Devilry Documentation

Release 2.0.20-

The Devilry Team

August 08, 2018

CHAPTER 1

Table of contents

Devilry user documentation

Getting started

We recommend that you start with a quick look at *Common concepts*. This avoids confusion with a few special terms that Devilry uses to allow it to be used in many different settings. When you have skimmed over the concepts, you can continue with the other Topic guides below.

Common for all roles

Common concepts

Special terms and concepts

Devilry has some special terms and concepts. The most important (that cause most confusion) is:

- **Term**: A range of time. Typically a semester or a year.
- **Examiner**: Someone that provides feedback.
- **Group**: Students are always in a (project) group even when they work alone on an assignment.

More details about these and more terms and concepts follows below.

Node A Node is a place to organise top-level administrators (I.E.: administrators responsible for more than one *Subject (course)*). Nodes are organised in a tree. This is very flexible, and can be used to emulate most administrative hierarchies. A node is often a department, or some other organizational unit, but the exact use in your local Devilry instance depends on how you choose to organize administrators in Devilry.

Subject (course) A subject is, as far as Devilry is concerned, a container of *Terms*. In a typical Devilry setup, a Subject is the same as a Course, and each *Term* within the Subject is a semester or year.

Term (semester, year, ...) A term is a limited span of time (I.E: *january to july 2011*) that you give a name (I.E.: *Spring 2011*). You register assignments on a term, and register students and examiners on each assignment.

Group, Candidate and Student Students are not registered directly on an assignment. Instead a group is created, and one or more students is added as Candidates on that group. This means that project assignments, where students cooperate, is organized exactly like any other assignment. The only difference is the number of Candidates in each group.

A Candidate can also have a candidate ID, which is used to identify the student on anonymous assignments like exams.

See also:

The Student role.

Deadline Deadlines are individual for each group. They are organized *below* a Group in the Devilry hierarchy. In other words: Each Group has one or more deadlines.

Examiner Examiner is someone that writes feedback. Examiners are often one of these:

- A teacher that corrects their own students. They are usually Term or Subject administrator in addition to Examiner.
- A teaching assistant.
- Someone giving anonymous feedback on an exam.

A user becomes examiner when they are assigned as examiner for a group (See *Group, Candidate and Student*) by an administrator.

See also:

The Examiner role.

Special terms in context — a typical Devilry hierarchy The tree below is an example of a typical Devilry hierarchy for a university named *Duckburgh University* with the special terms in brackets.

- **Duckburgh University [*Node*]**
    - **Department of Physics [*Node*]**
        - **PHYS 101 — Introduction to physics [*Subject (course)*]**
            - **Spring 2011 [*Term (semester, year, ...)*]**

              Assignment one

              Peter Pan and Wendy [*Group, Candidate and Student*]

              Deadline feb. 27 2012 19:30 [*Deadline*]

              Delivery 1

              Captain Hook [*Group, Candidate and Student*]

              Deadline mar. 12 2012 11:45 [*Deadline*]

              Delivery 3

              Deadline feb. 28 2012 12:30 [*Deadline*]

              Delivery 2

              Delivery 1

              John Doe [*Group, Candidate and Student*]

              Deadline feb. 25 2012 23:35 [*Deadline*]

Delivery 1

        · Spring 2012 [*Term (semester, year, ...)*]

        · Spring 2013 [*Term (semester, year, ...)*]

    * PHYS 302 — Advanced physics [*Subject (course)*]

    * ...

- **Department of Informatics [*Node*]**

    * INF 101 — Introduction to programming [*Subject (course)*]

    * INF 102 — Objectoriented programming [*Subject (course)*]

    * ...

- ...

Email sending in Devilry

Email sending is highly configurable in Devilry. This guide deals with the default configuration.

When does Devilry send email?

Whenever a student makes a delivery, and whenever they get a new feedback.

How to test email sending (for system admins)

Go to the frontpage, select the *Superuser* role, select *Users*, select a user and click *Send a test-email to USERNAME* in the upper right corner.

Student

The Student role

The student interface should be intuitive to use. Please post an issue with our Issue tracker if that is not the case.

How to create project groups (collaborate on an assignment)

How it works

Project groups are created on a per assignment basis. Features:

- Any member of the group can make deliveries on behalf of the group.
- Any feedback given to the project group is for all group members. If you are supposed to get individual feedback, you should not be in a project group.
- When you join a group, you get any deliveries and feedback made by the group before joining, and they get any deliveries and feedback you may have had when joining the group.
- When you leave a group, you do not loose any deliveries and feedback. You even keep deliveries and feedback made before you joined the group.

Invite other students to join your group

If your course administrator have enabled collaboration, you can invite other students to join your group as follows:

1. Log in to Devilry.
2. Select the *Student* role on the frontpage.
3. Select the assignment.
4. Select *Project group* in the list on the left.
5. It will say *Invite someone to join your group?* at the top of the page. Under you can select the student you want to invite to your group, and send an invite. Other students will get their invite via email, and they can accept or decline the invite. You can delete an unanswered invite. If any, your group members will be in the list *Project group members* at the bottom of the page.

Warning: All students you invite to your group will given all current deliveries and feedback, even deliveries and feedbacks made before they joined the group. If they leave the group, they will keep all deliveries and feedback you have received on the assignment, even feedback and deliveries made before they joined the group.

Note: Refer your course administrator to *How to administer project groups (students that collaborate)* if you think they should enable collaboration.

Leave a group / kick a member

You need to ask a course administrator if you want to leave a group or kick a group member. Leaving a group or kicking a member is perfectly safe. Any member leaving a group is simply moved into a complete copy of the group including all deliveries and feedback. The only difference is that the original group gets one less member, and the new group will only have one member.

Examiner/Corrector

The Examiner role

Note: Examiner is someone that writes feedback. Examiners are often one of these:

- Someone responsible for correcting some or all students in a subject/course.
- A teacher that corrects their own students. They are usually Period or Subject administrator in addition to Examiner.
- Someone giving anonymous feedback on an exam.

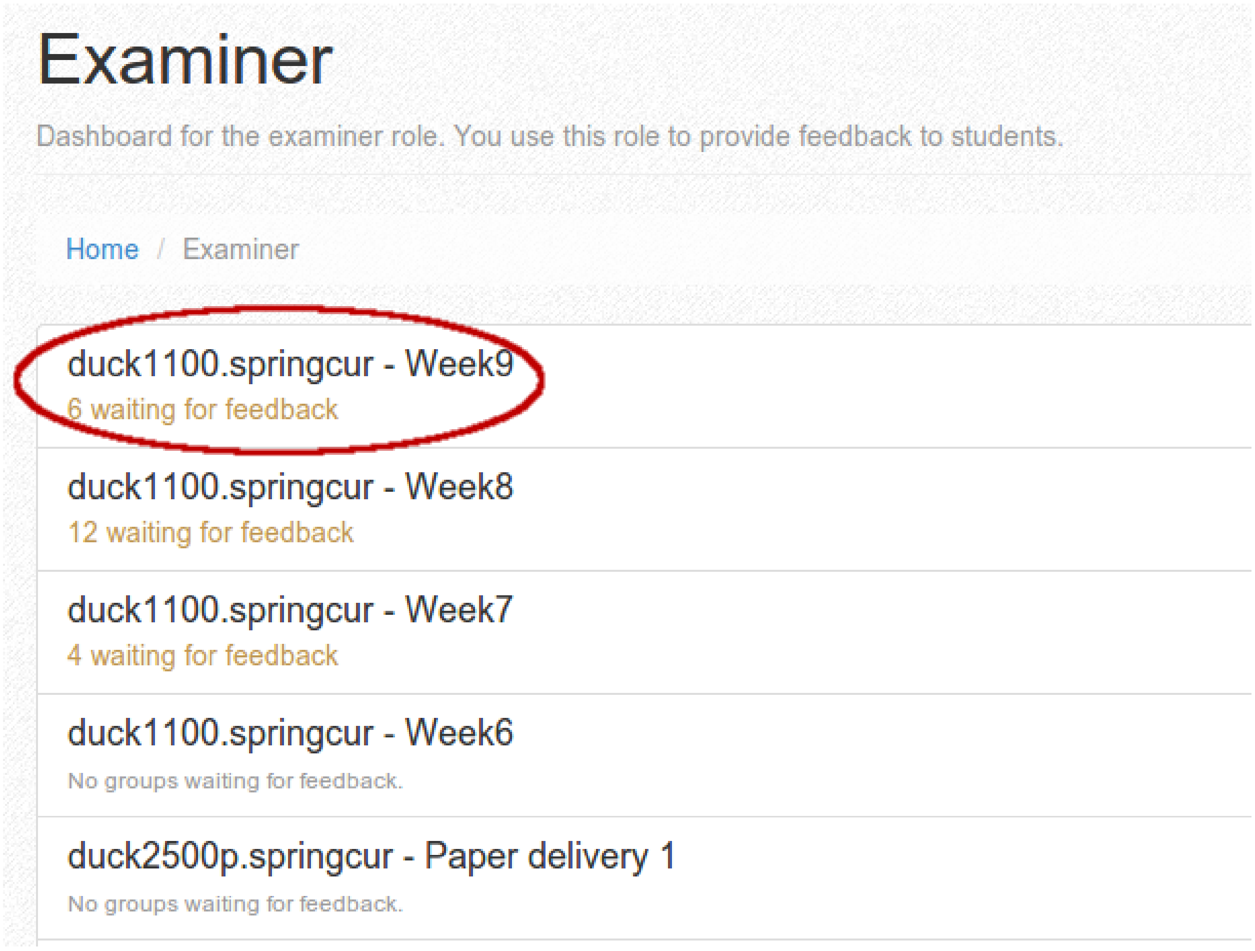
Getting started

Getting started guide for examiners

Note: To avoid confusion when reading this guide, please read *Common concepts*, at least the *Group, Candidate and Student* section.

Note: This guide is under construction. Please contact devilry-support@ifi.uio.no if you have questions of any kind related to Devilry.

Choose the examiner role After successful login you need to choose *Examiner* from the list of available roles.



Examiner

Dashboard for the examiner role. You use this role to provide feedback to students.

Home / Examiner

duck1100.springcur - Week9
6 waiting for feedback

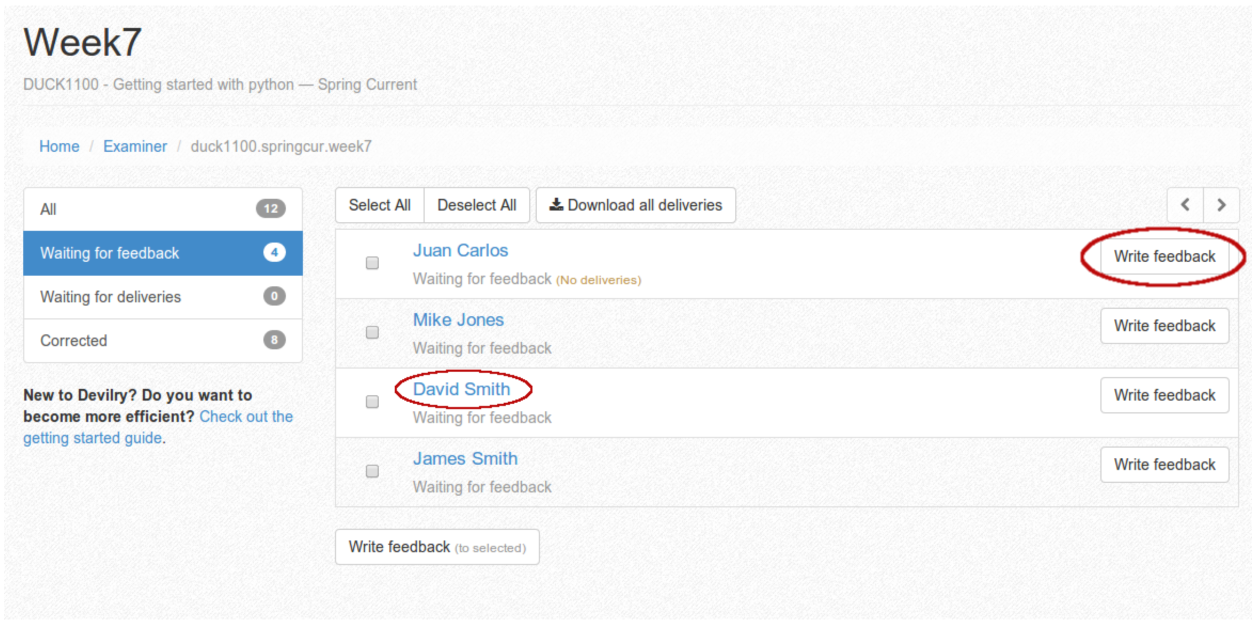
duck1100.springcur - Week8
12 waiting for feedback

duck1100.springcur - Week7
4 waiting for feedback

duck1100.springcur - Week6
No groups waiting for feedback.

duck2500p.springcur - Paper delivery 1
No groups waiting for feedback.

Select an assignment On the examiner *dashboard*, assignments where you are examiners are listed. To start correcting, you must choose the spesific assignment in which you want to work with. The list are ordered by the time the assignments where published in descending order. This ensure that your latest assignment are listed on top.



Select the group to correct After the assignment is choosen you are redirected to a view that list all groups on the assignment.

The list may be filtered to show a more fine grained selection:

Waiting for feedback Shows the assignment groups that have at least one valid delivery and are waiting for feedback to be provided by the examiner.

Waiting for deliveries Shows the assignment groups that have **not** provided a valid delivery yet.

Corrected Shows the assignment groups that are corrected. Be aware that *Corrected* will list both failed and passed groups as long as they are corrected.

All Shows every assignment group on the assignment where yourself are the assigned examiner. This is the default filter upon entering the view.

To start correcting you would normally filter with the *Waiting for feedback* option. The **Write feedback** button on the group item in the list will take you directly to the latest delivery provided by the group. see *Add feedback to the delivery* for further details.

If you need access to older deliveries provided by the group you must click on the group name header in the list item which redirect you a detailed view for that assignment. More information on this view in *Correct an earlier delivery*

Add feedback to the delivery Upon entering the delivery you will need to click the *Provide feedback* button to start giving feedback.

Provide Feedback This view may vary depending on the grading system configured for the assignment. The picture show a simple points system with a form to provide the number points achieved.

Every grading system features the ability to provide a feedback text in addition to the grade information. Click inside the textbox and a `WYSIWYG Markdown` editor will be shown. Just type in the feedback and push the *Publish* button to publish the feedback without examine a preview. If you lack experience with `Markdown` you would probably want to click the *Preview* button to be able to secure that the feedback appear as supposed.

Feedback drafts and history You may save you work for later by clicking the *Save Draft* button

Correct the next delivery When there is multiple groups *Waiting for feedback* you can loop through each group by clicking the arrow button in the upper right corner of the view.

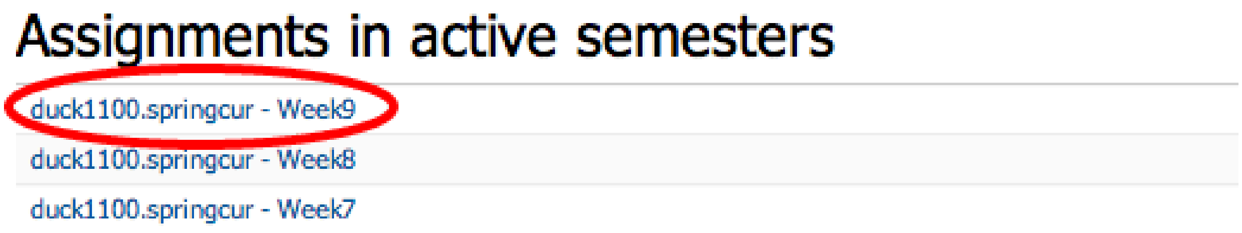
Correct an earlier delivery When choosing the spesific group every delivery attempts will be listed categorised by the their respective deadlines. This makes it possible to correct a delivery that is not the latest one.

How to examine in bulk or correct non electronic deliveries

Note: To avoid total confusion when reading this guide, please read *Common concepts*, at least the *Group, Candidate and Student*-section.

Introduction This is a guide for examiners who want to give the same feedback to multiple groups at once and for examiners giving feedback to assignments where Devilry is only used to register results(not for deliveries).

Choose the examiner role It should be one of the available roles on your frontpage.

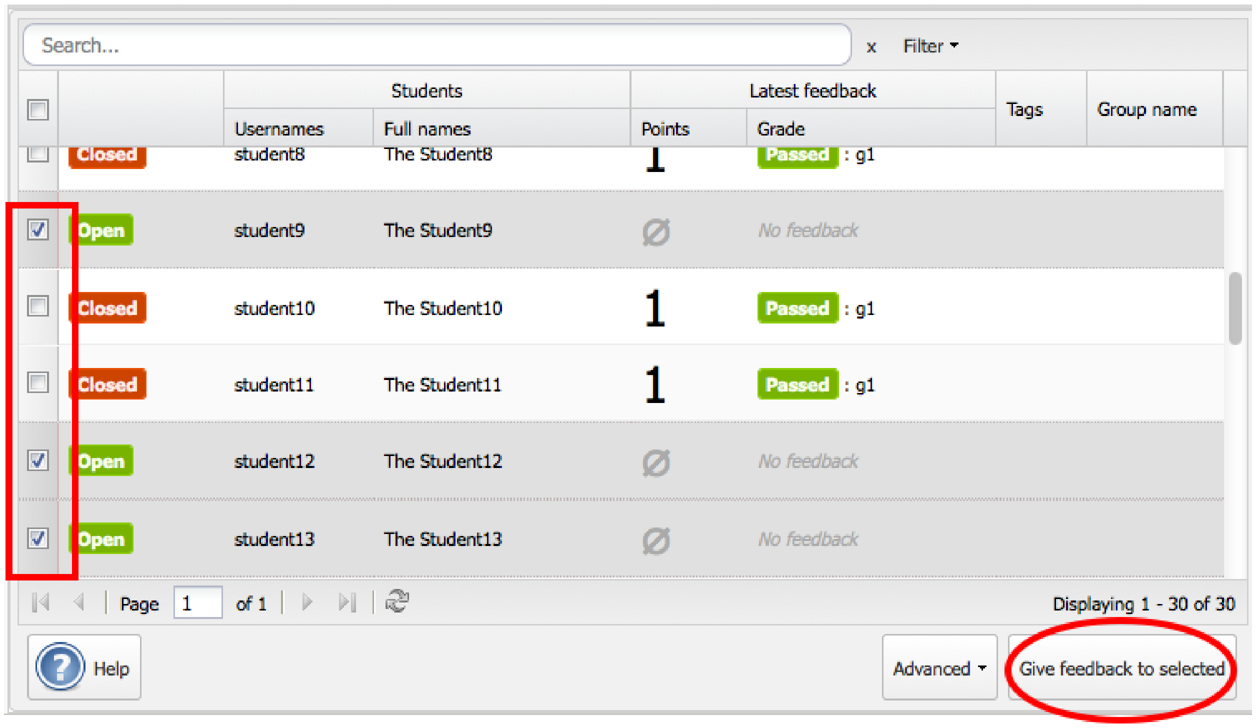

Assignments in active semesters

duck1100.springcur - Week9

duck1100.springcur - Week8

duck1100.springcur - Week7

Select an assignment On the examiner *dashboard*, assignments where you are examiners are listed ordered by publishing time in descending order. Choose an assignment from this list to get start giving feedback on that assignment.



		Students		Latest feedback		Tags	Group name
		Usernames	Full names	Points	Grade		
☐	Closed	student8	The Student8	1	Passed : g1		
☑	Open	student9	The Student9	∅	No feedback		
☐	Closed	student10	The Student10	1	Passed : g1		
☐	Closed	student11	The Student11	1	Passed : g1		
☑	Open	student12	The Student12	∅	No feedback		
☑	Open	student13	The Student13	∅	No feedback		

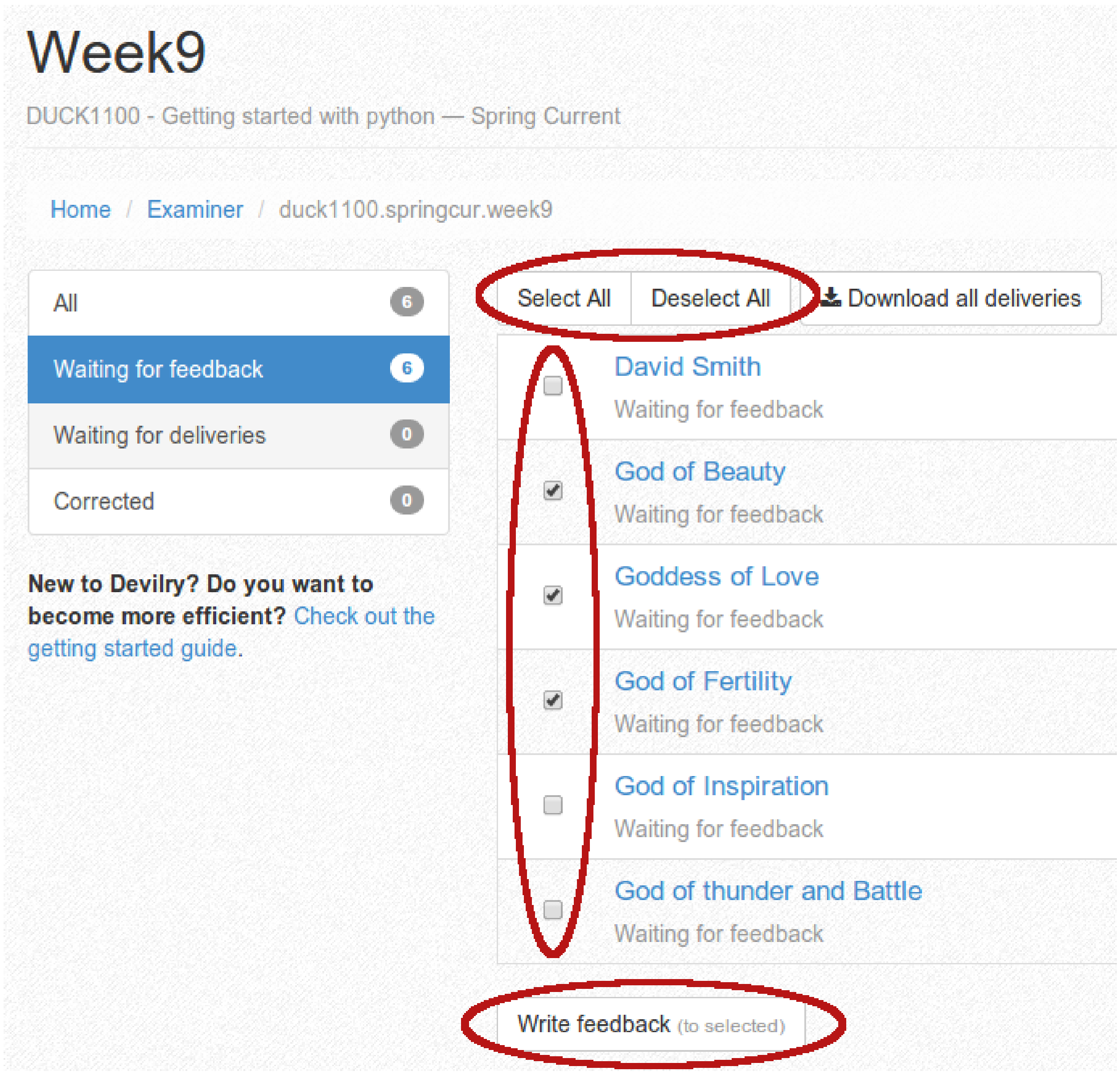
Group overview When you enter the examiner interfance, you will see an overview of all groups on that assignment. Choose one or more groups and click *Give feedback to selected*. A window containing the *grade editor* is shown. Click the Help-button in the lower left corner of the *grade editor* for more help.

