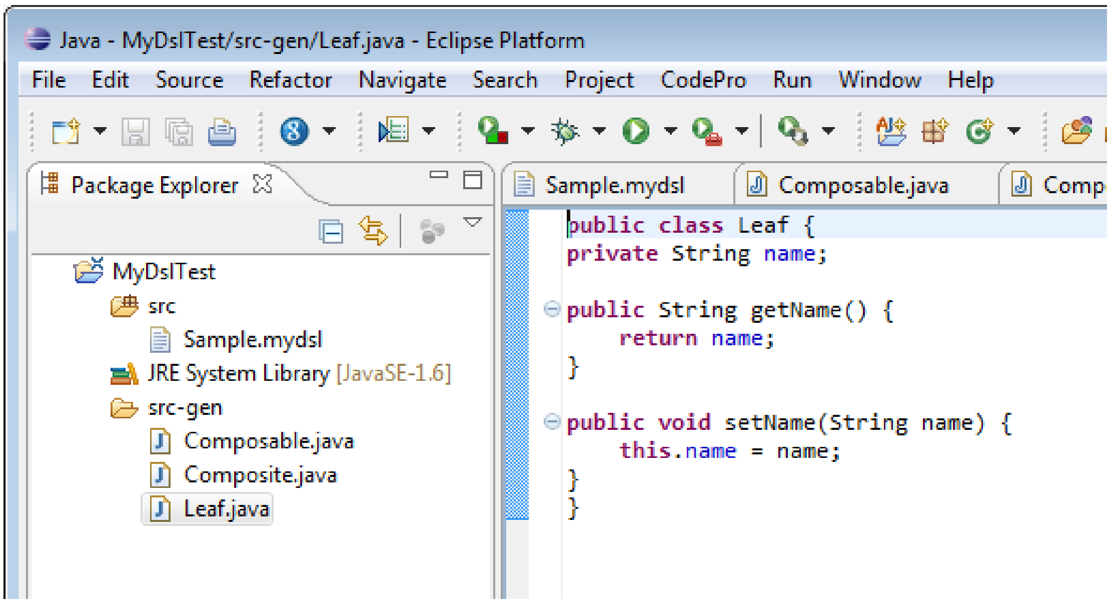
```
 * generated by Xtext 2.9.2
package org.xtext.example.mydsl.generator

import org.eclipse.emf.ecore.resource.Resource

/**
 * Generates code from your model files on save.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#code-generation
 */
class MyDslGenerator extends AbstractGenerator {

	override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
//		fsa.generateFile('greetings.txt', 'People to greet: ' +
//			resource.allContents
//				.filter(typeof(Greeting))
//				.map[name]
//				.join(', '))
	}
}
```

Replace the example code with the following code:

```
/*
 * generated by Xtext
 */
package org.xtext.example.mydsl.generator

import org.eclipse.emf.ecore.resource.Resource
import org.xtext.example.mydsl.myDsl.Entity
import org.xtext.example.mydsl.myDsl.Property
import org.eclipse.xtext.naming.IQualifiedNameProvider
import com.google.inject.Inject
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
```

```
/**
 * Generates code from your model files on save.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#code-
generation
 */
class MyDslGenerator extends AbstractGenerator {

	@Inject extension IQualifiedNameProvider nameProvider

	override doGenerate(Resource resource, IFileSystemAccess2 fsa,
IGeneratorContext c) {
		for (e : resource.allContents.toIterable.filter(typeof(Entity))) {

			fsa.generateFile(e.fullyQualifiedName.toString.replace(".",
"/") + ".java", e.compile)
		}
	}

	def compile(Entity e) '''
		public class «e.name» {
		«FOR p:e.properties»
		«p.compile»
		«ENDFOR»
		}
	'''

	def compile(Property p) '''
 private «p.type.name» «p.name»;

 public «p.type.name» get«p.name.toFirstUpper()»() {
     return «p.name»;
 }

 public void set«p.name.toFirstUpper()»(«p.type.name» «p.name»)
{
     this.«p.name» = «p.name»;
 }
 '''

}
```

Now you can give it a try! Launch a new Eclipse Application (Run As → Eclipse Application on the Xtext project) and create a *.mydsl file in a Java Project (or reuse the Sample.mydsl you created already before). Now simply create a new folder src-gen in that project and see how the compiler will pick up your sample Entities and generate Java code for them. To start the generation you probably have to modify and save the *.mydsl file again.


Java - MyDslTest/src-gen/Leaf.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project CodePro Run Window Help
Package Explorer
MyDslTest
src
Sample.mydsl
JRE System Library [JavaSE-1.6]
src-gen
Composable.java
Composite.java
Leaf.java
Sample.mydsl
Composable.java
Comp

```
public class Leaf {
private String name;

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}
```

If you want to play around with Xtend, you can try to use the Xtend tutorial which can be materialized into your workspace. Simply choose New → Example → Xtend Introductory Examples and have a look at Xtend's features. As a small exercise, you could implement support for the many attribute of a Feature or enforce naming conventions, e.g. field names should start with an underscore.