
*for beginners*



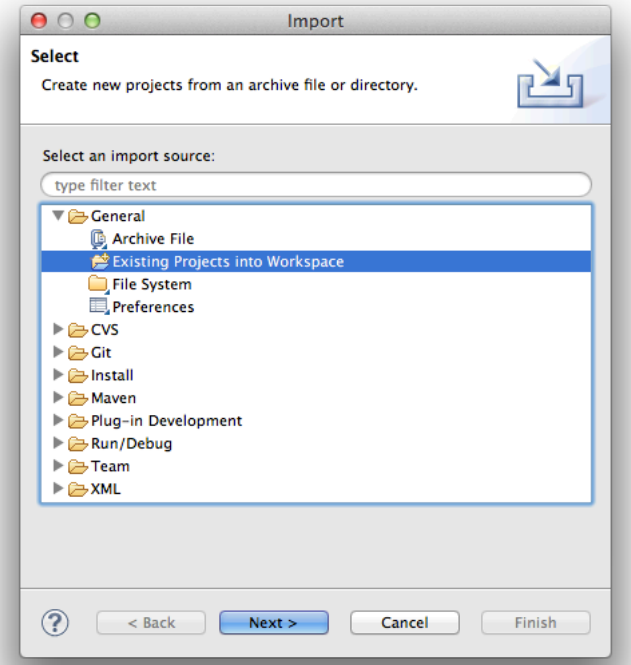
itemis

# Tutorial Preparations

### Setting up Eclipse and your Workspace

1. Copy the Eclipse application from the provided USB thumb drive to your hard drive. We provide Eclipse Versions for Windows, OS X and Linux. Launch Eclipse and select a folder on your hard drive for your workspace. This is the folder in which the files that you work with will be stored.

2. In Eclipse, close the welcome screen (if it opens), go to the Java Perspective and choose “Import...” from the context menu. A wizard pops up, in which you select “General” -> “Existing Projects into Workspace”



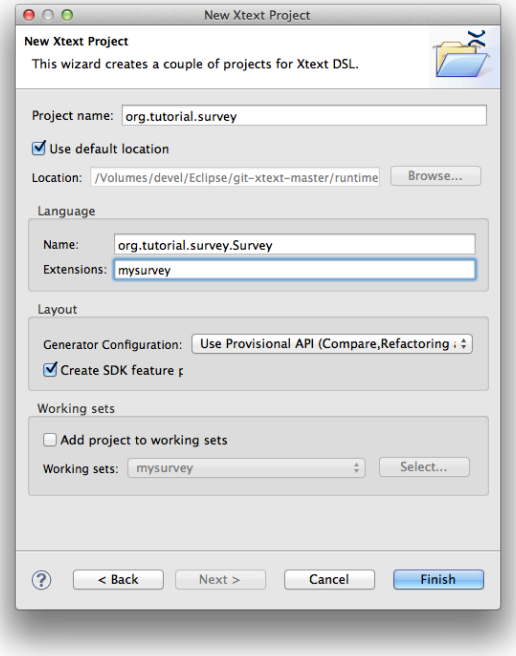

3. On the next page of the wizard, select the archive file “tutorial.zip” from the USB stick. This will create several new projects in your workspace which are in fact the solution to this tutorial. You may use it for inspiration in case you get stuck.

# Exercises

### Exercise 0: Create a New Xtext Project

Learning Objective: Familiarize yourself with the anatomy of an Xtext Project.

1. In the main menu, click on “File” -> “New” ->”Project...” and select “Xtext” -> “Xtext Project”.




2. Fill in the data for the new project:
   a. Project Name “org.tutorial.survey”
   b. Language Name “org.tutorial.survey.Survey”
   c. Language Extension “survey”
   d. For all other fields the default values will do just fine.

3. Click “Finish”.

4. Find the GenerateSurvey.mwe2 file and run it. You can do so via clicking on “Run As” -> “MWE2 Workflow” in the file’s context menu. You’ll see the Console View popping up and several log messages scrolling by. Please take a close look and ensure the last log message of the workflow is “Done.”. If it is not, you will probably find an error message instead.

5. Take a close look at the new projects in your workspace and see what is there.

### Exercise 1: Write a Grammar for an Example File

Learning Objective: Understand how grammars describe syntax and map the parsed tokens to trees of objects (aka Abstract Syntax Trees, ASTs).

Task: Write a grammar so that the following text can be parsed without errors:

```
survey tutorial "EclipseCon 2013 Tutorial Survey"

page Start (
	text name 'Your name'
	single choice like "Do you like the tutorial?" (
		yes "Yes"
		no "No"
	)
)

page Like (
	choice particular "What do you like in particular?" (
		xtext 'Xtext is awesome'
		excercises 'The funny exercises'
		tutors 'The attractive tutors'
	)
)

page Dislike (
	choice particular "What do you hate in particular?" (
		xtext 'Xtext sucks'
		excercises 'The boring exercises'
		tutors 'The tutors stink'
	)
)
```

You'll need to repeat the following steps until your *.survey-Editor doesn't show syntax errors anymore.