Note: You can right-click anywhere in the group overview for quick access to everything in the toolbar.

Other examiner guides

How to correct several groups at once

Note: To avoid confusion when reading this guide, please read *Common concepts*, at least the *Group, Candidate and Student*-section.


Week9

DUCK1100 - Getting started with python — Spring Current

Home / Examiner / duck1100.springcur.week9

All	6
Waiting for feedback	6
Waiting for deliveries	0
Corrected	0

New to Devilry? Do you want to become more efficient? Check out the getting started guide.

Select All | Deselect All | Download all deliveries

☐	David Smith Waiting for feedback
✓	God of Beauty Waiting for feedback
✓	Goddess of Love Waiting for feedback
✓	God of Fertility Waiting for feedback
☐	God of Inspiration Waiting for feedback
☐	God of thunder and Battle Waiting for feedback

Write feedback (to selected)

Introduction This guide show you how to edit feedback for a selection of groups. To get to the view described here you must first choose the examiner role in the Devilry frontpage and then choose an assignment.

Editing of multiple groups is easy and straightforward. Activate the checkboxes next to the group of interest to add it to your selection. If you want to choose all groups within the active filter you may click *Select All* button to speed up the process. When you have created your selection click the *Write feedback* button to start providing feedback.

Provide feedback The groups that you selected are listed after the `Edit feedback for:` text, please make sure that it is complete and correct before you take further actions. Provide your feedback. Click *Preview* button to get a glimpse on how your feedback will look when completely rendered. Be aware that *Preview* will save your feedback draft on each group. You will be able to come back and edit further in the next step.

If you want to publish your work right away just click *Publish*.

Manage deadlines — for Examiners

Add a deadline When an assignment is created, an initial deadline is set. You can not change this deadline, however you can extend the deadline by creating a new deadline. You usually create deadlines for the following reasons:

A student gets a failing grade If you want to give the student a change to get a new attempt Devilry will ask you (when you publish a failing feedback) if you wish to give the group a new deadline.

Move a deadline Currently, examiners can not move deadlines. This is coming along with an update of the examiner UI. For now, you have to ask an administrator if you need to move a deadline.

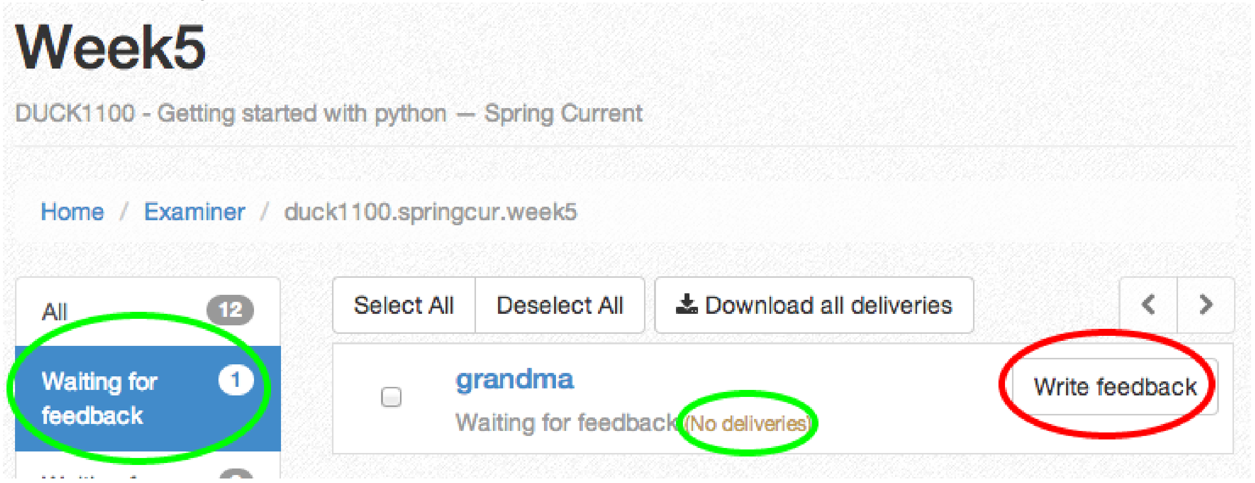
How to add feedback to a student without deliveries

Note: This guide is a supplement to *Getting started guide for examiners*.

Choose the examiner role It should be one of the available roles on your frontpage.

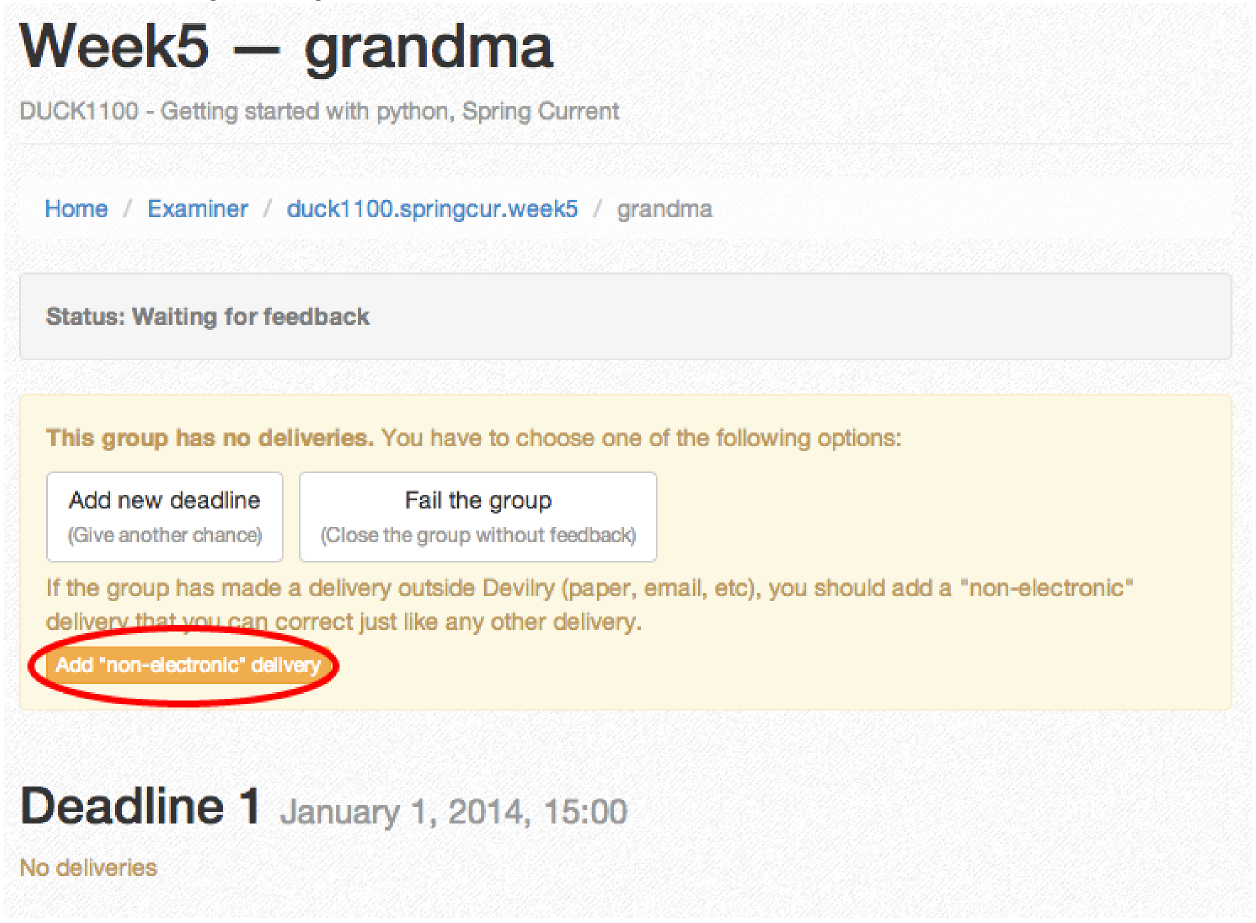
Select an assignment On the examiner *dashboard*, assignments where you are examiners are listed ordered by publishing time in descending order. Choose an assignment from this list to get start giving feedback on that assignment.

Select a group without any deliveries Select a group that waiting for feedback, but does not have any deliveries, as illustrated in the figure below:


Week5
DUCK1100 - Getting started with python — Spring Current
Home / Examiner / duck1100.springcur.week5
All 12
Waiting for feedback 1
Select All
Deselect All
Download all deliveries
grandma
Waiting for feedback (No deliveries)
Write feedback

Click the "add non-electronic delivery" button If the group: - Has no deliveries. - Is waiting for feedback (their last deadline has expired).

You will see a orange warning-box like the one in this illustration:



Week5 — grandma

DUCK1100 - Getting started with python, Spring Current

Home / Examiner / duck1100.springcur.week5 / grandma

Status: Waiting for feedback

This group has no deliveries. You have to choose one of the following options:

If the group has made a delivery outside Devilry (paper, email, etc), you should add a "non-electronic" delivery that you can correct just like any other delivery.

Deadline 1 January 1, 2014, 15:00

No deliveries

Click the *add non-electronic delivery* button, and follow the instructions on the next page.

Devilry flavoured Markdown When you write feedback to your students, you use the Markdown text formatting language.

With Markdown, you to write using an easy-to-read, easy-to-write plain text format, and let someone else (Devilry) worry about how the results will look. This makes it possible to write feedback text that Devilry can optimize for anything from smartphones to large desktop displays.

Basics

Paragraphs Paragraphs are just one or more lines of consecutive text followed by one or more blank lines:

```
Maecenas faucibus mollis interdum. Vestibulum id ligula porta felis euismod
semper. Vestibulum id ligula porta felis euismod semper. Aenean lacinia
bibendum nulla sed consectetur.

Donec id elit non mi porta gravida at eget metus. Vestibulum id ligula
porta felis euismod semper. Praesent commodo cursus magna, vel scelerisque
nisl consectetur et.
```

Headings

```
# Largest heading
## Second largest heading
...
##### Very small heading
```

Text styles

```
*Italic text*
**Bold text**
```

Links

```
Check out [http://devilry.org](The devilry website).
```

Lists Unordered lists (bullet lists):

```
* This
* is
* a
* test
```

Ordered lists (numbered lists):

```
1. Item one
2. Item two
3. Item three
```

Blockquotes

```
As stated on the first page of the 101 guide:

> You have to learn to walk before you can learn how to run
```

Advanced

Escape Markdown characters If you want to use a special Markdown character in your document (such as displaying literal asterisks), you can escape the character with a backslash. Markdown will ignore the character directly after a backslash. Example:

```
This is how the \_ (underscore) and \* asterisks characters look.
```

LaTeX Math Devilry Markdown supports LaTeX math through the MathJAX library/renderer. Examples:

````
A simple example:
$mathblock$
^3/_7
$/mathblock$

The Lorenz Equations:
$mathblock$
\begin{aligned}
\dot{x} & = \sigma(y-x) \\\\
\dot{y} & = \rho x - y - xz \\\\
\dot{z} & = -\beta z + xy
\end{aligned}
$/mathblock$
````

You have to escape special Markdown characters such as `\`, which is why we have `\\\\` at the end of our lines in the example above instead of just `\\`.

Code blocks You can easily show syntax highlighted code blocks:

~~~
Java code:
``` java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

Python code:
``` python
if __name__ == "__main__":
    print "Hello world"
```

C code:
``` c
#include<stdio.h>
int main() {
    printf("Hello World");
    return 0;
}
```

C++ code:
``` c++
#include <iostream>
int main() {
    std::cout << "Hello World!";
    return 0;
}
```

HTML example:
``` html
<html>
    <body>
        <h1>Hello world</h1>
~~~

~~~
    </body>
</html>
```

CSS example:
``` css
body {
    background-color: pink;
    color: green;
    font-size: 80px;
}
```

Any code:
```
for x in 1 through 3
    show x
```
~~~

Devilry supports all languages supported by Pygments.

Frequently asked questions

Examiner UI FAQ

How can I set a new deadline? When the deadline is up you will be asked if you want to provide a new deadline for students to deliver.

What can an examiner do?

Examiners can basically do anything non-destructive with groups (see *Group, Candidate and Student*) that they have been assigned to:

- View all feedback for the groups.
- Add new deadlines to their groups.
- Give feedback.
- Change feedback. Each change to the published feedback is logged, and the students can view *all* published feedback.

Examiners can not:

- Delete groups.
- Add new groups.
- Add or remove students to groups.
- Move deadlines. This is a missing feature, which will be implemented in a future release. It will be safe becuase students will remain protected because of logging of all changes. Moving deadlines is already implemented for administrators.
- Remove deadlines from groups. We will probably allow examiners to remove deadlines without deliveries when we start allowing them to move deadlines.

Subject/Course administrator

Note: For users managing one or more courses.

Introduction to the Subjectadmin role

Note: Please read, or at least take a quick look at, the *Common concepts* before reading this guide.

Who this guide is for

A subject is, as explained in the common concepts guide, typically a course. This means that this guide is for administrators managing a course or a term (semester) within a course. If you have *orange* background color in your header, you are using the user interface for the Subjectadmin role.

The responibilities of a Subjectadmin

A Subjectadmin manage one or more *Subject (course)*, and/or *Term (semester, year, ...)*. They set up assignments, organize students into *Groups*, and assignment *Examiners* to give feedback to students.

Commmon tasks

Finding the Subjectadmin frontpage The Subjectadmin frontpage is the page that you navigate to when you select the Subject administrator role (may be something like *Course administrator* in your local dialect), from the Devilry frontpage.

Create an assignment We provide an interactive guide to help you create assignments. Simply go to the Subjectadmin frontpage (see *Finding the Subjectadmin frontpage*), and select the guide on the right hand side.

Get an overview over all your students Select an active *Term (semester, year, ...)* from the Subjectadmin frontpage (see *Finding the Subjectadmin frontpage*), or an old/expired *Term (semester, year, ...)* via the link further down on the frontpage.

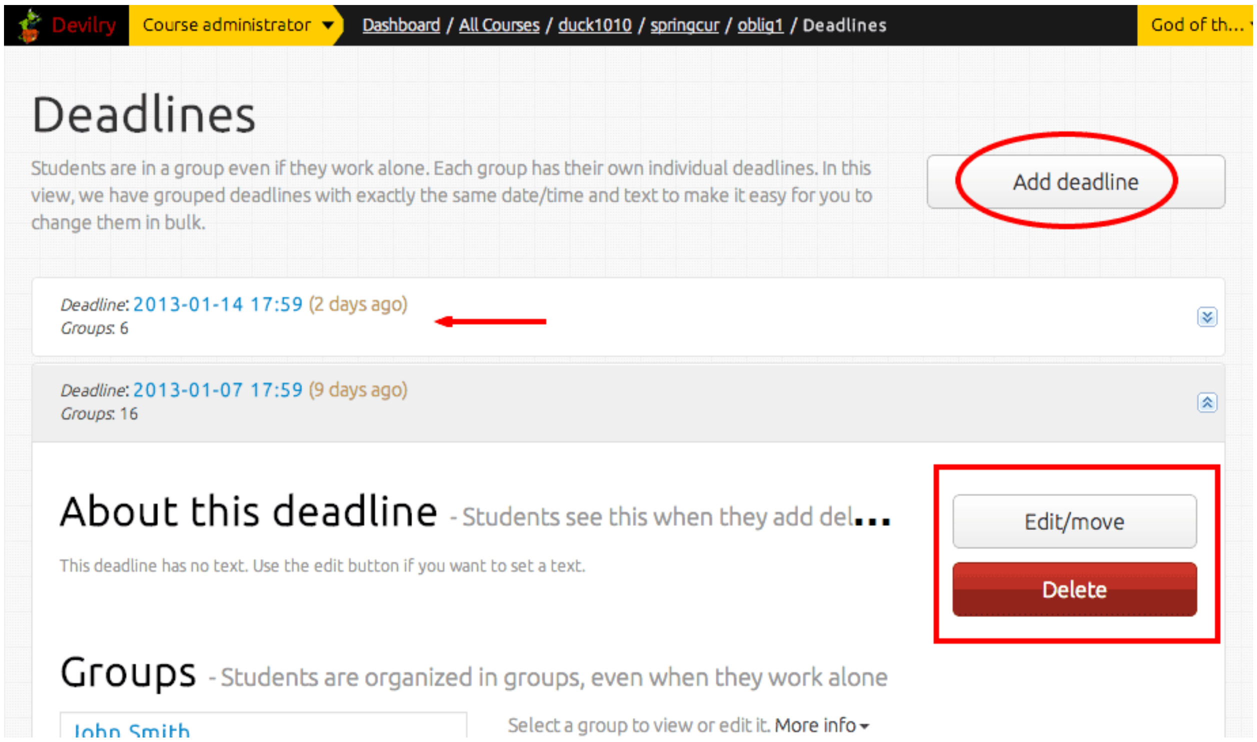
Under the *Edit and view related information* heading, you will find links to your *Students*, and a link to an *Overview of all results*. You can export (download) the overview as MS Excel, CSV, and a couple of other formats. The export-links is in the toolbar right below the page heading.

Manage deadlines — for Subjectadmins

Subject administrators can create, view, move and edit deadlines using the *Deadlines* link on the main page for an assignment.

Overview of the deadline view

The image below is a typical deadline overview. Each of the red-marked areas is explained below the image.


Devilry
Course administrator
Dashboard / All Courses / duck1010 / springcur / oblig1 / Deadlines
God of th...
Add deadline
Edit/move
Delete

Deadlines

Students are in a group even if they work alone. Each group has their own individual deadlines. In this view, we have grouped deadlines with exactly the same date/time and text to make it easy for you to change them in bulk.

Deadline: 2013-01-14 17:59 (2 days ago)
Groups: 6

Deadline: 2013-01-07 17:59 (9 days ago)
Groups: 16

About this deadline - Students see this when they add del...

This deadline has no text. Use the edit button if you want to set a text.

Groups - Students are organized in groups, even when they work alone

John Smith

Select a group to view or edit it. More info

Add deadline (hilighted with a red circle) Click the Add deadline button to a add deadline to one or more groups. You can select individual groups, and Devilry has shortcuts for a couple of common choices.

Expand/collapse deadline (hilighted by a red arrow) Deadlines start out collapsed when you enter a view. Click the deadline to expand or collapse it.

Move/edit deadline (hilighted by a red rectangle) To move or edit (the about-text for) a deadline, expand the deadline, and select the edit-button.

You can change the deadline for only some of the groups within the deadline, effectively splitting the deadline in two. This is explained on the right-hand side of the list of groups that appear when you choose that option.

Delete deadline (hilighted by a red rectangle) You can delete a deadline, but only superusers can delete deadlines where groups have made deliveries.

Hard VS soft deadlines

Soft or hard deadlines is configured on the left hand side in the assignment overview under the *Deadline handling* heading. Use the *More info* button to get detailed information about soft and hard deadlines.

What about examiners?

Examiners can add deadlines. It is a natural part of their workflow. Whenever they fail a student, they are asked to do one of the following:

- Leave the student with a failing grade.

- Give them another chance — create a new deadline.

How to administer project groups (students that collaborate)

Concepts and features

Devilry is designed with cooperative deliveries in mind.

- Students are *always* in a group even when they work alone.
- Course admins can safely create and split up project groups at any time.
- You can *enable students to create project groups on their own*.
- Groups are created and managed per assignment. This means that changing a group on one assignment does not affect that group on another assignment. You can copy groups from another assignment when you create a new assignment.
- Students can be organized in groups even after they have made deliveries and been given feedback. They their respective groups are simply merged into a single group with all deliveries and feedback. The last feedback (if any) is made the active feedback.

How to enable students to create project groups on their own

If you want to allow students to form project groups on their own, you have to enable this option on the assignment:

1. Go to the overview page for an assignment where you have administrator rights.
2. Click any of the edit buttons in the sidebar to your left, except the edit button for *Grading system*.
3. Edit the options in the *Allow students to form project groups* section.

Note: To see how students form project groups on their own, see *How to create project groups (collaborate on an assignment)*. You should refer your students to that guide when you ask them to form their own groups. Devilry does not notify students when you enable this feature.

Note: This is not as intuitive as it should be. It will be made more intuitive in the future.

How to manually create a project group

1. Open the students overview on the relevant assignment.
2. Select two or more groups/students.
3. Select *Create project group*.

Exactly what this means is explained when you click *Create project group*, and you have to confirm before the group is created. In short,

How to remove a student from a group

Students can not leave groups on their own (yet). So an admin has to manage that:

1. Open the students overview on the relevant assignment.

2. Select the group.
3. Click the red minus button on the right hand side of the student you wish to remove from the group.

This will do the following:

- Create a copy of the group with *all* deliveries and feedback, even deliveries made by other students before the student you are removing joined the group.
- Add the student you are removing to the copy of the original group.

Examiners — How to set examiners, and how to create feedback

How to give feedback to students when you are Subjectadmin

Starting with Devilry 1.2.1 it is no longer possible to give feedback using the Subjectadmin role. You have to make yourself examiner.

How to make yourself examiner

The easiest way of making yourself examiner is to make yourself examiner when creating a new assignment. If you are one of many examiners, you will have to make sure you are tagged appropriately. The Create new assignment wizard helps you with this when you get to that step.

The other way of making yourself examiner is to do it manually after the assignment has been created. Select an assignment, and select the *Students*-link. The see the help-column for more help.

How to make others examiner

This is basically the same as the previous section. Just choose other users than yourself.

What do examiners have permission to do

See *What can an examiner do?*.

Note: The reason why it is no longer possible to provide feedback as Subjectadmin is that we have need to be able to optimize the workflows for examiners and subject admins independently. Mixing the roles leads to confusion in all but the most simple cases, and it increases the development time required for each change to any of the user-interfaces significantly.

Choose students that qualify for final exams

Quickstart

We have an interractive guide for this on your right-hand side on the Subjectadmin frontpage (see *Finding the Subjectadmin frontpage*). Select the guide, and follow its instructions.

How the qualified for final exams system works

You select the students that qualify for final exams using one of the provided plugins. Nodeadmins, or automatic exporters, read these lists to determine who can participate in the final exams.

Students can see if they are qualified for exams or not, but they can not se *why* they are qualified (they can not see what plugin was used, and with what settings). You should use the course website, or other approprite channels to inform your students about the requirements for final exams.

Changing or retracting

You can retract or change a saved *qualified for final exams*-status.

Warning: *Nodeadmins* are **not notified** when you retract or change a status. We are working on a system that handles updates/retracting, but that did not make it into the first release of the *Qualifies for final exams* app. Please notify the people coordinating final exams for your department/organizational unit if you change a status.

To change a status, simply use the button at the bottom of the box at the top of the page showing a status.

No plugin fits my needs!

Contact the Devilry developers and we will try to help you.

Node/Department admin

Note: For users managing a Node containing multiple courses.

Introduction to the Nodeadmin role

Note: Please read, or at least take a quick look at, the *Common concepts* before reading this guide.

The responibilities of a Nodeadmin

A Nodeadmin manage one or more *Node*. They typically have responsibility for an entire department or organizational unit, where a node represents a department/organizational unit.

Finding the Nodeadmin frontpage

The Nodeadmin frontpage is the page that you navigate to when you select the *Node administrator* role (may be something like *Department administrator* in your local dialect), from the Devilry frontpage. The button is *orange*.

Find students, subjects, etc

We recommend that you use search to find items in Devilry a a Nodeadmin. You can find the search-panel in the header on the left hand side of your name in the upper right corner of any page.

View/browse students that qualify for final exams

To view/browse students that qualify for final exams, grouped by their subject/course, you need to:

1. *Finding the Nodeadmin frontpage*.
2. Navigate to a Node containing subjects/courses.
3. Select the *Qualifed for final exams*-link in the *Tools* section.

Please read

The guide for subject/course admins, to know how they interract with the *Qualified for final exams* system.

Devilry sysadmin docs

Getting started

Install required system packages

1. Python 2.7.X. Check your current version by running `python --version`.
2. PIP
3. VirtualEnv
4. PostgreSQL server. Alternatively, you can test out Devilry with SQLite, but you will need PostgreSQL for production.

Create a system user for Devilry

You should run Devilry as a non-privledged user. We suggest you name the user something like `devilryrunner`. **Run all commands in this documentation as this user unless stated otherwise**.

Make a directory for your Devilry deploy

You need a directory for your Devilry settings and other Devilry-related files. We suggest you use the `~/devilrydeploy/` directory (in the HOME folder of the `devilryrunner`-user):

```
$ mkdir ~/devilrydeploy
```

The rest of the guide will assume you use the `~/devilrydeploy`-directory

Make a requirements file for Python packages

To run Devilry in production, you need the Devilry library, and a couple of extra Python packages and perhaps you will want to install some third party devilry addons. We could just install these, but that would be messy to maintain. Instead, we use a PIP requirements-file. Create `~/devilrydeploy/requirements.txt` with the following contents:

```
# PostgreSQL python bindings
psycopg2

# Elastic search python bindings
elasticsearch

# Supervisord process manager
supervisor

# The devilry library/djangoproject
# - See http://devilry.org for the latest devilry version
devilry==VERSION
```

Where `VERSION` should be set to the latest version of Devilry.

Install from the requirements file

```
$ cd ~/devilrydeploy
$ virtualenv venv
$ venv/bin/pip install -r requirements.txt
```

Create a Django management script

Copy this script into `~/devilrydeploy/manage.py`:

```
import os
import sys

if __name__ == "__main__":
    os.environ["DJANGO_SETTINGS_MODULE"] = "devilry_settings"
    from django.core.management import execute_from_command_line
    execute_from_command_line(sys.argv)
```

Configure

Devilry is configured through a python file. We will start by configuring the essential parts to get a working Devilry server, and then move on to guides for the more complex parts like search and authentication in separate chapters.

Start by copying the following into `~/devilrydeploy/devilry_settings.py`:

```
from devilry.project.production.settings import *
import dj_database_url

# Make this 50 chars and RANDOM - do not share it with anyone
SECRET_KEY = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'

# Database config
DATABASE_URL = 'sqlite:///devilrydb.sqlite'
DATABASES = {'default': dj_database_url.config(default=DATABASE_URL)}

# Set this to False to turn of debug mode in production
DEBUG = True
TEMPLATE_DEBUG = DEBUG
```

```
#: Default from email - students receive emails from this address when they make deliveries
DEVILRY_EMAIL_DEFAULT_FROM = 'devilry-support@example.com'

#: The URL that is used to link back to devilry from emails
DEVILRY_SCHEME_AND_DOMAIN = 'https://devilry.example.com'

#: Where should Devilry store files delivered by students.
#: This directory should be backed up.
DEVILRY_FSHIERDELIVERYSTORE_ROOT = '/path/to/directory/for/deliveryfiles/'

#: The directory where user uploaded files such as attachments to feedback is uploaded.
#: This directory should be backed up.
MEDIA_ROOT = '/path/to/directory/for/uploadedfiles/'

#: Url where users are directed when they do not have the permissions they believe they should have.
DEVILRY_LACKING_PERMISSIONS_URL = None

#: Url where users are directed when they want to know what to do if their personal info in Devilry
DEVILRY_WRONG_USERINFO_URL = None

#: Url where users can go to get documentation for Devilry that your organization provides.
#: If you leave this blank, the only help link will be the official Devilry documentation.
DEVILRY_ORGANIZATION_SPECIFIC_DOCUMENTATION_URL = None

#: Text for the DEVILRY_ORGANIZATION_SPECIFIC_DOCUMENTATION_URL link.
#: Leave this blank to use the default text
DEVILRY_ORGANIZATION_SPECIFIC_DOCUMENTATION_TEXT = None

#: Deadline handling method:
#:
#:     0: Soft deadlines
#:     1: Hard deadlines
DEFAULT_DEADLINE_HANDLING_METHOD = 0

#: Configure an email backend
EMAIL_BACKEND = 'djcelery_email.backends.CeleryEmailBackend'
CELERY_EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
INSTALLED_APPS += ['djcelery_email']
EMAIL_HOST_USER = ''
EMAIL_HOST_PASSWORD = ''
EMAIL_PORT = 25
EMAIL_USE_TLS = False
```

If you have a `devilry_prod_settings.py` file from an older version of Devilry, you should be able to copy over most of these settings.

Make sure it works

Just to make sure everything works, run:

```
$ cd ~/devilrydeploy/
$ venv/bin/python manage.py syncdb --noinput
$ venv/bin/python manage.py migrate --noinput
```

This should create a file named `~/devilrydeploy/devilrydb.sqlite`. You can remove that file now - it was just for testing.

Configure a SECRET_KEY

Configure the SECRET_KEY (used for cryptographic signing) by editing the `SECRET_KEY` setting in your `devilry_settings.py` script. Make it a 50 characters long random string.

Configure the database

Configure a Postgres database by editing the `DATABASE_URL` setting in your `devilry_settings.py` script. The format is:

```
DATABASE_URL = "postgres://USER:PASSWORD@HOST:PORT/NAME"
```

Note: If you are just testing out Devilry, you can keep SQLite as the database.

Configure where to store files

Adjust the `DEVILRY_FSHIERDELIVERYSTORE_ROOT` setting to a directory where you want delivered files to be stored, and the `MEDIA_ROOT` setting to a directory where you want to place all other uploaded files, such as files uploaded as attachments when examiners provide feedback.

Configure various external pages

Make sure you create a website that you can link to for the `DEVILRY_LACKING_PERMISSIONS_URL` and `DEVILRY_WRONG_USERINFO_URL` pages. You may also want to configure a `DEVILRY_ORGANIZATION_SPECIFIC_DOCUMENTATION_URL`, but that is not required.

Configure Email sending

You will probably have to adjust the `EMAIL_*` settings. The use of `djcelery_email.backends.CeleryEmailBackend` means that all email is sent via a background queue instead of letting email sending become a potential bottleneck. The other email settings are documented in the Django settings.

Disable debug mode

Before running Devilry in production, you **must** set `DEBUG=False` in `devilry_settings.py`.

Warning: If you do not disable DEBUG mode in production, you database credentials and SECRET_KEY will be shown to any visitor when they encounter an error.

Create or migrate a database

No matter if the current the database contains a database from a previous Devilry version, or if you are starting from an empty database, you need to run:

```
$ cd ~/devilrydeploy/
$ venv/bin/python manage.py syncdb --noinput
$ venv/bin/python manage.py migrate --noinput
```

This will create any missing database tables, and migrate any unmigrated database changes.

Collect static files

Run the following command to collect all static files (CSS, javascript, ...) for Devilry:

```
$ cd ~/devilrydeploy/
$ venv/bin/python manage.py collectstatic
```

The files are written to the `staticfiles` sub-directory (`~/devilrydeploy/staticfiles`).

Run the gunicorn server

Run:

```
$ cd ~/devilrydeploy/
$ DJANGO_SETTINGS_MODULE=devilry_settings venv/bin/gunicorn devilry.project.production.wsgi -b 0.0.0
```

You can adjust the number of worker threads in the `--workers` argument, and the port number in the `-b` argument.

Note: This is not how you should run this in production. Below, you will learn how to setup SSL via a webserver proxy, and Supervisord for process management.

If you do not have an existing database — Add some data

If you do not have a Devilry database from a previous version of Devilry, you will want to add some data.

First, create a superuser:

```
$ cd ~/devilrydeploy/
$ venv/bin/python manage.py createsuperuser
```

Next:

- Go to http://localhost:8000/
- Login with your newly created superuser.
- Select the *Superuser* role.
- Add a **Node**. The toplevel node is typically the name of your school/university.
- Add a **Course** within the created node. Make sure you make yourself admin on the course.
- Go back to http://localhost:8000/. You should now have a new *Course manager* role available on the frontpage.

If you have an existing database

If you already have a working Devilry database, you will most likely have to configure and authentication backend before you can do any more testing (explained below).

Stop the gunicorn server

When you are done testing, stop the gunicorn server (with `ctrl-c`), and move on to setting up the more complex parts of the system.

Whats next?

You now have a working Devilry server, but you still need to:

- *Setup a Devilry authentication backend*.
- *Install and configure the ElasticSearch search server*.
- *Setup the Celery background task server*.
- *Setup Supervisord for process management, log handling and log rotation*.
- *Setup Nginx, Apache or some other web proxy server with SSL*.

Install and configure the ElasticSearch search server

Install ElasticSearch

See http://www.elasticsearch.org/.

Configure ElasticSearch as the Devilry search backend

Add the following to `~/devilrydeploy/devilry_settings.py`:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',
        'URL': 'http://127.0.0.1:9200/',
        'INDEX_NAME': 'devilry',
    },
}
```

Adjust the URL if you are running ElasticSearch on a separate server or another port.

Build the search index

To index the data currently in the database, run:

```
$ cd ~/devilrydeploy/
$ venv/bin/python manage.py rebuild_index --noinput
```

Setup a Devilry authentication backend

Choices

Devilry can work with any Django-compatible authentication backend.

The default authentication backend

If you do not have a user database that you wish to use for Devilry, you can use the default Django authentication backend, and add users to Devilry manually.

Authenticate using LDAP

Authenticating via LDAP requires the `django-auth-ldap` Python module and some small adjustments to your settings.

Install the django-auth-ldap module

Add a new line containing `django-auth-ldap` in your `~/devilrydeploy/requirements.txt`, then run:

```
$ cd ~/devilrydeploy
$ venv/bin/pip install -r requirements.txt
```

to install the new module.

Add the LDAP backend to your settings

Add the following to your `~/devilrydeploy/devilry_settings.py`:

```
AUTHENTICATION_BACKENDS = (
    'django_auth_ldap.backend.LDAPBackend',
)
```

You will also have to configure how to authenticate via LDAP. That is explained in the django-auth-ldap docs: https://pythonhosted.org/django-auth-ldap/authentication.html

Autoset email

If your authentication backend does not provide an email address for your users, you will most likely want to take a look at: *Autoset email from the authentication backend username*.

Setup the Celery background task server

If you want to scale Devilry to more than a couple of hundred users, you really have to configure the Celery background task server. Celery is installed by default, but you need to configure a task broker. We recommend Redis.

Install Redis

See https://redis.io/.

Configure Redis

Uncomment the requirepass setting in redis.conf to set a password. Remember to run Redis with this config:

```
$ redis-server /path/to/redis.conf
```

You can tweak other configuration parameters in this file, such as port and other things, so check it out.

Add Redis and Celery settings to Devilry

Add the following to `~/devilrydeploy/devilry_settings.py` (change `secret` to match the password in the redis.conf file) and set the correct config parameters in REDIS_CONFIG:

```
REDIS_CONFIG = {
    'port': 6379,
    'hostname': 'localhost',
    'password': 'secret',
    'db_number': 0
}

BROKER_URL = 'redis://:{password}@{hostname}:{port}/{db_number}'.format(
    password='secret',
    hostname='localhost',
    port=6379,
    db_number=0
)

CELERY_RESULT_BACKEND = 'redis://:{password}@{hostname}:{port}/{db_number}'.format(
    password='secret',
    hostname='localhost',
    port=6379,
    db_number=0
)
```

Run Celery

To run Celery, use:

```
$ cd ~/devilrydeploy/
$ DJANGO_SETTINGS_MODULE=devilry_settings venv/bin/celery -A devilry.project.common worker -l debug
```

If this starts without any errors, Celery should be working. You can stop the server using `ctrl-c`. For all other cases than debugging and testing, we will be running the Celery server via Supervisord (see *Setup Supervisord for process management, log handling and log rotation*).

Setup Supervisord for process management, log handling and log rotation

Note: This assumes the full path to your `~/devilrydeploy`-directory is `/home/devilryrunner/devilrydeploy` — adjust accordingly.

Create a Supervisord configuration file

Create a file named `~/devilrydeploy/supervisord.conf` and add the following:

```
[supervisord]
childlogdir = /home/devilryrunner/devilrydeploy/log
logfile = /home/devilryrunner/devilrydeploy/log/supervisord.log
logfile_maxbytes = 50MB
logfile_backups = 30
loglevel = info
pidfile = /home/devilryrunner/devilrydeploy/var/supervisord.pid
umask = 022
nodaemon = false
nocleanup = false

[inet_http_server]
port = 9001
username = devilryadmin
password = secret

[supervisorctl]
serverurl = http://localhost:9001
username = devilryadmin
password = secret

[rpcinterface:supervisor]
supervisor.rpcinterface_factory=supervisor.rpcinterface:make_main_rpcinterface

[program:gunicorn]
command = /home/devilryrunner/devilrydeploy/venv/bin/gunicorn devilry.project.production.wsgi -b 127
environment = DJANGO_SETTINGS_MODULE=devilry_settings
process_name = gunicorn
directory = /home/devilryrunner/devilrydeploy
redirect_stderr = true
stdout_logfile = /home/devilryrunner/devilrydeploy/log/gunicorn.log
stdout_logfile_maxbytes = 150MB
stdout_logfile_backups = 15

[program:celery]
command = /home/devilryrunner/devilrydeploy/venv/bin/celery -A devilry.project.common worker -l info
environment = DJANGO_SETTINGS_MODULE=devilry_settings
process_name = celery
directory = /home/devilryrunner/devilrydeploy
redirect_stderr = true
stdout_logfile = /home/devilryrunner/devilrydeploy/log/celery.log
stdout_logfile_maxbytes = 150MB
stdout_logfile_backups = 15
```

Password and security

Make sure you set some other password than `secret` in the `[inet_http_server]` and `[supervisorctl]` sections, and make sure `~/devilrydeploy/supervisord.conf` is only accessible to the `devilryrunner`-user.

Create the var/ and log/ directories

The supervisord.conf file refers to the `~/devilrydeploy/var/` and `~/devilrydeploy/log/` directories. These must be created:

```
$ cd ~/devilrydeploy
$ mkdir var/ log/
```

Make sure all services work as excpected

To run supervisord in the foreground (for testing), run:

```
$ cd ~/devilrydeploy
$ venv/bin/supervisord -n -c supervisord.conf
```

You should now be able to open http://localhost:8002 in a browser and use Devilry. Use `ctrl-c` to kill supervisord and all the services it is running.

Run Supervisord for production

To run supervisord in the background with a PID, run:

```
$ cd ~/devilrydeploy
$ venv/bin/supervisord -c supervisord.conf
```

Warning: Do NOT run supervisord as root. Run it as the `devilryrunner` user.

Init script

The following init script works well. You need to adjust the `DAEMON`-variable:

```
#! /bin/sh
### BEGIN INIT INFO
# Provides:          supervisord
# Required-Start:    $remote_fs
# Required-Stop:     $remote_fs
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Example initscript
# Description:       This file should be used to construct scripts to be
#                    placed in /etc/init.d.
### END INIT INFO

# Author: Dan MacKinlay <danielm@phm.gov.au>
# Based on instructions by Bertrand Mathieu
# http://zebert.blogspot.com/2009/05/installing-django-solr-varnish-and.html
# See: https://gist.github.com/176149

# Do NOT "set -e"

# PATH should only include /usr/* if it runs after the mountnfs.sh script
PATH=/sbin:/usr/sbin:/bin:/usr/bin
DESC="Description of the service"
NAME=supervisord
DAEMON=/usr/local/bin/supervisord
DAEMON_ARGS=""
PIDFILE=/var/run/$NAME.pid
```

```
SCRIPTNAME=/etc/init.d/$NAME

# Exit if the package is not installed
[ -x "$DAEMON" ] || exit 0

# Read configuration variable file if it is present
[ -r /etc/default/$NAME ] && . /etc/default/$NAME

# Load the VERBOSE setting and other rcS variables
. /lib/init/vars.sh

# Define LSB log_* functions.
# Depend on lsb-base (>= 3.0-6) to ensure that this file is present.
. /lib/lsb/init-functions

#
# Function that starts the daemon/service
#
do_start()
{
        # Return
        #   0 if daemon has been started
        #   1 if daemon was already running
        #   2 if daemon could not be started
        start-stop-daemon --start --quiet --pidfile $PIDFILE --exec $DAEMON --test > /dev/null \
                || return 1
        start-stop-daemon --start --quiet --pidfile $PIDFILE --exec $DAEMON -- \
                $DAEMON_ARGS \
                || return 2
        # Add code here, if necessary, that waits for the process to be ready
        # to handle requests from services started subsequently which depend
        # on this one.  As a last resort, sleep for some time.
}

#
# Function that stops the daemon/service
#
do_stop()
{
        # Return
        #   0 if daemon has been stopped
        #   1 if daemon was already stopped
        #   2 if daemon could not be stopped
        #   other if a failure occurred
        start-stop-daemon --stop --quiet --retry=TERM/30/KILL/5 --pidfile $PIDFILE --name $NAME
        RETVAL="$?"
        [ "$RETVAL" = 2 ] && return 2
        # Wait for children to finish too if this is a daemon that forks
        # and if the daemon is only ever run from this initscript.
        # If the above conditions are not satisfied then add some other code
        # that waits for the process to drop all resources that could be
        # needed by services started subsequently.  A last resort is to
        # sleep for some time.
        start-stop-daemon --stop --quiet --oknodo --retry=0/30/KILL/5 --exec $DAEMON
        [ "$?" = 2 ] && return 2
        # Many daemons don't delete their pidfiles when they exit.
        rm -f $PIDFILE
        return "$RETVAL"
}

#
# Function that sends a SIGHUP to the daemon/service
#
do_reload() {
        #
        # If the daemon can reload its configuration without
        # restarting (for example, when it is sent a SIGHUP),
        # then implement that here.
        #
        start-stop-daemon --stop --signal 1 --quiet --pidfile $PIDFILE --name $NAME
        return 0
}

case "$1" in
  start)
        [ "$VERBOSE" != no ] && log_daemon_msg "Starting $DESC" "$NAME"
        do_start
        case "$?" in
                0|1) [ "$VERBOSE" != no ] && log_end_msg 0 ;;
                2) [ "$VERBOSE" != no ] && log_end_msg 1 ;;
        esac
        ;;
  stop)
        [ "$VERBOSE" != no ] && log_daemon_msg "Stopping $DESC" "$NAME"
        do_stop
        case "$?" in
                0|1) [ "$VERBOSE" != no ] && log_end_msg 0 ;;
                2) [ "$VERBOSE" != no ] && log_end_msg 1 ;;
        esac
        ;;
  #reload|force-reload)
        #
        # If do_reload() is not implemented then leave this commented out
        # and leave 'force-reload' as an alias for 'restart'.
        #
        #log_daemon_msg "Reloading $DESC" "$NAME"
        #do_reload
        #log_end_msg $?
        #;;
  restart|force-reload)
        #
        # If the "reload" option is implemented then remove the
        # 'force-reload' alias
        #
        log_daemon_msg "Restarting $DESC" "$NAME"
        do_stop
        case "$?" in
          0|1)
                do_start
                case "$?" in
                        0) log_end_msg 0 ;;
                        1) log_end_msg 1 ;; # Old process is still running
                        *) log_end_msg 1 ;; # Failed to start
                esac
                ;;
          *)
```

```
                # Failed to stop
                log_end_msg 1
                ;;
        esac
        ;;
  *)
        #echo "Usage: $SCRIPTNAME {start|stop|restart|reload|force-reload}" >&2
        echo "Usage: $SCRIPTNAME {start|stop|restart|force-reload}" >&2
        exit 3
        ;;
esac

:
```

Setup Nginx, Apache or some other web proxy server with SSL

You need to configure your webserver to act as a reverse proxy that forwards all traffic from port 443 (the https port) to `127.0.0.0:8002`.