1. Edit your grammar (*.xtext file).
2. Execute your workflow (*.mwe2 file) to regenerate the parser.
3. Launch a Runtime Workbench by clicking on "Run" -> "Run Configurations..." in the main menu and choose "Launch Runtime Eclipse". A Runtime Workbench is a new instance of Eclipse in which the projects from your host workspace are installed as plugins.
4. Create or open a file that has the file extension survey. If you opened the file, you might need to change its contents and save it to re-validate it for syntax errors.

These requirements may guide you:

- A survey contains several pages.
- A page contains several questions.
- A question can be a choice question or a text question.
- A choice question presents a list of answers to the user. The user can choose either exactly one any number of answers.
- A text question will provide a text field for entering an answer to the user.
- An answer can be a static text or a text field with a label.

To get going you may start with the following parser rule:

```
Survey:
  'survey' name=ID title=STRING pages+=Page*;
```

This rule can parse the string `survey community "Xtext User Survey"` and any number of pages. The called rule `ID` parses `community` and the rule `STRING` parses `"Xtext User Survey"`. Xtext will also create a Java Class[1] which is then instantiated by the parser. The assignments `name=` and `title=` assign the parsed values to the object's properties.

To parse a page, you can use the rule:

```
Page:
  'page' name=ID '('
    questions+=Question*
  ')';
```

To express that a `Question` delegates to either a `FreeTextQuestion` or a `ChoiceQuestion` you can use the following rule:

```
Question:
  FreeTextQuestion | ChoiceQuestion;
```

This should be enough hints to get started. For further help you might want to consult the solution for this exercise.

### Exercise 1 Bonus

So far we do not have control on the page flow. The task now is to extend the grammar so that the text on the right can be parsed. Every page can hold a cross reference (e.g. `-> hobby`) that points to the next page. These references will then determinate the order of the pages.

The grammar syntax for cross references is:
`assignment=[TargetClassName|TerminalRule]`
For us, `TargetClassName` is `Page` and `TerminalRule` is `ID`.

In the Runtime Workbench, when you position the cursor at a cross reference and trigger content assist (CTRL + SPACE), this will show a dropdown-box listing all valid names for the cross reference.

```
survey community "Xtext User Survey"

page eclipse (
  single choice version
    "What is ... you used?" (
     text 'Other (please specify)'
  )
  -> hobby
)

page hobby (
  text motivation "Why ... software?"
  -> os
)

page oeprating_system (
  text os "What OS do you use?"
)
```

[1] To be precise, Xtext infers an EMF EClass and EMF generates the Java Class

### Exercise 2: Implement a Code Generator

Learning Objective: Understand how parsed documents can be transformed to code.

In this exercise, we want to implement a code generator that generates HTML files and a Java file (latter only if you do the bonus exercise). Each HTML file represents a page from the survey and contains a HTML form with input elements for all questions. The Java file defines the flow through the pages.

### Exercise 2.1: Generate a File

Open `/org.tutorial.survey/src/org/tutorial/survey/generator/SurveyGenerator.xtend`. The actual name may be different depending on how you named you project/language. In there, you'll find

```
class SurveyGenerator implements IGenerator {
  override void doGenerate(Resource resource, IFileSystemAccess fsa) {
  }
}
```

The method `doGenerate(...)` is invoked for every DSL file (*.survey in our case) for which re-generation is necessary. Now paste the following code into the methods body:

```
override void doGenerate(Resource resource, IFileSystemAccess fsa) {
  val survey = resource.contents.head as Survey
  if(survey != null) {
    for (page : survey.pages) {
      fsa.generateFile(page.name + ".html", toHtml(survey, page))
    }
  }
}
```

and create the new method

```
protected def toHtml(Survey survey, Page page) '''
   <html>
     <body>
       Hello World
     </body>
   </html>
'''
```

This is all we need to implement a basic code generator. If you now (re-)launch your Runtime Workbench and edit and save a *.survey file, you should see generated files showing up in the "src-gen" folder of that project. For each survey page, there should be one generated HTML file.

### Exercise 2.2: Generate Useful File Contents

Open `/org.eclipse.xtext.tutorial.survey.reference/webroot/Start.html` in a text editor (Context Menu -> Open With -> Text Editor) and paste the contents between the ‘” ‘” of the method `toHtml()`.

Of course `Start.html` is a concrete page with all the questions and choices hard coded in. Our next step will be to replace these hard coded values with placeholders and control structures. The placeholders will be fed with the Object Model create by the parser.