The webserver must use SSL, and it should redirect traffic from port 80 to port 443.

Refer to the Gunicorn documentation for more information.

Debug problems

To test that everything works as expected, you can use the Django devserver in DEBUG-mode. The devserver serves static files, so you do not need a webserver. It does not use SSL, so be VERY careful when running it on an extrnal NIC (like the example with `0.0.0.0` below).

First, enable debug-mode in your `~/devilrydeploy/devilry_settings.py`:

```
DEBUG = True
```

Then run the devserver:

```
$ venv/bin/python manange.py runserver
```

and open http://localhost:8000. You can tell the testserver to allow external connections, and to listen on another port with:

```
$ venv/bin/python manange.py runserver 0.0.0.0:9000 --insecure
```

Warning: NEVER use the devserver or `DEBUG=True` in production. It is insecure and slow.

Note: Some browsers have issues with loading the Devilry javascript sources from the devserver. We recommend that you use a recent version of Chrome, Firefox or Safari if you have problems.

Update Devilry

Warning: These are general instructions that work if we only have code changes. Refer to the migration guide for each new version for the correct instructions.

Note: Remember that you should run all these commands as the system user you created in *Getting started*. The exception is, of course, stopping/starting Supervisord if you use an init script.

1. Update the version of the `devilry` library in your `~/devilrydeploy/requirements.txt`.
2. Stop Supervisord.
3. Update Devilry using PIP:

   ```
   $ cd ~/devilrydeploy
   $ venv/bin/pip install -r requirements.txt
   ```

4. Start Supervisord.

Autoset email from the authentication backend username

If you have an authentication backend that uses username, and it does not set an email for your users, you will probably want to add the `devilry.devilry_autoset_empty_email_by_username` app to automatically set an email based on usernames.

To enable this app, add the following to your `~/devilrydeploy/devilry_settings.py`:

```
INSTALLED_APPS += ['devilry.apps.autoset_empty_email_by_username']

#: Email pattern. The 'devilry.devilry_autoset_empty_email_by_username' app
#: automatically sets email to "<username>@DEVILRY_DEFAULT_EMAIL_SUFFIX"
#: when a user is saved.
DEVILRY_DEFAULT_EMAIL_SUFFIX = 'example.com'
```

Devilry Managment Commands

This section describe the managment commands available in devilry. To learn more about Django and their administrative support visit the django managment commands page in their docs. The set of commands may be altered or extended by packages used in Devilry.

Django managment commands follow a strict and well defined interface and is easy to extend and customize. More info can be found on the custom django-admin commands page. Devilry provides the following commands to ease the administration tasks for Devilry maintainers. If you find the list incomplete and or want a broader support, you are welcome to post an issue on the Devilry project issue tracker at any time.

The source code of the commands can be found in the official Devilry repository in the superadmin managment commands directory.

devilry_nodeadd

django.py devilry_nodeadd <node path> <short name>

Creates a new node in the Devilry node hierarchy. To create a root node use *None* as <node path>.

--admins	Comma separated list of usernames to set as admins on the node
--long_name	A longer and more descriptive name of the node.

devilry_subjectadd

django.py devilry_subjectadd <node path> <short name>

Creates a new subject within the devilry hierarchy. The path and short name are required.

--admins	Comma separated list of usernames to set as admins on the node
--long_name	A longer and more descriptive name of the node.

devilry_subjectadminadd

django.py devilry_subjectadminadd <subject-short_name> <admin username>

Add a user as admin on the specified subject.

devilry_subjectadminclear

django.py devilry_subjectadminclear <subject short name>

Removes all administrators from the specified subject.

devilry_subjectsearch

django.py devilry_subjectsearch <short name>

Search for a subject by short name. Matches any part of the name.

--short_name-only	Only print short name (one line per short name)

devilry_periodadd

django.py devilry_periodadd <subject short name> <period short name>

Create a new period on a new subject.

--admins	Comma separated list of usernames to set as admins on the node.
--long_name	A longer and more descriptive name of the node.
--start-time	The start time of the period on ISO format *"%Y-%m-%dT%H:%M"*.
--end-time	The end time of the period on ISO format *"%Y-%m-%dT%H:%M"*.
--date-format	The date format expressed in a format according to strftime

devilry_periodadminadd

django.py devilry_periodadminadd <subject-short_name> <period-short-name> <admin-username>

Add a user as admin on the period.

devilry_periodadminclear

django.py devilry_periodadminclear <subject short name> <period short name>

Clear administrators on the the subject.

devilry_periodsearch

django.py devilry_periodsearch <period short name>

Searches for periods based on the specified short name

--short_name-only	Only print short name (one line per short name)

devilry_periodsetrelatedexaminers

django.py devilry_periodsetrelatedexaminers <subject short name> <period short name>

Set related examiners on a period. Users are read from stdin, as a JSON encoded array of arguments to the RelatedExaminer model. See relatedexaminers.json for an example.

--clearall	Clear all related examiners before adding

devilry_periodsetrelatedstudents

django.py devilry_periodsetrelatedstudents <subject short name> <period short name>

Set related students on a period. Users are read from stdin, as a JSON encoded array of arguments to the RelatedStudent model. See relatedstudents.json for an example.

--clearall	Clear all related students before adding

devilry_resave_all_users

django.py devilrly_resave_all_users

Resaves all users. This command is useful if you have any apps that listens for *post_save* signals on **User**.

devilry_sync_candidates

django.py devilry_sync_candidates

Sync the cached fields in Candidate with the actual data from User.

devilry_useradd

django.py devilry_userad <username>

Creates a new user.

--email	The user email address
--full_name	Full name of the user
--superuser	Make the user a superuser. Be careful this will give the user complete access to everything in Devilry.
--password	Password for the user login credential.

Returns a non-zero value when the user already exists in Devilry.

devilry_useraddbulk

django.py devilry_useraddbulk

Reading usernames from *stdin*

--emailsuffix	Email suffix are set on all users in the list. Example: <username>@example.com

devilry_usermod

django.py devilry_usermod <username>

Modify the credentials of an existing user

--email	The user email address
--full_name	Full name of the user
--superuser	Make the user a superuser. Be careful this will give the user complete access to everything in Devilry.
--normaluser	Make the user a normal user, with access to everything within their rank in Devilry hierarchy

devilry_usersearch

django.py devilry_usersearch <username>

Search for a user by username. Matches any part of the username.

--username-only	Only print usernames
--no-email	Only matching users without an email address.
--superusers	Only matching superusers
--normalusers	Only matching normalusers, everybody except superusers

Migration guides

If a minor version is not listed here, it is a code-only update, which means that the *update guide* is all you need.

Migrating from 2.0.1 to 2.0.3

Note: We skipped 2.0.2 because of a forgotten update to version.json.

Backup database and files

BACKUP. YOUR. DATABASE. AND. FILES.

Update devilry

Follow the *Update* guide, and set the version in requirements.txt to:

```
devilry==2.0.3
```

Migrating from 2.0.3 to 2.0.4

Backup database and files

BACKUP. YOUR. DATABASE. AND. FILES.

Update devilry

Follow the *Update* guide, and set the version in requirements.txt to:

```
devilry==2.0.4
```

Migrating from 1.X.X to 2.0.0

Backup database and files

BACKUP. YOUR. DATABASE. AND. FILES.

Update devilry

This is the first release using the new deployment/production setup. Please follow the *Getting started* guide.

Devilry developer documentation

Note: Welcome to the Devilry developer documentation. See http://devilry.org/ for general information about Devilry, and https://github.com/devilry/devilry-django for the code.

Essentials

Setup a local development environment

Check out from GIT

If you plan to develop devilry, you should fork the devilry-django repo, changes to your own repo and request inclusion to the master repo using github pull requests. If you are just trying out Devilry, use:

```
$ git clone https://github.com/devilry/devilry-django.git
```

The `master` branch, which git checks out by default, is usually the latest semi-stable development version. The latest stable version is in the `latest-stable` branch.

Install dependencies/requirements

Note: Devilry should work perfectly well with only Python 2.7 or later Python2 versions. Devilry does not work with Python3 yet, but we will support it when Django and all our dependencies gets good Python3 support.

Other dependencies than are not really required, but we recommend that you:

- use Virtualenv to avoid installing anything globally, and to get a clean environment
- use Fabric because we have a lot of useful scripts written for Fabric that will ease setting up your development environment and building various components of Devilry. See *Fabric*.

Note that all instructions below assume you have and want to install Fabric and Virtualenv.

Mac OSX

1. Install **XCode** (from app store).
2. Install command line tools for XCode (includes Git and Python):

   ```
   $ xcode-select --install
   ```

3. Install other dependencies/requirements:

   ```
   $ sudo easy_install virtualenv
   ```

Ubuntu Linux

```
$ sudo apt-get install build-essential python-dev python-virtualenv libncurses5-dev virtualenvwrappe
```

Setup the development virtualenv

```
$ mkvirtualenv devilry-django
$ pip install -r requirements/development.txt
```

Create a database

We have several alternatives for setting up a demo database. They all use Fabric tasks. See *Fabric*.

First, make sure you are in the `devilry-django` virtualenv:

```
$ workon devilry-django
```

You can create a fairly full featured demo database with:

```
$ fab autodb
```

... or you can create a much more minimalistic demo database with:

```
$ fab demodb
```

... or you can create an empty database with:

```
$ fab reset_db
```

Note: Creating the testdata with `autodb` takes a lot of time, but you can start using the server as soon as the users have been created (one of the first things the script does).

Run the Django development server

First, make sure you are in the `devilry-django` virtualenv:

```
$ workon devilry-django
```

Start the Django development server with:

```
$ python manage.py runserver
```

Go to http://localhost:8000/ and log in as a superuser using:

```
user: grandma
password: test
```

Or as a user which is student, examiner and admin using:

```
user: thor
password: test
```

Note: All users have `password==test`, and you can see all users in the superadmin interface. See the demo page on the wiki for more info about the demo database, including recommended test users for each role.

Fabric

We use Fabric to simplify common tasks. Fabric simply runs the requested `@task` decorated functions in `fabfile.py`.

`fabfile.py` is very straigt forward to read if you wonder what the tasks actually do. The `fabric.api.local(...)` function runs an executable on the local machine.

The devilry testsuite

Run **all** test:

```
$ DJANGOENV=test python manage.py test devilry
```

Skip the *selenium* tests using:

```
$ SKIP_SELENIUMTESTS=1 DJANGOENV=test python manage.py test
```

Specify a browser for the selenium tests using (example uses Firefox):

```
$ SELENIUM_BROWSER=Firefox DJANGOENV=test python manage.py test
```

Chrome is the default browser (configured in `devilry.project.develop.settings.base`).

Note: We use `DJANGOENV=test python manage.py` to run tests, because that makes `manage.py` use `devilry.project.develop.settings.test`, which does not load Haystack or Celery.

Mocking tests

Always try to mock objects instead of creating real data unless you are actually testing something that needs real data. Use https://pypi.python.org/pypi/mock to mock your tests.

corebuilder — Setup devilry core data structures for tests

`devilry.project.develop.testhelpers.corebuilder` is a module that makes it easy to create `devilry.apps.core.models` data for tests.

When to use

Use this for end-to-end tests and tests where you really need real data. Always try to mock objects instead of creating real data unless you are actually testing something that needs real data. See *Mocking tests*.

Howto

Each class in the core has a wrapper class in `devilry.project.develop.testhelpers.corebuilder` that makes it easy to perform operations that we need to setup tests. We call these wrappers *builders*, and they are all prefixed with the name of their corresponding core model and suffixed with `Builder`.

Using the builders is very easy:

```
from devilry.project.develop.testhelpers.corebuilder import NodeBuilder
duck1010builder = NodeBuilder('duckuniversity').add_subject('duck1010')
assert(duck1010builder.subject == Subject.objects.get(short_name='duck1010'))
```

They can all easily be updated with new attributes:

```
duck1010builder.update(long_name='DUCK1010 - Programming')
assert(duck1010builder.subject.long_name == 'DUCK1010 - Programming')
```

And they have sane defaults optimized for testing, so you can easily create a deeply nested core object. This creates the duck1010-subject with an active period that started 3 months ago and ends in 3 months, with a single assignment (week1), with a single group, with deadline one week from now with a single `helloworld.txt` delivery:

```
from devilry.project.develop.testhelpers.corebuilder import NodeBuilder
from devilry.project.develop.testhelpers.corebuilder import UserBuilder
peterpan = UserBuilder(username='peterpan')
helloworld_filemetabuilder = NodeBuilder('ducku')\
    .add_subject('duck1010')\
    .add_6month_active_period('current')\
    .add_assignment('week1')\
    .add_group(students=[peterpan.user])\
    .add_deadline_in_x_weeks(weeks=1)\
    .add_delivery()\
    .add_filemeta(filename='helloworld.txt', data='Hello world')
```

Since we often need to add a single subject or a single active period, we have shortcuts for that:

```
from devilry.project.develop.testhelpers.corebuilder import SubjectBuilder
from devilry.project.develop.testhelpers.corebuilder import PeriodBuilder
duck1010_builder = SubjectBuilder.quickadd_ducku_duck1010()
currentperiod_builder = PeriodBuilder.quickadd_ducku_duck1010_active()
```

Note: These shortcuts is not there just to save a couple of keystrokes. They are there to make sure we use a uniform test setup in 98% of our tests. As long as you just need a single subject or period, you MUST use these shortcuts (to get a patch accepted in Devilry).

Magic and defaults

The builders have very little magic, but they have some defaults that make sense when testing:

- `long_name` is set to `short_name` when it is not specified explicitly.
- All BaseNodes (the models with short and long name) takes the `short_name` as the first argument and the `long_name` as the second argument.
- Time of delivery (for `DeliveryBuilder` and `DealdineBuilder.add_delivery()`) default to *now*.
- Default `publishing_time` for assignments is *now*.
- UserBuilder defaults to setting email to `<username>@example.com`.

These defaults are all handled in the constructor of their builder-class. All the defaults can be overridden by specifying a value for them.

Reload from DB

You often need to create an object that is changed by the code you are testing, and then check that the change has made it to the database. All our builders implement `ReloadableDbBuilderInterface` which includes `reload_from_db()`.

ReloadableDbBuilderInterface

class `devilry.project.develop.testhelpers.corebuilder.`**`ReloadableDbBuilderInterface`**

All the builders implement this interface.

`update``(`*****attributes***`)`

Update the object wrapped by the builder with the given attributes. Saves the object, and reloads it from the database.

`reload_from_db``(`*****attributes***`)`

Reloads the object wrapped by the builder from the database. Perfect when you create an object that is changed by the code you are testing, and you want to check that the change has made it to the database.

UserBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**`UserBuilder`**

Creates a User object for testing. Also creates the DevilryUserProfile, and methods for editing both the User and the profile.

`__init__`(*username*, *full_name=None*, *email=None*)

Creates a new User with password set to test, and the `devilry.apps.core.models.DevilryUserProfile` created.

Parameters

- **username** – The username of the new user.
- **full_name** – Optional full_name. Defaults to `None`.
- **email** – Optional email. Defaults to `<username>@example.com`.

`update`(***attributes*)

Update the User with the given attributes. Reloads the object from the database.

`update_profile`(***attributes*)

Update the `devilry.apps.core.models.DevilryUserProfile` with the given attributes. Reloads the object from the database.

NodeBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**`NodeBuilder`**

`node`

The `Node` wrapped by this builder.

`__init__`(*short_name*, *long_name=None*, ***kwargs*)

Creates a new `Node` with the given attributes.

Parameters

- **short_name** – The `short_name` of the Node.
- **long_name** – The `long_name` of the Node. Defaults to `short_name` if `None`.
- **kwargs** – Other arguments for the Node constructor.

`add_node`(**args*, ***kwargs*)

Adds a childnode to the node. `args` and `kwargs` are forwarded to `NodeBuilder` with `kwargs['parentnode']` set to this `node`.

Return type `NodeBuilder`.

`add_subject`(**args*, ***kwargs*)

Adds a subject to the node. `args` and `kwargs` are forwarded to `SubjectBuilder` with `kwargs['parentnode']` set to this `node`.

Return type `SubjectBuilder`.

SubjectBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**`SubjectBuilder`**

`subject`

The `Subject` wrapped by this builder.

`__init__`(*short_name*, *long_name=None*, ***kwargs*)
Creates a new `Subject` with the given attributes.

Parameters
- **short_name** – The `short_name` of the Subject.
- **long_name** – The `long_name` of the Subject. Defaults to `short_name` if `None`.
- **kwargs** – Other arguments for the Subject constructor.

classmethod `quickadd_ducku_duck1010`()
When we need just a single subject, we use this shortcut method instead of writing:

```
NodeBuilder('ducku').add_subject('duck1010')
```

This is not just to save a couple of letters, but also to promote a common setup for simple tests.

`add_period`(**args*, ***kwargs*)
Adds a period to the subject. `args` and `kwargs` are forwarded to `PeriodBuilder` with `kwargs['parentnode']` set to this `subject`.

Return type `PeriodBuilder`.

`add_6month_active_period`(**args*, ***kwargs*)
Shortcut for adding `add_period()` with `start_time` `3*30` days ago, and `end_time` in `3*30` days. `args` and `kwargs` is forwarded to `add_period`, but with `start_time` and `end_time` set in `kwargs`.

If no `short_name` is provided, it defaults to `active`.

Return type `PeriodBuilder`.

`add_6month_lastyear_period`(**args*, ***kwargs*)
Shortcut for adding `add_period()` with `start_time` `365-30*3` days ago, and `end_time` `365+3*30` days ago. `args` and `kwargs` is forwarded to `add_period`, but with `start_time` and `end_time` set in `kwargs`.

If no `short_name` is provided, it defaults to `lastyear`. :rtype: `PeriodBuilder`.

`add_6month_nextyear_period`(**args*, ***kwargs*)
Shortcut for adding `add_period()` with `start_time` in `365-30*3` days, and `end_time` in `365+3*30` days. `args` and `kwargs` is forwarded to `add_period`, but with `start_time` and `end_time` set in `kwargs`.

If no `short_name` is provided, it defaults to `nextyear`.

Return type `PeriodBuilder`.

PeriodBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**`PeriodBuilder`**

`period`
The `Period` wrapped by this builder.

`__init__`(*short_name*, *long_name=None*, ***kwargs*)
Creates a new `Period` with the given attributes.

Parameters
- **short_name** – The `short_name` of the Period.
- **long_name** – The `long_name` of the Period. Defaults to `short_name` if `None`.
- **kwargs** – Other arguments for the Period constructor.

add_assignment(**args*, ***kwargs*)

Adds an assignment to the period. `args` and `kwargs` are forwarded to `AssignmentBuilder` with `kwargs['parentnode']` set to this `period`.

Return type `AssignmentBuilder`.

classmethod quickadd_ducku_duck1010_active()

When we need just a single active period, we use this shortcut method instead of writing:

```
NodeBuilder('ducku').add_subject('duck1010').add_6month_active_period('current')
```

This is not just to save a couple of letters, but also to promote a common setup for simple tests.

AssignmentBuilder

class devilry.project.develop.testhelpers.corebuilder.**AssignmentBuilder**

assignment

The `Assignment` wrapped by this builder.

__init__(*short_name*, *long_name=None*, ***kwargs*)

Creates a new `Assignment` with the given attributes.

Parameters

- **short_name** – The `short_name` of the Assignment.
- **long_name** – The `long_name` of the Assignment. Defaults to `short_name` if `None`.
- **publishing_time** – The `publishing_time` of the Assignment. Defaults to now.
- **kwargs** – Other arguments for the Assignment constructor.

add_group(**args*, ***kwargs*)

Adds an assignment group to the period. `args` and `kwargs` are forwarded to `AssignmentGroupBuilder` with `kwargs['parentnode']` set to this `assignment`.

Return type `AssignmentGroupBuilder`.

AssignmentGroupBuilder

class devilry.project.develop.testhelpers.corebuilder.**AssignmentGroupBuilder**

assignment_group

The `AssignmentGroup` wrapped by this builder.

__init__(*students=[ ]*, *candidates=[ ]*, *examiners=[ ]*, ***kwargs*)

Creates a new `AssignmentGroup` with the given attributes.

Parameters

- **students** – Forwarded to `add_students()`.
- **candidates** – Forwarded to `add_candidates()`.
- **examiners** – Forwarded to `add_examiners()`.
- **kwargs** – Arguments for the AssignmentGroup constructor.

add_students(**users*)

Add the given users as candidates without a candidate ID on this assignment group.

Returns `self` (to enable us to nest the method call).

add_examiners(**users*)
Add the given users as examiners on this assignment group.
Returns `self` (to enable us to nest the method call).

add_students(**candidates*)
Add the given candidates to this assignment group.
Parameters **candidates** – `devilry.apps.core.models.Candidate` objects.
Returns `self` (to enable us to nest the method call).

add_deadline(**args*, ***kwargs*)
Adds an deadline to the assignment. `args` and `kwargs` are forwarded to `DeadlineBuilder` with `kwargs['assignment_group']` set to this `assignment_group`.
Return type `AssignmentGroupBuilder`.

add_deadline_in_x_weeks(*weeks*, **args*, ***kwargs*)
Calls `add_deadline()` with `kwargs[deadline]` set `weeks` weeks in the future.
Return type `AssignmentGroupBuilder`.

add_deadline_x_weeks_ago(*weeks*, **args*, ***kwargs*)
Calls `add_deadline()` with `kwargs[deadline]` set `weeks` weeks in the past.
Return type `DeadlineBuilder`.

DeadlineBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**DeadlineBuilder**

deadline
The `Deadline` wrapped by this builder.

__init__(***kwargs*)
Creates a new `AssignmentGroup` with the given attributes.
Parameters **kwargs** – Arguments for the Deadline constructor.

add_delivery(***kwargs*)
Adds a delivery to the deadline. `args` and `kwargs` are forwarded to `DeliveryBuilder` with `kwargs['deadline']` set to this `deadline` and `kwargs['successful']` defaulting to `True`.
Parameters **kwargs** – Extra kwargs for the `DeliveryBuilder` constructor.
Return type `DeliveryBuilder`.

add_delivery_after_deadline(*timedeltaobject*, ***kwargs*)
Add a delivery `timedeltaobject` time after this deadline expires.

Shortcut that calls `add_delivery()` with `kwargs['time_of_delivery']` set to `deadline.deadline + timedeltaobject`.

Example - add delivery 3 weeks and 2 hours after deadline:

```
from datetime import datetime, timedelta
deadlinebuilder = DeadlineBuilder(deadline=datetime(2010, 1, 1))
deadlinebuilder.add_delivery_after_deadline(timedelta(weeks=3, hours=2))
```

Parameters **kwargs** – Extra kwargs for the `DeliveryBuilder` constructor.
Return type `DeliveryBuilder`.

add_delivery_before_deadline(*timedeltaobject*, ***kwargs*)
Add a delivery `timedeltaobject` time before this deadline expires.

Shortcut that calls `add_delivery()` with `kwargs['time_of_delivery']` set to `deadline.deadline + timedeltaobject`.

Example - add delivery 5 hours before deadline:

```
from datetime import datetime, timedelta
deadlinebuilder = DeadlineBuilder(deadline=datetime(2010, 1, 1))
deadlinebuilder.add_delivery_before_deadline(timedelta(hours=5))
```

Parameters **kwargs** – Extra kwargs for the `DeliveryBuilder` constructor.
Return type `DeliveryBuilder`.

`add_delivery_x_hours_after_deadline`(*timedeltaobject*, ***kwargs*)
Add a delivery `hours` hours after this deadline expires.

Shortcut that calls `add_delivery_after_deadline()` with timedeltaobject set to `timedelta(hours=hours)`.

Parameters
- **hours** – Number of hours.
- **kwargs** – Extra kwargs for the `DeliveryBuilder` constructor.

Return type `DeliveryBuilder`.

`add_delivery_x_hours_before_deadline`(*timedeltaobject*, ***kwargs*)
Add a delivery `hours` hours before this deadline expires.

Shortcut that calls `add_delivery_before_deadline()` with timedeltaobject set to `timedelta(hours=hours)`.

Parameters
- **hours** – Number of hours.
- **kwargs** – Extra kwargs for the `DeliveryBuilder` constructor.

Return type `DeliveryBuilder`.

DeliveryBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**`DeliveryBuilder`**

`delivery`
The `Delivery` wrapped by this builder.

`__init__`(***kwargs*)
Creates a new `Delivery` with the given attributes. If `time_of_delivery` is not provided, it defaults to *now*.

Parameters **kwargs** – Arguments for the Delivery constructor.

`add_filemeta`(***kwargs*)
Adds a filemeta to the delivery. `kwargs` is forwarded to `FilteMetaBuilder` with `kwargs['delivery']` set to this `delivery`.

Example:

```
deliverybuilder.add_filemeta(
    filename='test.txt',
    data='This is a test.'
)
```

Parameters **kwargs** – Kwargs for the `FileMetaBuilder` constructor.
Return type `FileMetaBuilder`.

add_feedback(***kwargs*)

Adds a feedback to the delivery. `kwargs` is forwarded to `StaticFeedbackBuilder` with `kwargs['delivery']` set to this `delivery`.

Example:

```
deliverybuilder.add_feedback(
    points=10,
    grade='10/100',
    is_passing_grade=False,
    saved_by=UserBuilder('testuser').user
)
```

Parameters **kwargs** – Kwargs for the `StaticFeedbackBuilder` constructor.

Return type `StaticFeedbackBuilder`.

add_passed_feedback(***kwargs*)

Shortcut that adds a passed feedback to the delivery. `kwargs` is forwarded to `add_feedback()` with:

- `points=1`
- `grade="Passed"`
- `is_passing_grade=True`

Example:

```
deliverybuilder.add_passed_feedback(saved_by=UserBuilder('testuser').user)
```

Parameters **kwargs** – Extra kwargs for `add_feedback()`. Is updated with :points, grade and is_passing_grade as documented above.

Return type `StaticFeedbackBuilder`.

add_failed_feedback(***kwargs*)

Shortcut that adds a failed feedback to the delivery. `kwargs` is forwarded to `add_feedback()` with:

- `points=0`
- `grade="Failed"`
- `is_passing_grade=False`

Example:

```
deliverybuilder.add_failed_feedback(saved_by=UserBuilder('testuser').user)
```

Parameters **kwargs** – Extra kwargs for `add_feedback()`. Is updated with :points, grade and is_passing_grade as documented above.

Return type `StaticFeedbackBuilder`.

FileMetaBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**FileMetaBuilder**

filemeta

The `FileMeta` wrapped by this builder.

__init__(*delivery*, *filename*, *data*)

Creates a new `FileMeta`. Since FileMeta just points to files on disk, and creating those files requires iterators and extra stuff that is almost never needed for tests, we provide an easier method for creating files with FileMetaBuilder.

Parameters

- **delivery** – The Delivery object.

- **filename** – A filename.
- **data** – The file contents as a string.

StaticFeedbackBuilder

class `devilry.project.develop.testhelpers.corebuilder.`**`StaticFeedbackBuilder`**

`feedback`
The `StaticFeedback` wrapped by this builder.

`__init__`(***kwargs*)
Creates a new `StaticFeedback` with the given attributes.
Parameters **kwargs** – Arguments for the StaticFeedback constructor.

Essential wiki pages for developers

- How to write API documentation - wiki page
- More info available on the Developer wiki page.

API and utilities

`devilry.apps.core.models` — Devilry core datastructure

(edit the images umldiagram1 and umldiagram2 using yuml.me)

Functions and attributes

`devilry.apps.core.models.model_utils.`**`pathsep`**
Path separator used by node-paths. The value is `'.'`, and it must not be changed.

`devilry.apps.core.models.model_utils.`**`splitpath`**(*path*, *expected_len=0*)
Split the path on `pathsep` and return the resulting list. Example:

```
>>> splitpath('uio.ifi.matnat')
['uio', 'ifi', 'matnat']
>>> splitpath('uio.ifi.matnat', expected_len=2)
Traceback (most recent call last):
...
ValueError: Path must have exactly 2 parts
```

Parameters **expected_len** – Expected length of the resulting list. If the resulting list is not exactly the given length, `ValueError` is raised. If `expected_len` is 0 (default), no checking is done.

BaseNode

class `devilry.apps.core.models.`**`BaseNode`**
Bases: `devilry.apps.core.models.abstract_is_admin.AbstractIsAdmin`, `devilry.apps.core.models.save_interface.SaveInterface`

The base class of the Devilry hierarchy. Implements basic functionality used by the other Node classes. This is an abstract datamodel, so it is never used directly.

short_name
A django.db.models.SlugField with max 20 characters. Only numbers, letters, '_' and '-'.

long_name
A django.db.models.CharField with max 100 characters. Gives a longer description than `short_name`.

AbstractIsAdmin

class `devilry.apps.core.models.`**`AbstractIsAdmin`**
Bases: `object`

Abstract class implemented by all classes where it is natural to need to check if a user has admin rights.

classmethod `q_is_admin`(*user_obj*)
Get a django.db.models.Q object matching all objects of this type where the given user is admin. The matched result is not guaranteed to contain unique items, so you should use distinct() on the queryset if this is required.

This must be implemented in all subclassed.

classmethod `where_is_admin`(*user_obj*, **related_fields*)
Get all objects of this type where the given user is admin.

classmethod `where_is_admin_or_superadmin`(*user_obj*, **related_fields*)
Get all objects of this type where the given user is admin, or all objects if the user is superadmin.

AbstractIsExaminer

class `devilry.apps.core.models.`**`AbstractIsExaminer`**
Bases: `object`

Abstract class implemented by all classes where it is natural to need to check if a user is examiner.

classmethod `q_published`(*old=True*, *active=True*)
Return a django.models.Q object which matches all items of this type where `Assignment.publishing_time` is in the past.

Parameters
- **old** – Include assignments where `Period.end_time` is in the past?
- **active** – Include assignments where `Period.end_time` is in the future?

classmethod `q_is_examiner`(*user_obj*)
Return a django.models.Q object which matches items where the given user is examiner.

classmethod `where_is_examiner`(*user_obj*)
Get all items of this type where the given `user_obj` is examiner on one of the assignment groups.
Parameters **user_obj** – A django.contrib.auth.models.User object.
Return type QuerySet

classmethod `published_where_is_examiner`(*user_obj*, *old=True*, *active=True*)
Get all published items of this type where the given `user_obj` is examiner on one of the assignment groups. Combines `q_is_examiner()` and `q_published()`.

Parameters
- **user_obj** – `q_is_examiner()`.
- **old** – `q_published()`.
- **active** – `q_published()`.

Returns A django.db.models.query.QuerySet with duplicate assignments eliminated.

classmethod **`active_where_is_examiner`**(*user_obj*)
Shortcut for `published_where_is_examiner()` with `old=False`.

classmethod **`old_where_is_examiner`**(*user_obj*)
Shortcut for `published_where_is_examiner()` with `active=False`.

Node

A node at the top of the navigation tree. It is a generic element used to organize administrators. A Node can be organized below another Node, and it can only have one parent.

Let us say you use Devilry within two departments at *Fantasy University*; informatics and mathematics. The university has an administration, and each department have their own administration. You would end up with this node-hierarchy:

- **Fantasy University**
  - Department of informatics
  - Department of mathematics

class `devilry.apps.core.models.`**`Node`**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`, `devilry.apps.core.models.basenode.BaseNode`, `devilry.apps.core.models.model_utils.Etag`

This class is typically used to represent a hierarchy of institutions, faculties and departments.

`parentnode`
A django.db.models.ForeignKey that points to the parent node, which is always a Node.

`admins`
A django.db.models.ManyToManyField that holds all the admins of the Node.

`child_nodes`
A set of child_nodes of type Node for this node

`subjects`
A set of `subjects` for this node

`etag`
A DateTimeField containing the etag for this object.

`iter_childnodes`()
Recursively iterates over all child nodes, and their child nodes. For a list of direct child nodes, use atribute child_nodes instead.

`clean`(**args*, ***kwargs*)
Validate the node, making sure it does not do something stupid.

Always call this before save()! Read about validation here: http://docs.djangoproject.com/en/dev/ref/models/instances/#id1

Raises ValidationError if:
- The node is it’s own parent.
- The node is the child of itself or one of its childnodes.

`is_empty`()
Returns `True` if this Node does not contain any childnodes or subjects.

Subject

A subject is a course, seminar, class or something else being given regularly. A subject is further divided into periods.

class `devilry.apps.core.models.`**`Subject`**`(`**args*, ***kwargs*`)`

Bases: `django.db.models.base.Model`, `devilry.apps.core.models.basenode.BaseNode`, `devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer`, `devilry.apps.core.models.abstract_is_candidate.AbstractIsCandidate`, `devilry.apps.core.models.model_utils.Etag`

`parentnode`
A django.db.models.ForeignKey that points to the parent node, which is always a Node.

`admins`
A django.db.models.ManyToManyField that holds all the admins of the Node.

`short_name`
A django.db.models.SlugField with max 20 characters. Only numbers, letters, '_' and '-'. Unlike all other children of `BaseNode`, Subject.short_name is **unique**. This is mainly to avoid the overhead of having to recurse all the way to the top of the node hierarchy for every unique path.

`periods`
A set of `periods` for this subject.

`etag`
A DateTimeField containing the etag for this object.

`get_path``()`
Only returns `short_name` for subject since it is guaranteed to be unique.

`is_empty``()`
Returns `True` if this Subject does not contain any periods.

Period

A Period is a limited period of time, like *spring 2009*, *week 34 2010* or even a single day.

class `devilry.apps.core.models.`**`Period`**`(`**args*, ***kwargs*`)`

Bases: `django.db.models.base.Model`, `devilry.apps.core.models.basenode.BaseNode`, `devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer`, `devilry.apps.core.models.abstract_is_candidate.AbstractIsCandidate`, `devilry.apps.core.models.model_utils.Etag`

A Period represents a period of time, for example a half-year term at a university.

`parentnode`
A django.db.models.ForeignKey that points to the parent node, which is always a Subject.

`start_time`
A django.db.models.DateTimeField representing the starting time of the period.

`end_time`
A django.db.models.DateTimeField representing the ending time of the period.

`admins`
A django.db.models.ManyToManyField that holds all the admins of the node.

`assignments`
A Django RelatedManager of `assignments` for this period.

relatedexaminer_set
A Django RelatedManager of `RelatedExaminers` for this period.

relatedstudent_set
A Django RelatedManager of `RelatedStudents` for this period.

etag
A DateTimeField containing the etag for this object.

clean(**args*, ***kwargs*)
Validate the period.

Always call this before save()! Read about validation here: http://docs.djangoproject.com/en/dev/ref/models/instances/#id1

Raises ValidationError if start_time is after end_time.

is_active()
Returns true if the period is active

classmethod **q_is_active**()
Get a `django.db.models.Q` object that matches all active periods (periods where start_time is in the past, and end_time is in the future).

Example:

```
activeperiods = Period.objects.filter(Period.q_is_active())
```

is_empty()
Returns `True` if this Period does not contain any assignments.

subject
More readable alternative to `self.parentnode`.

RelatedUserBase

Base class for `devilry.apps.core.models.RelatedStudent` and `devilry.apps.core.models.RelatedExamine`

class `devilry.apps.core.models.relateduser.`**RelatedUserBase**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`, `devilry.apps.core.models.abstract_is_admin.AbstractIsA`

Common fields for examiners and students related to a period.

period
The period that the user is related to.

user
A django.contrib.auth.models.User object. Must be unique within this period.

tags
Comma-separated list of tags. Each tag is a word with the following letters allowed: a-z and 0-9. Each word is separated by a comma, and no whitespace.

RelatedStudent — Student on a period

A RelatedStudent is a student *related* to a `devilry.apps.core.models.Period`.

class `devilry.apps.core.models.`**`RelatedStudent`**(**args*, ***kwargs*)

Bases: `devilry.apps.core.models.relateduser.RelatedUserBase`

Related student.

`candidate_id`

If a candidate has the same Candidate ID for all or many assignments in a semester, this field can be set to simplify setting candidate IDs on each assignment.

RelatedExaminer — Examiner on a period

A RelatedExaminer is an examiner *related* to a `devilry.apps.core.models.Period`.

class `devilry.apps.core.models.`**`RelatedExaminer`**(**args*, ***kwargs*)

Bases: `devilry.apps.core.models.relateduser.RelatedUserBase`

Related examiner.

Adds no fields to RelatedUserBase.

Assignment

Represents one assignment within a given Period in a given Subject. Each assignment contains one Assignment-Group for each student or group of students permitted to submit deliveries. We have three main classifications of assignments:

1. A *old assignment* is a assignment where `Period.end_time` is in the past.
2. A *published assignment* is a assignment where `publishing_time` is in the past.
3. A *active assignment* is a assignment where `publishing_time` is in the past and current time is before `Period.end_time`.

class `devilry.apps.core.models.`**`Assignment`**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`, `devilry.apps.core.models.basenode.BaseNode`, `devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer`, `devilry.apps.core.models.abstract_is_candidate.AbstractIsCandidate`

`parentnode`

A django.db.models.ForeignKey that points to the parent node, which is always a Period.

`publishing_time`

A django.db.models.DateTimeField representing the publishing time of the assignment.

`anonymous`

A models.BooleanField specifying if the assignment should be anonymously for correcters.

`admins`

A django.db.models.ManyToManyField that holds all the admins of the Node.

`assignmentgroups`

A set of `assignmentgroups` for this assignment

`examiners_publish_feedbacks_directly`

Should feedbacks published by examiners be made avalable to the students immediately? If not, an administrator have to publish feedbacks. See also `Deadline.feedbacks_published`.

`scale_points_percent`

Percent to scale points on this assignment by for period overviews. The default is 100, which means no change to the points.

delivery_types

An integer identifying the type of deliveries allowed. Possible values:

0 Electronic deliveries using Devilry

1 Non-electronic deliveries, or deliveries made through another electronic system.

2 An alias/link to a delivery made in another Period.

deadline_handling

An integer identifying how deadlines are handled.

0 Soft deadlines. Deliveries can be added until groups are closed.

1 Hard deadlines. Deliveries can not be added after the deadline has expired.

first_deadline

A DateTimeField containing an optional first deadline for this assignment. This is metadata that the UI can use where it is natural.

max_points

An IntegerField that contains the maximum number of points possible to achieve on this assignment. This field may be `None`, and it is normally set by the grading system plugin.

DO NOT UPDATE MANUALLY. You can safely set an initial value for this manually when you create a new assignment, but when you update this field, do so using `set_max_points()`.

passing_grade_min_points

An IntegerField that contains the minimum number of points required to achive a passing grade on this assignment. This means that any feedback with more this number of points or more is considered a passing grade.

WARNING: Changing this does not have any effect on existing feedback. To actually change existing feedback, you would have to update all feedback on the assignment, effectively creating new StaticFeedbacks from the latest published FeedbackDrafts for each AssignmentGroup.

points_to_grade_mapper

Configures how points should be mapped to a grade. Valid choices:

- `passed-failed` - Points is mapped directly to passed/failed. Zero points results in a failing grade, other points results in a passing grade.
- `raw-points` - The grade is `<points>/<max-points>`.
- `table-lookup` - Points is mapped to a grade via a table lookup. This means that someone configures a mapping from point thresholds to grades using `devilry.apps.core.models.PointRangeToGrade`.

grading_system_plugin_id

A CharField containing the ID of the grading system plugin this assignment uses.

students_can_create_groups

BooleanField specifying if students can join/leave groups on their own.

If this is `True` students should be allowed to join/leave groups. If `students_can_not_create_groups_after` is specified, this students can not create groups after `students_can_not_create_groups_after` even if this is `True`.

This does not in any way affect an admins ability to organize students in groups manually.

students_can_not_create_groups_after

Students can not create project groups after this time. Ignored if `students_can_create_groups` is `False`.

DateTimeField that defaults to `None` (null).

students_can_create_groups_now

Return `True` if `students_can_create_groups` is `True`, and `students_can_not_create_groups_after` is in the future or `None`.

is_electronic()
Returns `True` if `deliverytypes` is `0` (electric).

New in version 1.4.0.

is_nonelectronic()
Returns `True` if `deliverytypes` is `1` (non-electric).

New in version 1.4.0.

set_max_points(*max_points*)
Sets `max_points`, and invalidates any `PointToGradeMap` configured for this assignment if the new value for `max_points` differs from the old one.

Invalidating the PointToGradeMap ensures that the course admin has to re-evaluate the grade to point mapping when they change `max_points`.

NOTE: This saves the PointToGradeMap, but not the assignment.

get_gradingsystem_plugin_api()
Shortcut for:

```
devilry.devilry_gradingsystem.pluginregistry.gradingsystempluginregistry.get(
    self.grading_system_plugin_id)(self)
```

See: `devilry.devilry_gradingsystem.pluginregistry.GradingSystemPluginRegistry.get()`.

has_valid_grading_setup()
Checks if this assignment is configured correctly for grading.

setup_grading(*grading_system_plugin_id*, *points_to_grade_mapper*, *passing_grade_min_points=None*, *max_points=None*)
Setup all of the simple parts of the grading system:

- `grading_system_plugin_id`
- `points_to_grade_mapper`
- `passing_grade_min_points`
- `max_points`

Does not setup:

- Grading system plugin specific configuration.
- A `PointToGradeMap`.

get_point_to_grade_map()
Get the `PointToGradeMap` for this assinment, creating if first if it does not exist.

points_is_passing_grade(*points*)
Checks if the given points represents a passing grade.

WARNING: This will only work if `passing_grade_min_points` is set. The best way to check that is with `has_valid_grading_setup()`.

points_to_grade(*points*)
Convert the given points into a grade.

WARNING: This will not work if `has_valid_grading_setup()` is not `True`.

clean(**args*, ***kwargs*)
Validate the assignment.

Always call this before save()! Read about validation here: http://docs.djangoproject.com/en/dev/ref/models/instances/#id1

Raises ValidationError if `publishing_time` is not between `Period.start_time` and `Period.end_time`.

is_empty()
Returns `True` if this Assignment does not contain any deliveries.

is_active()
Returns `True` if this assignment is published, and the period has not ended yet.

Examiner

class `devilry.apps.core.models.`**Examiner**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`, `devilry.apps.core.models.abstract_is_admin.AbstractIsA`

assignmentgroup
The AssignmentGroup where this groups belongs.

user
A foreign key to a User.

Candidate

class `devilry.apps.core.models.`**Candidate**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

assignment_group
The AssignmentGroup where this groups belongs.

student
A student (a foreign key to a User).

candidate_id
A optional candidate id. This can be anything as long as it is not more than 30 characters. When the assignment is anonymous, this is the "name" shown to examiners instead of the username of the student.

AssignmentGroup

class `devilry.apps.core.models.`**AssignmentGroup**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`, `devilry.apps.core.models.abstract_is_admin.AbstractIsA`
`devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer`,
`devilry.apps.core.models.model_utils.Etag`

Represents a student or a group of students.

parentnode
A django.db.models.ForeignKey that points to the parent node, which is always an Assignment.

name
An optional name for the group.

candidates
A django `RelatedManager` that holds the `candidates` on this group.

examiners
A django.db.models.ManyToManyField that holds the examiner(s) that are to correct and grade the assignment.

is_open
A django.db.models.BooleanField that tells you if the group can add deliveries or not.