Steps:

1. Replace `<title>Eclipse Community Survey 2012</title>` with `<title>«survey.name»</title>`. This will dynamically insert the survey’s name from the DSL file.
2. Replace

```
<div class="control-group">
  <label class="control-label">Please leave any other comments...</label>
  <div class="controls">
    <input type="text" name="comments">
  </div>
</div>
```

with

```
«FOR question: questions»
  «question.controlGroup»
«ENDFOR»
```

and declare two new methods

```
protected def dispatch controlGroup(FreeTextQuestion it) '''
  <div class="control-group">
    <label class="control-label">«text»</label>
    <div class="controls">
      <input type="text" name="«name»">
    </div>
  </div>
'''

protected def dispatch controlGroup(ChoiceQuestion it) {
}
```

This adds support to generate HTML code for free-text questions. You may implement the template for ChoiceQuestion by yourself.

3. Replace the remaining part from the HTML to take into account the values from AST. You may also use if-statements: `«IF choices.size > 30»foo«ENDIF»`.

### Exercise 2 Bonus

The file `/org.eclipse.xtext.tutorial.survey.reference/src/main/PageFlow.java` defines the order of the pages: Is specifies for every page, which follow-up page will be presented when the user submits a form with the “next” button.

Until now, this file is hard coded in Java and specific to the pages that we shipped with the reference project. Surly the page flow needs to take into account the pages that are defined in the *.survey file. You can solve this by generating `PageFlow.java`.

Steps:

1. Open `/org.tutorial.survey/src/org/tutorial/survey/generator/SurveyGenerator.xtend`
2. Extend `SurveyGenerator.xtend` so that a file `src-gen/main/PageFlow.java` is generated. Initially you can generate that content as you find it in `/org.eclipse.xtext.tutorial.survey.reference/src/main/PageFlow.java`
3. Customize how the method `public String getNextPage(IFormState formState)` is generated: This method receives the and object that knows the current page (`formState.getCurrentPage()`) as a parameter and returns the name of the follow-up page as a string.
   a. If you have completed exercise 1 bonus, every page object in your AST should have a cross reference called “next” that points to the expected follow-up page.
   b. If you have not completed exercise 1 bonus, you can use the order of pages in which they are declared in the *.survey files. You’ll find the AST-element‘s Java-List `survey.getPages()` to be in exactly that order.

### Exercise 3: Implement a Custom Validation Rule

Learning Objective: Get familiar with the API to write validation rules.

### Exercise 3.1: Raise Errors if an Answer is an Empty String.

Steps:

1. Open the file `/org.tutorial.survey/src/org/tutorial/survey/validation/SurveyValidator.xtend`
2. In this file, add `import static org.tutorial.survey.survey.SurveyPackage$Literals.*`. This class holds constants that represent properties in our AST.
3. Paste the method

```
@Check
def textMustNotBeEmpty(Question question) {
  if(question.text.empty) {
    error("Empty question is illegal", question, QUESTION__TEXT)
  }
}
```

4. To try out the validation rule, launch the Runtime Workbench and create *.survey file with empty question text. An error marker should appear immediately and put a red curly line under the empty quotes.

### Exercise 3.1: Raise Errors if two Answers have the same Text.

For the user it doesn't make sense if two possible answers to a question are the same, i.e. have the same text. Therefore it's now your task to implement a validation rule that compares answer texts for equality.

Hint: It's a common performance pitfall to compare every text with every other text (quadratic complexity). A better pattern is to iterate over the list of texts once and populate a `Map<String, Choice>`. Before adding a text to the map, you can check it is already contained. In that case, the current Choice object and the one from the map have equal names and deserver error markers.

### Exercise 4: Customize Linking and Content Assist via Scoping

Learning Objective: Understand Cross References and how cross-referencing is controlled by Scoping.

Until now the order in which the user is guided through the survey pages was defined by the order of the pages in the *.survey file or, if you have solved Exercise 1 Bonus, by cross references pointing from one page to another.
We want to extend this by allowing guarded transitions from page to page. Example:

```
page cloud (
  single choice intent "Do you work with clouds?" (
      'Yes'
      no 'No'
      don_t_know "Don't know"
  )
  if intent=no -> mobile
  if intent=don_t_know -> mobile
  -> cloud_2
)
```

One page can have multiple following pages and the first page of which the guard evaluates to true is chosen as the actual follower. Guards are boolean conditions and they're evaluated on the answers the user already has provided. This allows to choose the next page based on the selected answers. To keep things simple, we only want to support a comparison operator for now.

1. Make sure your grammar contains these parser rules:

```
Page:
  'page' name=ID '('
    questions+=Question*
    followUps+=FollowUp*
  ')';

FollowUp:
  guard=Guard? '->' next=[Page];

Guard:
  'if' question=[ChoiceQuestion|QualifiedName] '=' answer=[Choice|QualifiedName];

QualifiedName:
  ID ('.' ID)*;
```

2. Open the file `/org.tutorial.survey/src/org/tutorial/survey/scoping/SurveyScopeProvider.xtend`

3. Paste the method

```
def scope_Guard_answer(Guard guard, EReference ref) {
  if(guard.question == null)
    IScope::NULLSCOPE
  else
    Scopes::scopeFor(guard.question.choices)
}
```

4. Adopt the code generator for the page flows to support guards.