`deadlines`
A django `RelatedManager` that holds the `deadlines` on this group.

`tags`
A django `RelatedManager` that holds the `tags` on this group.

`feedback`
The last StaticFeedback (by save timestamp) on this assignmentgroup.

`last_deadline`
The last `devilry.apps.core.models.Deadline` for this assignmentgroup.

`etag`
A DateTimeField containing the etag for this object.

`delivery_status`
A CharField containing the status of the group. Valid status values:

- "no-deadlines"
- "corrected"
- "closed-without-feedback"
- "waiting-for-something"

`save`(**args*, ***kwargs*)

Parameters

- **update_delivery_status** – Update the `delivery_status`? This is a somewhat expensive operation, so we provide the option to avoid it if needed. Defaults to `True`.
- **autocreate_first_deadline_for_nonelectronic** – Autocreate the first deadline if non-electronic assignment? Defaults to `True`.

classmethod `q_is_candidate`(*user_obj*)
Returns a django.models.Q object matching AssignmentGroups where the given student is candidate.

classmethod `where_is_candidate`(*user_obj*)
Returns a QuerySet matching all AssignmentGroups where the given user is student.

Parameters **user_obj** – A django.contrib.auth.models.User object.
Return type QuerySet

classmethod `published_where_is_candidate`(*user_obj*, *old=True*, *active=True*)
Returns a QuerySet matching all *published* assignment groups where the given user is student.

Parameters **user_obj** – A django.contrib.auth.models.User object.
Return type QuerySet

classmethod `active_where_is_candidate`(*user_obj*)
Returns a QuerySet matching all *active* assignment groups where the given user is student.

Parameters **user_obj** – A django.contrib.auth.models.User object.
Return type QuerySet

classmethod `old_where_is_candidate`(*user_obj*)
Returns a QuerySet matching all *old* assignment groups where the given user is student.

Parameters **user_obj** – A django.contrib.auth.models.User object.
Return type QuerySet

`should_ask_if_examiner_want_to_give_another_chance`
`True` if the current state of the group is such that the examiner should be asked if they want to give them another chance.

`True` if corrected with failing grade or closed without feedback.

`missing_expected_delivery`
Return `True` if the group has no deliveries, and we are expecting them to have made at least one delivery on the last deadline.

`subject`
Shortcut for `parentnode.parentnode.parentnode`.

`period`
Shortcut for `parentnode.parentnode`.

`assignment`
Alias for `parentnode`.

`short_displayname`
A short displayname for the group. If the assignment is anonymous, we list the candidate IDs. If the group has a name, the name is used, else we fall back to a comma separated list of usernames. If the group has no name and no students, we use the ID.

See also:

https://github.com/devilry/devilry-django/issues/498

`long_displayname`
A long displayname for the group. If the assignment is anonymous, we list the candidate IDs.

If the assignment is not anonymous, we use a comma separated list of the displaynames (full names with fallback to username) of the students. If the group has a name, we use the groupname with the names of the students in parenthesis.

See also:

https://github.com/devilry/devilry-django/issues/499

`get_students``()`
Get a string containing all students in the group separated by comma and a space, like: `superman, spiderman, batman`.

WARNING: You should never use this method when the user is not an administrator.

`get_examiners`(*separator=u', '*)
Get a string contaning the username of all examiners in the group separated by comma (`','`).

Parameters **separator** – The unicode string used to separate candidates. Defaults to `u', '`.

`is_examiner`(*user_obj*)
Return True if user is examiner on this assignment group

`can_delete`(*user_obj*)
Check if the given user is permitted to delete this AssignmentGroup. A user is permitted to delete an object if the user is superadmin, or if the user is admin on the assignment (uses `is_admin()`). Only superusers are allowed to delete AssignmentGroups where `AssignmentGroup.is_empty()` returns `False`.

Note: This method can also be used to check if candidates can be removed from the group.

Returns `True` if the user is permitted to delete this object.

`is_empty``()`
Returns `True` if this AssignmentGroup does not contain any deliveries.

`get_active_deadline``()`
Get the active `Deadline`.

This is always the last deadline on this group.

Returns The latest deadline or `None`.

`can_save`(*user_obj*)

Check if the user has permission to save this AssignmentGroup.

`can_add_deliveries`()

Returns true if a student can add deliveries on this assignmentgroup

Both the assignmentgroups is_open attribute, and the periods start and end time is checked.

`copy_all_except_candidates`()

Note: Always run this is a transaction.

`pop_candidate`(*candidate*)

Make a copy of this group using `copy_all_except_candidates`, and add given candidate to the copied group and remove the candidate from this group.

Parameters **candidate** – A `devilry.apps.core.models.Candidate` object. The candidate must be among the candidates on this group.

Note: Always run this is a transaction.

`recalculate_delivery_numbers`()

Query all `successful` deliveries on this AssignmentGroup, ordered by `time_of_delivery` ascending, and number them with the oldest delivery as number 1.

`merge_into`(*target*)

Merge this AssignmentGroup into the `target` AssignmentGroup. Algorithm:

- Copy in all candidates and examiners not already on the AssignmentGroup.
- **Delete all copies where the original is in `self` or `target`:**
  - Delete all deliveries from `target` that are `copy_of` a delivery `self`.
  - Delete all deliveries from `self` that are `copy_of` a delivery in `target`.
- Loop through all deadlines in this AssignmentGroup, and for each deadline:

  If the datetime and text of the deadline matches one already in `target`, move the remaining deliveries into the target deadline.

  If the deadline and text does NOT match a deadline already in `target`, change assignmentgroup of the deadline to the master group.
- Recalculate delivery numbers of `target` using `recalculate_delivery_numbers()`.
- Run `self.delete()`.
- Set the latest feedback on `target` as the active feedback.

Note: The `target.name` or `target.is_open` is not changed.

Note: Everything except setting the latest feedback runs in a transaction. Setting the latest feedback does not run in transaction because we need to save the with `feedback=None`, and then set the *new* latest feedback to avoid IntegrityError.

classmethod `merge_many_groups`(*sources*, *target*)

Loop through the `sources`-iterable, and for each `source` in the iterator, run `source.merge_into(target)`.

`get_status`()

Get the status of the group. Calculated with this algorithm:

```
if ``delivery_status == 'waiting-for-something'``
    if assignment.delivery_types==NON_ELECTRONIC:
        "waiting-for-feedback"
    else
        if before deadline
            "waiting-for-deliveries"
        if after deadline:
            "waiting-for-feedback"
else
    delivery_status
```

AssignmentGroupTag

class `devilry.apps.core.models.`**`AssignmentGroupTag`**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`

An AssignmentGroup can be tagged with zero or more tags using this class.

`assignment_group`

The AssignmentGroup where this groups belongs.

`tag`

The tag. Max 20 characters. Can only contain a-z, A-Z, 0-9 and “_”.

Deadline

Each AssignmentGroup have zero or more deadlines.

class `devilry.apps.core.models.`**`Deadline`**(**args*, ***kwargs*)

Bases: `django.db.models.base.Model`, `devilry.apps.core.models.abstract_is_admin.AbstractIsA` `devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer`, `devilry.apps.core.models.abstract_is_candidate.AbstractIsCandidate`

A deadline on an AssignmentGroup. A deadline contains zero or more deliveries, the time of the deadline and an optional text.

`assignment_group`

The AssignmentGroup where the deadline is registered.

`deadline`

The deadline a DateTimeField.

`text`

A optional deadline text.

`deliveries`

A django `RelatedManager` that holds the deliveries on this group. NOTE: You should normally not use this directly, but rather use meth:.*query_successful_deliveries*.

`deliveries_available_before_deadline`

Should deliveries on this deadline be available to examiners before the deadline expires? This is set by students.

`feedbacks_published`

If this boolean field is `True`, the student can see all `StaticFeedback` objects associated with this Deadline through a `Delivery`. See also `Assignment.examiners_publish_feedbacks_directly`.

`added_by`
The User that added this deadline. Can be `None`, and all deadlines created before Devilry version `1.4.0` has this set to `None`.

New in version 1.4.0.

`why_created`
Why was this deadline created? Valid choices:

- `None`: Why the deadline was created is unknown.
- `"examiner-gave-another-chance"`: Created because the examiner elected to give the student another chance to pass the assignment.

Can be `None`, and all deadlines created before Devilry version `1.4.0` has this set to `None`.

New in version 1.4.0.

classmethod `reduce_datetime_precision`(*datetimeobj*)
Reduce the precition of the `datetimeobj` to make it easier to compare and harder to make distinct deadlines that is basically the same time. We:

- Set seconds and microseconds to `0`. This makes "Friday 14:59", "Friday 14:59:00" and "Friday 14:59:59" equal. We do not allow specifying seconds in the UI, and handling this right in the core makes this easier to handle across the board.
- Set tzinfo to None. We do not support timezones in Devilry, so including it makes no sense.

Returns A copy of `datetimeobj` with second and microsecond set to `0`, and tzinfo set to `None`.

`clean`(**args*, ***kwargs*)
Validate the deadline.

Always call this before save()! Read about validation here: http://docs.djangoproject.com/en/dev/ref/models/instances/#id1

Raises ValidationError if:

- `deadline` is before `Assignment.publishing_time`.
- `deadline` is not before `Period.end_time`.

`save`(**args*, ***kwargs*)
Parameters **autocreate_delivery_if_nonelectronic** – Autocreate a delivery if this save creates the deadline, and the assignment is non-electronic. Defaults to `True`.

`query_successful_deliveries`()
Returns a django QuerySet that filters all the successful deliveries on this group.

`is_empty`()
Returns `True` if this Deadline does not contain any deliveries.

`can_delete`(*user_obj*)
Check if the given user is permitted to delete this object. A user is permitted to delete an Deadline if the user is superadmin, or if the user is admin on the assignment. Only superusers are allowed to delete deadlines with any deliveries.

Returns `True` if the user is permitted to delete this object.

`copy`(*newgroup*)
Copy this deadline into `newgroup`, including all deliveries and filemetas, with the actual file data.

Note: Always run this is a transaction.

Warning: This does not autoset the latest feedback as active on the group. You need to handle that yourself after the copy.

is_in_the_future()
Return True if this deadline is in the future.

is_in_the_past()
Return True if this deadline is in the past.

has_text()
Checks that the text is not None or an empty string.

Delivery

Examples Simple example:

```
assignmentgroup = AssignmentGroup.objects.get(id=1)
assignmentgroup.deliveries.create(delivered_by=student1,
                                  successful=True)
```

More advanced example:

```
assignmentgroup = AssignmentGroup.objects.get(id=1)
delivery = assignmentgroup.deliveries.create(delivered_by=student1,
                                             successful=False)
delivery.add_file('test.py', ['print', 'hello world'])
delivery.add_file('test2.py', ['print "hi"'])
delivery.successful = True
delivery.save()
```

The input to add_file() will normally be a file-like object, but as shown above it can be anything you want.

Delivery API

class devilry.apps.core.models.**Delivery**(**args*, ***kwargs*)
Bases: django.db.models.base.Model, devilry.apps.core.models.abstract_is_admin.AbstractIsA
devilry.apps.core.models.abstract_is_candidate.AbstractIsCandidate,
devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer

A class representing a given delivery from an AssignmentGroup.

How to create a delivery:

```
deadline = Deadline.objects.get(....)
candidate = Candidate.objects.get(....)
delivery = Delivery(
    deadline=deadline,
    delivered_by=candidate)
delivery.set_number()
delivery.full_clean()
delivery.save()
```

time_of_delivery
A django.db.models.DateTimeField that holds the date and time the Delivery was uploaded.

deadline
A django.db.models.ForeignKey pointing to the Deadline for this Delivery.

number
A django.db.models.fields.PositiveIntegerField with the delivery-number within this assignment-group.

This number is automatically incremented within each assignmentgroup, starting from 1. Must be unique within the assignment-group. Automatic incrementation is used if number is None when calling `save()`.

`delivered_by`
A django.db.models.ForeignKey pointing to the user that uploaded the Delivery

`successful`
A django.db.models.BooleanField telling whether or not the Delivery was successfully uploaded.

`after_deadline`
A django.db.models.BooleanField telling whether or not the Delivery was delived after deadline..

`filemetas`
A set of `filemetas` for this delivery.

`feedbacks`
A set of `feedbacks` on this delivery.

`etag`
A DateTimeField containing the etag for this object.

`copy_of`
Link to a delivery that this delivery is a copy of. This is set by `Delivery.copy()`.

`last_feedback`
The last StaticFeedback on this delivery. This is updated each time a feedback is added.

`copy_of`
If this delivery is a copy of another delivery, this ForeignKey points to that other delivery.

`copies`
The reverse of `copy_of` - a queryset that returns all copies of this delivery.

`after_deadline`
Compares the deadline and time of delivery. If time_of_delivery is greater than the deadline, return True.

classmethod `q_is_candidate`(*user_obj*)
Returns a django.models.Q object matching Deliveries where the given student is candidate.

`is_last_delivery`
Returns `True` if this is the last delivery for this AssignmentGroup.

`assignment_group`
Shortcut for `self.deadline.assignment_group.assignment`.

`assignment`
Shortcut for `self.deadline.assignment_group.assignment`.

`add_file`(*filename*, *iterable_data*)
Add a file to the delivery.

Parameters
- **filename** – A filename as defined in `FileMeta`.
- **iterable_data** – A iterable yielding data that can be written to file using the write() method of a storage backend (byte strings).

`clean`(**args*, ***kwargs*)
Validate the delivery.

`copy`(*newdeadline*)
Copy this delivery, including all FileMeta's and their files, and all feedbacks into `newdeadline`. Sets the `copy_of` attribute of the created delivery.

Note: Always run this in a transaction.

Warning: This does not autoset the latest feedback as `feedback` or the `last_delivery` on the group. You need to handle that yourself after the copy.

Returns The newly created, cleaned and saved delivery.

`is_electronic``()`
Returns `True` if `Delivery.delivery_type` is `0` (electric).

`is_nonelectronic``()`
Returns `True` if `Delivery.delivery_type` is `1` (non-electric).

StaticFeedback

class `devilry.apps.core.models.`**`StaticFeedback`**`(`**args*, ***kwargs*`)`
Bases: `django.db.models.base.Model`, `devilry.apps.core.models.abstract_is_admin.AbstractIsA` `devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer`, `devilry.apps.core.models.abstract_is_candidate.AbstractIsCandidate`

Represents a feedback for a Delivery.

Each delivery can have zero or more feedbacks. Each StaticFeedback object stores static data that an examiner has published on a delivery. StaticFeedback is created and edited in a *grade+feedback editor* in a *grade plugin*, and when an examiner choose to publish feedback, a static copy of the data he/she created in the *grade+feedback editor* is stored in a StaticFeedback.

Feedbacks are only visible to students when `Deadline.feedbacks_published` on the related deadline is `True`. Feedbacks are related to Deadlines through its `delivery`.

Students are presented with the last feedback on a delivery, however they can browse every StaticFeedback on their deliveries. This history is to protect the student from administrators or examiners that change published feedback to avoid that a student can make an issue out of a bad feedback.

NOTE: When a StaticFeedback is saved, the corresponding `AssignmentGroup.feedback` is updated to the newly created StaticFeedback.

`rendered_view`
The rendered HTML view.

`saved_by`
The django.contrib.auth.models.User that created the StaticFeedback.

`save_timestamp`
Date/time when this feedback was created.

`delivery`
A django.db.models.ForeignKey that points to the Delivery where this feedback belongs.

`grade`
The grade as a short string (max 12 chars).

`points`
The number of points (integer).

`is_passing_grade`
Boolean is passing grade?

classmethod **`q_is_candidate`**`(`*user_obj*`)`
Returns a django.models.Q object matching Deliveries where the given student is candidate.

classmethod q_is_examiner(*user_obj*)
Returns a django.models.Q object matching Feedbacks where the given student is candidate.

classmethod from_points(*points*, *assignment=None*, ***kwargs*)
Shortcut method to initialize the StaticFeedback object from points.

Initializes a StaticFeedback with the given points, with grade and is_passing_grade inferred from the points with the help of `devilry.apps.core.models.Assignment.points_to_grade()` and `devilry.apps.core.models.Assignment.points_is_passing_grade()`.

Example:

```
feedback = StaticFeedback.from_points(
    assignment=myassignment,
    points=10,
    delivery=mydelivery,
    saved_by=someuser)
assert(feedback.id == None)
assert(feedback.grade != None)
```

Parameters
- **points** – The number of points for the feedback.
- **assignment** – An Assignment object. Should be the assignment where delivery this feedback is for belongs, but that is not checked.

  Defaults to `self.delivery.deadline.assignment_group.assignment`.

  We provide the ability to take the assignment as argument instead of looking it up via `self.delivery.deadline.assignment_group` because we want to to be efficient when creating feedback in bulk.
- **kwargs** – Extra kwargs for the StaticFeedback constructor.

Returns An (unsaved) StaticFeedback.

save(**args*, ***kwargs*)

Parameters
- **autoset_timestamp_to_now** – Automatically set the `timestamp`-attribute of this model to *now*? Defaults to `True`.
- **autoupdate_related_models** – Automatically update related models:
  - Sets the `last_feedback`-attribute of `self.delivery` and saved the delivery.
  - Sets the `feedback` and `is_open` attributes of `self.delivery.deadline.assignment_group` to this feedback, and `False`. Saves the AssignmentGroup.

  Defaults to `True`.

copy(*newdelivery*)
Copy this StaticFeedback into `newdeadline`.

Note: This only copies the StaticFeedback, not any data related to it via any grade editors.

Warning: This does not autoset the feedback as active on the group or as latest on the delivery. You need to handle that yourself after the copy.

FileMeta

class `devilry.apps.core.models.`**`FileMeta`**`(`**args*, ***kwargs*`)`

Bases: `django.db.models.base.Model`, `devilry.apps.core.models.abstract_is_admin.AbstractIsA` `devilry.apps.core.models.abstract_is_examiner.AbstractIsExaminer`, `devilry.apps.core.models.abstract_is_candidate.AbstractIsCandidate`

Represents the metadata for a file belonging to a Delivery.

A file meta is just information about a single file, which is stored in a `deliverystore`. Use the `deliverystore` to manage the file stored in its physical location. Example:

```
filemeta = FileMeta.objects.get(pk=0)
if filemeta.deliverystore.exists(filemeta):
    filemeta.deliverystore.remove(filemeta)

# Write or read just as with the builtin open()
fobj = filemeta.deliverystore.write_open(filemeta)
fobj.write('Hello')
fobj.write('World')
fobj.close()
fobj = filemeta.deliverystore.read_open(filemeta)
print fobj.read()
```

See *DeliveryStore* for more details on deliverystores.

`delivery`

A django.db.models.ForeignKey that points to the Delivery of the given feedback.

`filename`
Name of the file.

`size`
Size of the file in bytes.

`deliverystore`
The current *DeliveryStore*. *Class variable*.

`get_all_data_as_string``()`
Get all data store in the deliverystore for this FileMeta as a string. THIS IS ONLY FOR TESTING, and should NEVER be used for production code, since it will eat all memory on the server for huge files.

`copy``(`*newdelivery*`)`
Copy this filemeta into `newdelivery`. Copies the database object and the data in the deliverystore.

DevilryUserProfile

See also: *The Devilry User object*.

class `devilry.apps.core.models.`**`DevilryUserProfile`**`(`**args*, ***kwargs*`)`

Bases: `django.db.models.base.Model`

User profile with a one-to-one relation to `django.contrib.auth.models.User`.

Ment to be used as a Django *user profile* (AUTH_PROFILE_MODULE).

`full_name`
Django splits names into first_name and last_name. They are only 30 chars each. Read about why this is not a good idea here:

http://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/
Since we require support for *any* name, we use our own `full_name` field, and ignore the one in Django. Max length 300.

languagecode
Used to store the preferred language for a user. Not required (The UI defaults to the default language)

get_displayname()
Get a name for this user, preferrably the full name, but falls back to username of that is unavailable.

The Devilry User object

Django user

Devilry users are Django django.contrib.auth.models.User objects. However we only use a subset of the fields:

- username
- email
- is_superuser
- password (if authenticating using the default Django auth)

Additional data

Additional data is stored in a one-to-one relation to `devilry.apps.core.models.DevilryUserProfile`. The profile object is available through the `devilryuserprofile` attribute of any django.contrib.auth.models.User object in devilry. It can be used in queries just like any other one-to-one relation, like this:

```
from django.contrib.auth.models import User
supermen = User.objects.filter(devilryuserprofile__full_name__contains="Superman")
```

`devilry.apps.core.deliverystore` — DeliveryStore

A *DeliveryStore* is a place to put the files from deliveries. In more technical terms, it is a place where each file related to a `devilry.apps.core.models.FileMeta` is stored.

Selecting a DeliveryStore

Devilry on comes with one DeliveryStore ready for production use, `FsDeliveryStore`. To enable a DeliveryStore, you have to set the `DELIVERY_STORE_BACKEND`-setting in your *settings.py* like this:

```
DELIVERY_STORE_BACKEND = 'devilry.apps.core.deliverystore.FsDeliveryStore'
```

The FsDeliveryStore also require you to define where on the disk you wish to store your files in the `DELIVERY_STORE_ROOT`-setting like this:

```
DELIVERY_STORE_ROOT = '/path/to/root/directory/of/my/deliverystore'
```

Creating your own DeliveryStore

To create your own DeliveryStore you have to implement `DeliveryStoreInterface`. A good example is `FsDeliveryStore`:

```
class FsDeliveryStore(DeliveryStoreInterface):
    """
    Filesystem-based DeliveryStore suitable for production use.

    It stores files in a filesystem hierarcy with one directory for each
    Delivery, with the delivery-id as name. In each delivery-directory, the
    files are stored by FileMeta id.
    """
    def __init__(self, root=None):
        """
        :param root: The root-directory where files are stored. Defaults to the value of the ``D
        """
        self.root = root or settings.DELIVERY_STORE_ROOT

    def _get_dirpath(self, delivery_obj):
        return join(self.root, str(delivery_obj.pk))

    def _get_filepath(self, filemeta_obj):
        return join(self._get_dirpath(filemeta_obj.delivery),
                    str(filemeta_obj.pk))

    def read_open(self, filemeta_obj):
        filepath = self._get_filepath(filemeta_obj)
        if not exists(filepath):
            raise FileNotFoundError(filemeta_obj)
        return open(filepath, 'rb')

    def _create_dir(self, filemeta_obj):
        dirpath = self._get_dirpath(filemeta_obj.delivery)
        if not exists(dirpath):
            makedirs(dirpath)

    def write_open(self, filemeta_obj):
        self._create_dir(filemeta_obj)
        return open(self._get_filepath(filemeta_obj), 'wb')

    def remove(self, filemeta_obj):
        filepath = self._get_filepath(filemeta_obj)
        if not exists(filepath):
            raise FileNotFoundError(filemeta_obj)
        remove(filepath)

    def exists(self, filemeta_obj):
        filepath = self._get_filepath(filemeta_obj)
        return exists(filepath)

    def copy(self, filemeta_obj_from, filemeta_obj_to):
        frompath = self._get_filepath(filemeta_obj_from)
        topath = self._get_filepath(filemeta_obj_to)
        self._create_dir(filemeta_obj_to)
        shutil_copy(frompath, topath)
```

Testing your own DeliveryStore We provide a mixing-class, `devilry.apps.core.testhelpers.DeliveryStoreTestMi` for you to extend when writing unit-tests for your DeliveryStore. Here is how we test `FsDeliveryStore`:

```
class TestFsDeliveryStore(DeliveryStoreTestMixin, TestCase):
    def setUp(self):
        self.root = mkdtemp()
        super(TestFsDeliveryStore, self).setUp()

    def get_storageobj(self):
        return FsDeliveryStore(self.root)

    def tearDown(self):
        rmtree(self.root)
```

class `devilry.apps.core.testhelpers.`**`DeliveryStoreTestMixin`**

Bases: `devilry.apps.core.testhelper.TestHelper`

Mixin-class that tests if `devilry.core.deliverystore.DeliveryStoreInterface` is implemented correctly.

You only need to override `get_storageobj()`, and maybe `setUp()` and `tearDown()`, but make sure you call `super(..., self).setUp()` if you override it.

You **must** mixin this class before `django.test.TestCase` like so:

```
class TestMyDeliveryStore(DeliveryStoreTestMixin, django.test.TestCase):
    ...
```

`get_storageobj``()`

Return a object implementing `devilry.core.deliverystore.DeliveryStoreInterface`

`setUp``()`

Make sure to call this if you override it in subclasses, or the tests **will fail**.

Setting the DeliveryStore manually - for tests

You might need to set the DeliveryStore manually if you need to handle deliveries in your own tests. Just set `devilry.apps.core.FileMeta.deliveryStore` like this:

```
from django.test import TestCase
from devilry.apps.core.models import FileMeta, Delivery
from devilry.apps.core.deliverystore import MemoryDeliveryStore

class MyTest(TestCase):
    def test_something(self):
        FileMeta.deliverystore = MemoryDeliveryStore()
        delivery = Delivery.begin(assignmentgroup, user)
        delivery.add_file('hello.txt', ['hello', 'world'])
        delivery.finish()
```

The recommended production deliverystore

The recommended DeliveryStore is `devilry.apps.core.deliverystore.FsHierDeliveryStore`.

It stores files in a filesystem hierarcy with one directory for each Delivery, with the delivery-id as name. In each delivery-directory, the files are stored by FileMeta id.

Directory hierachy The delivery directories are stored in a hierarchy with two parent directories. The parent directories are numeric intervals. We have one top-level directory for each N in `interval_size*interval_size*N`. Within each toplevel directory, we have one subdirectory for each N in `interval_size*N`.

Directory hierarchy example For `interval_size` of `1000`, this will use the following hierarchy:

```
0/
    0/
        0/
        1/
        .
        .
    1/
        1000/
        2000/
        .
        .
    2/
    .
    .
    999/
1/
    0/
        1000000/
        1000001/
        .
        .
    1/
        1001000/
        1001001/
    2/
    .
    .
    999/
2/
.
.
999/
```

API

exception `devilry.apps.core.deliverystore.`**`FileNotFoundError`**(*filemeta_obj*)

Bases: `exceptions.Exception`

Exception to be raised when the remove method of a DeliveryStore does not find the given file.

class `devilry.apps.core.deliverystore.`**`DeliveryStoreInterface`**

Bases: `object`

The interface all deliverystores must implement. All methods raise `NotImplementedError`.

`read_open`(*filemeta_obj*)

Return a file-like object opened for reading.

The returned object must have `close()` and `read()` methods as defined by the documentation of the standard python file-class.

Parameters **filemeta_obj** – A `devilry.core.models.FileMeta`-object.

write_open (*filemeta_obj*)

Return a file-like object opened for writing.

The returned object must have `close()` and `write()` methods as defined by the documentation of the standard python file-class.

Parameters **filemeta_obj** – A `devilry.core.models.FileMeta`-object.

remove (*filemeta_obj*)

Remove the file.

Note that this method is called *before* the filemeta_obj is removed. This means that the file might be removed, and the removal of the filemeta_obj can still fail. To prevent users from having to manually resolve such cases implementations should check if the file exists, and raise FileNotFoundError if it does not.

The calling function has to check for FileNotFoundError and handle any other error.

Parameters **filemeta_obj** – A `devilry.core.models.FileMeta`-object.

exists (*filemeta_obj*)

Return `True` if the file exists, `False` if not.

Parameters **filemeta_obj** – A `devilry.core.models.FileMeta`-object.

copy (*filemeta_obj_from*, *filemeta_obj_to*)

Copy the underlying file-object for `filemeta_obj_from` into the file-object for `filemeta_obj_to`.

Defaults to an inefficient implementation using `read_open()` and meth:.*write_open*. Should be overridden for backends with some form of native copy-capability.

class `devilry.apps.core.deliverystore.`**FsDeliveryStore** (*root=None*)

Bases: `devilry.apps.core.deliverystore.DeliveryStoreInterface`

Filesystem-based DeliveryStore suitable for production use.

It stores files in a filesystem hierarcy with one directory for each Delivery, with the delivery-id as name. In each delivery-directory, the files are stored by FileMeta id.

Parameters **root** – The root-directory where files are stored. Defaults to the value of the `DELIVERY_STORE_ROOT`-setting.

class `devilry.apps.core.deliverystore.`**FsHierDeliveryStore** (*root=None*, *interval=None*)

Bases: `devilry.apps.core.deliverystore.FsDeliveryStore`

Filesystem-based DeliveryStore suitable for production use with huge amounts of deliveries.

Parameters

- **root** – The root-directory where files are stored. Defaults to the value of the `DEVILRY_FSHIERDELIVERYSTORE_ROOT`-setting.
- **interval** – The interval. Defaults to the value of the `DEVILRY_FSHIERDELIVERYSTORE_INTERVAL`-setting.

get_path_from_deliveryid (*deliveryid*)

```
>>> fs = FsHierDeliveryStore('/stuff/', interval=1000)
>>> fs.get_path_from_deliveryid(deliveryid=2001000)
(2, 1)
>>> fs.get_path_from_deliveryid(deliveryid=1000)
(0, 1)
>>> fs.get_path_from_deliveryid(deliveryid=1005)
(0, 1)
>>> fs.get_path_from_deliveryid(deliveryid=2005)
(0, 2)
```

```
>>> fs.get_path_from_deliveryid(deliveryid=0)
(0, 0)
>>> fs.get_path_from_deliveryid(deliveryid=1)
(0, 0)
>>> fs.get_path_from_deliveryid(deliveryid=1000000)
(1, 0)
```

class `devilry.apps.core.deliverystore.`**`MemoryDeliveryStore`**

Bases: `devilry.apps.core.deliverystore.DeliveryStoreInterface`

Memory-base DeliveryStore ONLY FOR TESTING.

This is only for testing, and it does not handle parallel access. Suitable for unittesting.

`devilry.utils` — Various utility functions

`devilry.utils.assignmentgroup`

class `devilry.utils.assignmentgroup.`**`GroupDeliveriesByDeadline`**(*group*)

Deliveries on an assignmentgroup is returned in a list of tuples, where each tuple contains the deadline, and all the deliveries on that deadline. If the default deadline (head) contains no deliveries, it is ignored.

`devilry.utils.ordereddict`

class `devilry.utils.`**`OrderedDict`**

If python version >=2.7, collections.OrderedDict is imported. For older python versions, the fallback mentioned in the OrderedDict docs is imported.

`devilry.utils.delivery_collection`

class `devilry.utils.delivery_collection.`**`ArchiveException`**

Bases: `exceptions.Exception`

Archive exceptions

`devilry.utils.delivery_collection.`**`create_archive_from_assignmentgroups`**(*request, assignmentgroups, file_name, archive_type*)

Creates an archive of type archive_type, named file_name, containing all the deliveries in each of the assignmentgroups in the list assignmentgroups.

`devilry.utils.delivery_collection.`**`create_archive_from_delivery`**(*request, delivery, archive_type*)

Creates an archive of type archive_type, named assignment.get_path(), containing all files in the delivery.

`devilry.utils.delivery_collection.`**`iter_archive_deliveries`**(*archive, group_name, directory_prefix, deliveries*)

Adds files one by one from the list of deliveries into the archive. After writing each file to the archive, the new bytes in the archive is yielded. If a file is bigger than DEVILRY_MAX_ARCHIVE_CHUNK_SIZE, only

DEVILRY_MAX_ARCHIVE_CHUNK_SIZE bytes are written before it's yielded. The returned object is an iterator.

`devilry.utils.delivery_collection.`**`iter_archive_assignmentgroups`**(*archive*, *assignmentgroups*)

Creates an archive, adds files delivered by the assignmentgroups and yields the data.

`devilry.utils.delivery_collection.`**`verify_groups_not_exceeding_max_file_size`**(*assignmentgroups*)

For each assignmentgroups in groups, calls `verify_deliveries_not_exceeding_max_file_size()`. If the size of a file in a delivery exceeds the settings.DEVILRY_MAX_ARCHIVE_CHUNK_SIZE, an ArchiveException is raised.

`devilry.utils.delivery_collection.`**`verify_deliveries_not_exceeding_max_file_size`**(*deliveries*)

Goes through all the files in each deliverery, and if the size of a file exceeds the DEVILRY_MAX_ARCHIVE_CHUNK_SIZE, an ArchiveException is raised.

devilry.utils.groupnodes

class `devilry.utils.`**`GroupNode`**

The node object containing a node, and GroupNode children.

`group_assignmentgroups`(*assignment_group_list*)

Creates a tree where each assignmentgroup is represented as a GroupNode. assignmentgroups with the same parent (period) are grouped together.

`group_assignments`(*assignment_list*)

Creates a tree where each assignment is represented as a GroupNode. assignments with the same parent (period) are grouped together.

`group_nodes`(*node_list*, *tree_height*)

Creates a tree where each node is represented as a GroupNode. nodes with the same parent (period) are grouped together.

devilry.utils.devilry_email

exception **`NoEmailAddressException`**

Raised when email adress is missing on users.

`send_email`(*user_objects_to_send_to*, *subject*, *message*)

Send email to the list of users in user_objects_to_send_to

`send_email_admins`(*subject*, *message*, *fail_silently=False*)

Send email to admins registered in settings.ADMINS.

devilry.utils.groups_groupedby_relatedstudent_and_assignment

Provides an easy-to-use API for generating overviews over the results of all students in a period. Collects students that are not related as well as related.

Example Create CSV with the grades of all students on the period, including those ignored because they are not related:

```
grouper = GroupsGroupedByRelatedStudentAndAssignment(myperiod)

header = ['USER','IGNORED']
for assignment in grouper.iter_assignments():
    header.append(assignment.short_name)
print ';'.join(header)

def print_aggregated_relstudentinfo(aggregated_relstudentinfo, ignored):
    user = aggregated_relstudentinfo.user
    row = [user.username, ignored]
    for grouplist in aggregated_relstudentinfo.iter_groups_by_assignment():
        # NOTE: There can be more than one group if the same student is in more than one
        #       group on an assignment - we select the "best" feedback.
        feedback = grouplist.get_feedback_with_most_points()
        if feedback:
            row.append(feedback.grade)
        else:
            row.append('NO-FEEDBACK')
    print ';'.join(row)

# Print all related students
for aggregated_relstudentinfo in grouper.iter_relatedstudents_with_results():
    print_aggregated_relstudentinfo(aggregated_relstudentinfo, 'NO')

# Last we print the ignored students (non-related students that are in a group)
for aggregated_relstudentinfo in grouper.iter_students_with_feedback_that_is_candidate_but_not_in_rel
    print_aggregated_relstudentinfo(aggregated_relstudentinfo, 'YES')
```

API

class devilry.utils.groups_groupedby_relatedstudent_and_assignment.**GroupList**

Bases: list

Represents a list of devilry.apps.core.models.AssignmentGroup objects, with utility functions for commonly needed actions. The list is ment to hold groups where the same student in candidate on a single assignment, and the utilities is ment to make it easier to work with the added complexity of supporting the same user in multiple groups on a single assignment.

get_feedback_with_most_points()

Get the devilry.apps.core.models.StaticFeedback with the most points in the list.

get_best_gradestring()

Uses get_feedback_with_most_points() to get the feedback with most points, and returns the grade-attribute of that feedaback.

Returns The grade or None.

class devilry.utils.groups_groupedby_relatedstudent_and_assignment.**AggreatedRelatedStudentInf**

Bases: object

Used by GroupsGroupedByRelatedStudentAndAssignment to stores all results for a single student on a period.

user = None
The Django user object for the student.

assignments = None
Dict of assignments where the key is the assignment-id, and the value is a `GroupList`.

relatedstudent = None
The `devilry.apps.core.models.RelatedStudent` for users that are related students. This is only available for the objects returned by `GroupsGroupedByRelatedStudentAndAssignment.iter_relatedstudents_with_results()`, and not for the objects returned by the ignored students iterators.

iter_groups_by_assignment()
Returns an iterator over all `GroupList` objects for this student. Shortcut for `self.assignments.itervalues()`.

add_group(*group*)
Used by `GroupsGroupedByRelatedStudentAndAssignment` to add groups.

prettyprint()
Prettyprint for debugging.

class `devilry.utils.groups_groupedby_relatedstudent_and_assignment.`**GroupsGroupedByRelatedStuc**
Bases: `object`

Provides an easy-to-use API for overviews over the results of all students in a period.
Parameters period – A `devilry.apps.core.models.Period` object.

get_assignment_queryset()
Get the queryset used to fetch all assignments on the period. Override for custom ordering or if you need to optimize the query for your usecase (`select_related`, `prefetch_related`, etc.)

get_relatedstudents_queryset()
Get the queryset used to fetch all relatedstudents on the period. Override if you need to optimize the query for your usecase (`select_related`, `prefetch_related`, etc.)

get_groups_queryset()
Get the queryset used to fetch all groups on the period. Override if you need to optimize the query for your usecase (`select_related`, `prefetch_related`, etc.)

iter_assignments()
Iterate over all the assignments, yielding Assignment-objects. The objects are iterated in the order returned by `get_assignment_queryset()`.

iter_relatedstudents_with_results()
Iterate over all relatedstudents, yielding a dict with the following attributes for each related student:

user The Django user-object for the student.

assignments An OrderedDict, ordered the same as `iter_assignments()`, where the key is the assignment-id, and the value is a list of AssignmentGroup-objects where the user is candidate. The list may have 0 or more groups, 0 if the user is not in any group on the assignment, and more than 1 if the user is in more than one group on the assignment.

iter_students_that_is_candidate_but_not_in_related()
Iterate over the students that is candidate on one or more groups, but not registered as related students.

This iterator includes everything yielded by both:

- `iter_students_with_feedback_that_is_candidate_but_not_in_related()`
- `iter_students_with_no_feedback_that_is_candidate_but_not_in_related()`

iter_students_with_feedback_that_is_candidate_but_not_in_related()
Same as `iter_students_that_is_candidate_but_not_in_related()`, but it does not include the students that have no feedback.

iter_students_with_no_feedback_that_is_candidate_but_not_in_related()
Iterate over everything returned by `iter_students_that_is_candidate_but_not_in_related()` except for the students returned by `iter_students_with_feedback_that_is_candidate_but_not_in_re`

serialize()
Serialize all the collected data as plain python objects.

There are more utils than the ones listed above. Read the source. The most useful is probably:

- `devilry.utils.passed_in_previous_period` — Find students that passed the course in previous periods/semesters.

Advanced topics

Most developers will not need to bother with these topics.

Developing and testing Celery background tasks

How Celery is configured

Celery is configured according to the Celery first steps with Django guide. The app is in `devilry.project.common.celery`, and it is imported as `celery_app` in `devilry/project/common/__init__.py`.

For production, we leave the configuration up to sysadmins.

For development, we default to running Celery in eager mode, but we have commented out settings in `devilry.project.develop.develop` for "real" Celery testing. Eager mode means that all celery tasks runs in blocking mode in the current thread, so celery tasks runs just like any other function.

For unit tests, we run Celery in eager mode (configured in `devilry.project.develop.test`).

Testing with non-eager Celery

Install Redis See https://redis.io/. On Mac OSX, you can install Redis using Homebrew:

```
$ brew install redis
```

Start the Redis server To start the redis server, run:

```
$ redis-server
```

To stop the server, run:

```
$ redis-server stop
```

To stop the server on OSX, run:

```
$ redis-cli shutdown
```

Start the Celery worker Run:

```
$ celery -A devilry.project.common worker -l debug
```

It should print some info about the config, the tasks that it detects in Devilry, and stop for input with the following message: `celery@<your machine name> is ready`.

Try one of the test-tasks Open the Django shell, and run one the test-tasks (while Redis and the Celery worker are both running):

```
$ python manage.py shell
>>> from devilry.project.develop.tasks import add
>>> result = add.delay(10, 20)
>>> result.wait()
30
```

If this works, Celery is configured correctly, and you should be able to see the job in the terminal where the worker is running.

Things to remember (when running Celery tasks through the Celery worker)

- The output (stdout and stderr) goes to the Celery worker, not to runserver.
- You can get more verbose output from the worker with `worker -l debug`.

Testing email sending with django-celery-email Uncomment the following lines in `devilry.project.develop.settings.develop`:

```
# INSTALLED_APPS += ['djcelery_email']
# EMAIL_BACKEND = 'djcelery_email.backends.CeleryEmailBackend'
# CELERY_EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend"
```

And run the following in the Django shell:

```
>>> from django.contrib.auth import get_user_model
>>> from devilry.utils.devilry_email import send_message
>>> send_message('Testsubject', 'Testmessage', get_user_model().objects.get(username='april'))
```

How to write a plugin

Warning: Plugins will be phased out in 2.1.0 when we update to Django 1.7.

A plugin is basically just a normal Django application. The only thing making it a *pugin* is that it integrates itself into the Devilry system in some way.

Setting up your testsite

In this *howto* we assume you have created a django site, `mysite/`, and and that your plugin is a application in this site called `myplugin`. It should look something like this:

```
mysite/
    settings.py
    manage.py
    urls.py
    myplugin/
        models.py
        urls.py
```

Autoload plugins

There are several ways a plugin can integrate itself, but they all need some place to do the integration. Just like `admin.py` can be used to integrate your application with the Django admin interface, devilry provides a place where you can put code that you want to autoload.

First initialize the plugin system by adding:

```
from devilry.apps.core import pluginloader
pluginloader.autodiscover()
```

to your `mysite/urls.py`, making it look something like this:

```
from django.conf.urls import *

# Uncomment the next two lines to enable the admin:
#from django.contrib import admin
#admin.autodiscover()

from devilry.apps.core import pluginloader
pluginloader.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.foo.urls')),

    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # (r'^admin/', include(admin.site.urls)),
)
```

`pluginloader.autodiscover()` will autoload any module named `devilry_plugin` in any application in `INSTALLED_APPS`.

Your first plugin

Create a file named `mysite/myplugin/devilry_plugin.py`, and put the following code into the file:

```
print
print "Hello plugin world!"
print
```

Start the development server with `python manage.py runserver`, go to http://localhost:8000/ and you should see the message you printed in the terminal/shell running the server.

Plugin errors

`pluginloader.autodiscover()` will fail if you have any errors in your `devilry_plugin`-module. It will not auto-reload failed modules before you restart the server.

Devilry localization/internationalization/translation

Devilry uses the Django localization platform/system. This means that:

- Our translations are in the gettext `.po` format.
- We mark translation strings in Python code, templates and in JavaScript.

For the actual translation process, we use transifex.com.

Warning: Pushing files to Transifex requires you to be part of the Devilry core developer team.
If you are a normal developer and not responsible for managing translations, you just need to make sure to mark all trings for translation as described in the Django docs.
If you are a translator, you only need to ask for permission to the translation catalogs for you languages in our Transifex project, and none of the information in these docs should concern you.

How we organize the translations

All translations are added to `devilry/locale/`. We do not add translation per app for the following reasons:

- There are lots of overlapping translation strings.
- Easier to upload and maintain a single translation catalog on Transifex.

Configure Transifex

Before you can start pushing and pulling translation files to/from Transifex, you will need to create a `~/.transifexrc`. It should look like this:

```
[https://www.transifex.com]
hostname = https://www.transifex.com
username = myuser
password = supersecret
token =
```

More information here: http://docs.transifex.com/developer/client/config.

Translation process

We translate using Transifex. This means that the workflow is:

1. Mark new translations or change existing translations.
2. Build the translation files (`.po` files).
3. Push translation files (`.po` files) to Transifex.

4. Wait for translators to translate using Transifex.
5. Pull translation files (`.po` files) from Transifex.
6. Compile translations and commit the `.mo` files.

Below we go in detail for each of these steps. All commands assume the following:

```
$ cd /path/to/reporoot
$ workon devilry-django
```

Mark new translations or change existing translations Read the Django internationalization docs.

Build the translation files First, make sure you have the latest po-files from transifex:

```
$ tx pull
```

We have a fabric task for that:

```
$ fab makemessages
```

Commit the changes to the `.po`-files in `devilry/locale/`.

Push translation files to Transifex Run:

```
$ tx push -s -t
```

to push the .po files to transifex.

Compile translations and commit the .mo files We have a fabric task for compiling the translations:

```
$ cd /path/to/reporoot
$ workon devilry-django
$ fab compilemessages
```

This should change some `.mo`-files in `devilry/locale/`. Commit those files.

JavaScript — Libraries and guidelines/code style

Most of our UIs are developed in JavaScript using the ExtJS4 framework from Secha (http://sencha.com).

Libraries

At the time of writing, we only use ExtJS4. We are open to including more libraries, but we have not had the need yet.

Guidelines and code style

Indent 4 spaces

Naming Real meaningful names like:

```
var age = 10;
var username_to_name_map = {peterpan: 'Peter', wendy: 'Wendy'};
```

NOT:

```
var a = 10;
var u = {peterpan: 'Peter', wendy: 'Wendy'};
var usrmap = {peterpan: 'Peter', wendy: 'Wendy'};
```

Private methods and functions Same format as semi-private python methods/functions (prefix by _):

```
_my_private_method: function() {
    return null;
}
```

Code format Should pass without any errors from JSHint (see *JSHint*).

Code layout The ExtJS app layout. See the `devilry_subjectadmin` app and the ExtJS4 docs.

Documentation Use the JSDuck format (https://github.com/senchalabs/jsduck). Note that you do not have to document every single function, but you should at least document:

- Functions, methods and variables used outside its context (I.E.: you do not have to document view-functions that is only used by its controller, but you have to document it of multiple controllers use it).
- Properties and config parameters for ExtJS classes.
- Events for ExtJS classes, especially if they are used outside their controller.

File naming Name controllers by what the control (E.g: `controller/period/PeriodController.js`), and the views after their purpose (E.g.: `view/period/PeriodOverview.js`). Try to use unique names instead of generic names like `Overview.js`. To see why, try to find (quick open) a file with tens of matches in an IDE like PyCharm or Eclipse that only search for file names, not for folder names (hint: it is not quick to open such files). We learned this when developing `devilry_subjectadmin` with controllers and views named `Overview.js`.

JSHint

For info about JSHint, see http://www.jshint.com/.

Install Install NodeJS and Node Package Manager (part of NodeJS):

- Ubuntu: `sudo apt-get install nodejs npm`
- OSX with homebrew: `brew install npm`
- Others, see: http://nodejs.org/

Install JSHint in `/usr/local` on most nix systems, like Linux and OSX:

```
$ sudo npm install jshint -g
```

Usage Simply point JSHint at a directory:

```
$ jshint src/devilry_subjectadmin/devilry_subjectadmin/static/devilry_subjectadmin/app/
```

The defaults are sane (unlike JSLint), so you should not need to supply any options.

Building the ExtJS javascript apps

Note: This is only needed if you have made changes to javascript sources, or if you are making your own ExtJS app.

Building We use webpack for building javascript. Go into the static directory of the app, where package.json and webpack.develop.js is, and run `npm run jsbuild` to build for development, and `npm run jsbuild-production` to build for production. If this is the first time you build javascript for the app, you must run `npm install` first.

Example:

```
$ cd devilry/devilry_subjectadmin/static/devilry_subjectadmin
$ npm install
$ npm run jsbuild
```

During development, you should use:

```
$ npm run jsbuild-watch
```

When the code is stable, you should build for production with:

```
$ npm run jsbuild-production
```

and commit the changes to production.js and production.js.map

Testing a production build Change the `EXTJS4_DEBUG` setting to `False` in `devilry/project/develop/settings/develop.py`. This should make all the javascript views serve `production.js` instead of `debug.js`.

Update old sencha tools app to build with Webpack

Run the following management command:

```
$ python manage.py make_require_statements_from_jsb3 <appname> devilry/<appname>/static/<appname>/app
$ ... E.g.: ...
$ python manage.py make_require_statements_from_jsb3 devilry_nodeadmin devilry/devilry_nodeadmin/stat
```

This will create an `entry.js` file with `require` statements for all the required files extracted from the `app.jsb3` file.

Copy the `webpack.develop.config.js` and `webpack.production.config.js` files from `devilry/devilry_nodeadmin/static/devilry_nodeadmin` into the app. Update the `package.json` file to contain the weback requirements and scripts from `devilry/devilry_nodeadmin/static/devilry_nodeadmin/package.json`

You should now be able to follow the building guide above. You should now run both `npm run jsbuild` and `npm run jsbuild-production`, and commit the generated `debug.js`, `production.js` and `production.js.map`.

The last thing you need to do is to make the view that serves the javascript to inherit from `devilry.devilry_extjsextras.views.DevilryExtjs4AppView` instead of from `Extjs4AppView`. You should not need to make any other changes, just switch the superclass of the view.

If the javascript builds, and you have changed the superclass of the view, you should now be able to test the code in your browser. Make sure to check the network tab in chrome developer tools to ensure that the view serves:

- `debug.js` instead of `app.js` with the `EXTJS4_DEBUG` setting set to `True`.
- `production.js` instead of ``app-all.js`` with the `EXTJS4_DEBUG` setting set to `False`.

When you have verified that both development and production builds work, you can remove:

- `app-all.js`
- `app.jsb3`
- `all-classes.js`

(they are all replaced by webpack + entry.js).

Apps

`devilry_subjectadmin` — Subject administrator GUI

About the app

The *subjectadmin* app provides a GUI for administrators on Subject, Period and Assignment, including a dashboard.

JavaScript

The application is mostly written in JavaScript as an ExtJS app.

`devilry_qualifiesforexam`

Database models, APIs and UI for qualifying students for final exams.

UI workflow

How users are qualified for final exam i plugin-based. The subject/period admin is taken through a wizard with the following steps/pages:

1. **If no configuration exists for the period:** List the title and description of each plugin (see *Plugins* below), and let the user select the plugin they want to use. The selection is stored in `QualifiesForFinalExamPeriodStatus.plugin`.

   If a configuration exists for the period: Show the overview of the semester (basically the same as the preview described as page 3 below). Includes a button to change the configuration. Clicking this button will show the list of plugins, just like when no configuration exists, with the previously used plugin selected. The *change*-button is only available on active periods.

2. Completely controlled by the plugin. May be more than one page if that should be needed by the plugin. The plugin can also just redirect directly to the next page if it does not require any input from the user. We supply a box with save and back buttons that should be the same for all plugins.
3. Preview the results with the option to save or go back to the previous page.

Plugins

A plugin is a regular Django app. Your best source for a simple example is the `devilry_qualifiesforexam_approved`-module which contains two plugins. You will find the package in the `src/`-directory of the devilry repository.

The role of the plugin A plugin is basically one or more Django views that, for the qualifies-for-exam system, acts like a black box with the following input and output:

- The input is a dict store by the qualifies-for-exam system in the users session (`request.session`):

  `periodid` The ID of the class:*devilry.apps.core.models.Period*.

  `pluginsessionid` An ID that is generated by the qualifies-for-exam system. It is used to ensure that we do not get session key collisions when using the wizard from multiple browser windows at the same time.

- The output is a `devilry_qualifiesforexam.pluginhelpers.PreviewData`-object stored in the users session (`request.session`) under the `qualifiesforexam-<pluginsessionid>` key. The output object is used by the REST-api that generates the preview-data.

Registering an app as a qualifiesforexam plugin Add something like the following to `yourapp/devilry_plugin.py`:

```
from devilry_qualifiesforexam.registry import qualifiesforexam_plugins
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _

qualifiesforexam_plugins.add(
    id='myapp',
    url=reverse('myapp-myplugin'), # The url of the view to use for step/page 2 in the workflow - the
    title=_('My plugin'),
    description=_('Does <strong>this</strong> and <em>that</em>.')
)
```

Create the view See *Plugin helpers* and take a look at the sourcecode for `devilry_qualifiesforexam_approved` (in the `src/` directory of the Devilry sources).

Configure available plugins Available plugins are configured in `settings.DEVILRY_QUALIFIESFOREXAM_PLUGINS`, which is a list of plugin ids. Note that the apps containing the plugin must also be in `settings.INSTALLED_APPS`, and the urls must be registered. The plugins are shown in listed order on page 1 of the wizard described in the *UI workflow*.

Note: You can safely remove plugins from `settings.DEVILRY_QUALIFIESFOREXAM_PLUGINS`. They will simply not be available in the list of plugins in the *UI workflow*.

Write tests If you want your plugin to be considered for inclusion in Devilry you will have to write good tests. These plugins handle very sensitive data, so it would be madness to deploy them in production without proper tests. We provide a helper-mixin for tests, `devilry_qualifiesforexam.pluginhelpers.QualifiesForExamPluginTestMixin`, which you should use. See the `tests`-module in `devilry_qualifiesforexam_approved` for examples.

Plugin helpers

The mixin classes `QualifiesForExamPluginViewMixin` is a mixin class that simplifies the common tasks for all plugin views (getting input and setting output).

Basic usage Basic usage of the class turns the input and output steps described in *The role of the plugin* into two methods: `get_plugin_input_and_authenticate()`, `save_plugin_output()`. Those two methods greatly simplify writing plugins. For example, we can create a view like this:

```
from django.views.generic import View
class MyPluginView(View, QualifiesForExamPluginViewMixin):
    def post(self, request):
        try:
            self.get_plugin_input_and_authenticate()
        except PermissionDenied:
            return HttpResponseForbidden()
        # Your code to detect passing students
        passing_relatedstudentsids = [1,2,3]
        self.save_plugin_output(passing_relatedstudentsids)
        return HttpResponseRedirect(self.get_preview_url())
```

A more complete example The example above is very simple. You will usually have to iterate over all the students in a period to find out who qualifies:

```
from django.views.generic import View
from devilry_qualifiesforexam.pluginhelpers import PeriodResultsCollector
from devilry_qualifiesforexam.pluginhelpers import QualifiesForExamPluginViewMixin

class MyPeriodResultsCollector(PeriodResultsCollector):
    def student_qualifies_for_exam(self, aggregated_relstudentinfo):
        # Test if the student in the AggreatedRelatedStudentInfo qualifies.
        # Typically something like this (all students must pass all assignments):
        for assignmentid, grouplist in aggregated_relstudentinfo.assignments.iteritems():
            feedback = grouplist.get_feedback_with_most_points()
            if not feedback or not feedback.is_passing_grade:
                return False
        return True

class MyPluginView(View, QualifiesForExamPluginViewMixin):
    def post(self, request):
        try:
            self.get_plugin_input_and_authenticate()
        except PermissionDenied:
            return HttpResponseForbidden()
        # Your code to detect passing students
        passing_relatedstudentsids = MyPeriodResultsCollector().get_relatedstudents_that_qualify_for_
        self.save_plugin_output(passing_relatedstudentsids)
        return HttpResponseRedirect(self.get_preview_url())
```

class `devilry_qualifiesforexam.pluginhelpers.`**`QualifiesForExamPluginViewMixin`**

`periodid`
The ID of the period — set by `get_plugin_input()`.

`period`
The period object loaded using the `django.shortcuts.get_object_or_404()` — set by `get_plugin_input()`.

`pluginsessionid`
The pluginsessionid described in *The role of the plugin* — set by `get_plugin_input()`.

`get_plugin_input_and_authenticate``()`
Reads the parameters (periodid and pluginsessionid) from the querystring and store them as in the following instance variables: `periodid`, `period`, `pluginsessionid`.

Raise `django.core.exceptions.PermissionDenied` if the request user is not administrator on the period.

`save_plugin_output``(*args, **kwargs)`
Shortcut that saves a `PreviewData` in the session key generated using `create_sessionkey()`. Args and kwargs are forwarded to `PreviewData`.

`save_settings_in_session``(settings)`
Save settings in the session. You get this back as an argument to your `post_statussave`-handler if your plugin is configured with `uses_settings=True`.

`get_preview_url``()`
Get the preview URL - the URL you must redirect to after saving the output (`save_plugin_output()`) to proceed to the preview.

`get_selectplugin_url``()`
Get the preview URL - the URL you should navigate to when users select *Back* from your plugin view.

`redirect_to_preview_url``()`
Returns a `HttpResponseRedirect` that redirects to `get_preview_url()`.

Helper for unit tests

class `devilry_qualifiesforexam.pluginhelpers.`**`QualifiesForExamPluginTestMixin`**
Mixin-class for test-cases for plugin-views (the views that typically inherit from `QualifiesForExamPluginViewMixin`). This class has a couple of helpers that simplifies writing tests, and some unimplemented methods that ensure you do not forget to write permission tests.

Note: If you use this class as base for your tests, your chances of getting a plugin approved for inclusion as part of Devilry is greatly increased. You have to include at least one test in addition to the unimplemented tests, a test that uses a realistic dataset to make sure your plugin behaves as intended (E.g.: Approves/disapproves the expected students). You may need more than one extra test if your plugin is complex.

`testhelper`
A `devilry.apps.core.testhelper.TestHelper`-object which is required for `create_feedbacks()` and `create_relatedstudent()` to work.

Typcally created with something like this in `setUp`:

```python
from django.test import TestCase
from devilry.apps.core.testhelper import TestHelper

class TestMyPluginView(TestCase, QualifiesForExamPluginTestMixin):
    def setUp(self):
        self.testhelper = TestHelper()

        # Create:
        # - the uni-node with ``uniadmin`` as admin
        # - the uni.sub.p1 period with ``periodadmin`` as admin.
        # - the a1 and a2 assignments within ``p1``, with separate groups on each
        #   assignment for student1 and student2, and with examiner1 as examiner.
        # - a deadline on each group
        self.testhelper.add(nodes='uni:admin(uniadmin)',
            subjects=['sub'],
            periods=['p1:admin(periodadmin):begins(-3):ends(6)'],
            assignments=['a1', 'a2'],
            assignmentgroups=[
                'gstudent1:candidate(student1):examiner(examiner1)',
                'gstudent2:candidate(student2):examiner(examiner1)'],
            deadlines=['d1:ends(10)']
        )
```

period
The period you use in your tests. Needs to be set in the `setUp`-method for `create_relatedstudent()` to work. Typically defined with the following code after the core in the example in `testhelper`:

```
self.period = self.testhelper.sub_p1
```

create_relatedstudent(*username*)
Create and return a related student on the `period`. A user with the given username is created if it does not exist.

create_feedbacks(*feedbacks):
Create feedbacks on groups from the given list of `feedbacks`.

Parameters feedbacks – Each item in the arguments list is a `(group, feedback)` tuple where `group` is the `devilry.apps.core.models.AssignmentGroup`-object that it to be given feedback, and `feedbacks` is a dict with attributes for the `devilry.apps.core.models.StaticFeedback` with the following keys:

grade See `devilry.apps.core.models.StaticFeedback.grade`.
points See `devilry.apps.core.models.StaticFeedback.points`.
is_passing_grade See `devilry.apps.core.models.StaticFeedback.is_passing`

A delivery to save the feedback on is created automatically, so all that is needed of the groups is an examiner, a candidate and a deadline.

Example:

```
self.create_feedbacks(
    (self.testhelper.sub_p1_a1_gstudent2, {'grade': 'B', 'points': 86, 'is_passing_grade':
    (self.testhelper.sub_p1_a2_gstudent2, {'grade': 'A', 'points': 97, 'is_passing_grade':
)
```

test_perms_as_periodadmin()
Must be implemented in subclasses.

test_perms_as_nodeadmin()
Must be implemented in subclasses.

test_perms_as_superuser
Must be implemented in subclasses.

test_perms_as_nobody
Must be implemented in subclasses.

test_invalid_period
Must be implemented in subclasses.

Other helpers

class devilry_qualifiesforexam.pluginhelpers.**PreviewData**(*passing_relatedstudentids*)
Stores the output from a plugin. You should not need to use this directly. Use `QualifiesForExamPluginViewMixin.save_plugin_output()` instead.

Parameters passing_relatedstudentids – See `passing_relatedstudentids`.

passing_relatedstudentids
List of the IDs of all `devilry.apps.core.models.RelatedStudent` that qualifies for final exams according to the plugin that generated the data.

devilry_qualifiesforexam.pluginhelpers.**create_sessionkey**(*pluginsessionid*)
Generate the session key for the plugin output as described in *The role of the plugin*. You should not need to use this directly. Use `QualifiesForExamPluginViewMixin.get_plugin_input_and_authenticate()` instead.

Plugins shipped with Devilry

devilry_qualifiesforexam_approved TODO

Database models

How the models fit together Each time a periodadmin qualifies students for final exams, even when they only partly qualify their students, a new `Status`-record is saved in the database. A status has a ForeignKey to `devilry.apps.core.models.Period`, so the last saved Status is the active qualified-for-exam status for a Period.

Each time a `Status` is saved, all of the `devilry.apps.core.models.RelatedStudent`s for that period gets a :class:`.QualifiesForFinalExam‘-record, which saves the qualifies-for-exam status for the student. When a status is `almostready`, we use `NULL` in the `QualifiesForFinalExam.qualifies`-field to indicate students that are not ready.

Node administrators or systems that intergrate with Devilry uses `Status.exported_timestamp` to mark `Status`-records that have been exported to an external system. It is important to note that we export statuses, not periods. This means that we can create new statuses, and re-export them. An automatic system can check timestamps to handle status changes, and the Node admin UI can show/hilight periods with exported statuses and more recent statuses.

`DeadlineTag` is used to organize periods by the time when they should have made a `ready`-`Status`.

The models

class devilry_qualifiesforexam.models.**DeadlineTag**
A deadlinetag is used to tag `devilry.apps.core.models.Period`-objects with a timestamp and an optional tag describing the timestamp.

timestamp
Database field containing the date and time when a period admin should be finished qualifying students for final exams.

tag
A tag for node-admins for this deadlinetag. Max 30 chars. May be empty or `null`.

class `devilry_qualifiesforexam.models.`**`PeriodTag`**

This table is used to create a one-to-many relation from `DeadlineTag` to `devilry.apps.core.models.Period`.

`deadlinetag`
Database foreign key to the `DeadlineTag` that the Period should be tagged by.

`period`
Database foreign key to the `devilry.apps.core.models.Period` that this tag points to.

class `devilry_qualifiesforexam.models.`**`Status`**

Every time the admin updates qualifies-for-exam on a period, we save new object of this database model.

This gives us a history of changes, and it makes it possible for subject/period admins to communicate simple information to whoever it is that is responsible for handling examinations.

`period`
Database foreign key to the `devilry.apps.core.models.Period` that the status is for.

`exported_timestamp`
Database datetime field that tells when the status was exported out of Devilry to an external system. This is `null` if the status has not been expored out of Devilry.

`status`
Database char field that accepts the following values:

- `ready` is used to indicate the the entire period is ready for export/use.
- `almostready` is used to indicate that the period is almost ready for export/use, and that the exceptions are explained in the `message`.
- `notready` is used to indicate that the period has no useful data yet. This is typically only used when the period used to be *ready* or *almostready*, but had to be retracted for a reason explained in the status

`createtime`
Database datetime field where we store when we added the status.

`message`
Database field with an optional message about the status change.

`user`
Database foreign key to the user that made the status change.

`plugin`
Database char field that stores the id of the plugin (see *Plugins*) that was used to change the status.

class `devilry_qualifiesforexam.models.`**`QualifiesForFinalExam`**

`relatedstudent`
Database one-to-one relation to `devilry.apps.core.models.RelatedStudent`.

`qualifies`
Boolean database field telling if the student qualifies or not. This may be `None` (`NULL`), if the status is `almostready`, to mark students as not ready for export.

`status`
Foreign key to a `QualifiesForFinalExamPeriodStatus`.

`devilry_gradingsystem` — The devilry grading system plugin architecture

How we configure the grading system on an assignment

1 - Select a grading system plugin. User selects one of the plugins in the devilry.devilry_gradingsystem.pluginregistry.gradingsystempluginregistry.

2 - Configure the grading system plugin User configures the grading system using the view pointed to by devilry.devilry_gradingsystem.pluginregistry.GradingSystemPluginInterface.get_configuratio

Note: This step is skipped unless the plugin has set requires_configuration to True

3 - Configure the maximum number of points possible User sets the maximum number of points possible.

Note: Plugins can opt out of this step by setting sets_max_points_automatically to True

4 - Choose how students are graded

The user selects one of the possible values for `devilry.apps.core.models.Assignment.points_to_grade_mapper`

- Passed failed
- Raw points
- Custom table

5 - Configure the points to grade mapping table (only if `custom-table`) If the user selected custom-table, they need to setup that table.

6 - Configure required points to pass The user selects the number of points required to pass the assignment (devilry.apps.core.models.Assignment.passing_grade_min_points). How they do this depends on the points_to_grade_mapper:

- If raw-points or passed-failed: Select a number of points between 0 and max_points, including both ends.
- If custom table: Select a grade from the table.

Note: Plugins can opt out of this step by setting sets_passing_grade_min_points_automatically)

Creating a Plugin

A grading system plugin must implement the devilry.devilry_gradingsystem.pluginregistry.GradingSystemPlu and it must register the implemented class with devilry.devilry_gradingsystem.pluginregistry.gradingsystempl

Please refer to one of the simple grading system plugins, such as devilry_gradingsystemplugin_points, for a starting point for implementing your own plugin.

API

class devilry.devilry_gradingsystem.pluginregistry.**GradingSystemPluginInterface**(*assignment*)

Bases: object

Interface for new grading system plugins. Makes the grading system plugin ready for the global registry.

This interface must be implemented by each grade plugin in the running system. this holds all the global information necessary to be able to manage the grade plugin layout and to cover smooth transition between different grade plugins on different Assignments

`id` = **None**
: The ID of the registry. Should be a unique string, typically the python path of the module implementing the plugin. This attribute MUST be overidden by each plugin.

`title` = **None**
: The title of the plugin. Should be a short title, and it should be translated.

`description` = **None**
: The description of the plugin. Should be translated. Shown with css `white-space:pre-wrap`.

`requires_configuration` = **False**
: `True` if the plugin require configuration before it can be used. If a plugin sets this to `True`, `is_configured_correctly()` and `get_configuration_url()` must be overridden.

`sets_passing_grade_min_points_automatically` = **False**
: `True` if the plugin sets `devilry.apps.core.models.Assignment.passing_grade_min_points` automatically. If this is `True`, the plugin must implement `get_passing_grade_min_points()`.

`sets_max_points_automatically` = **False**
: `True` if the plugin sets `devilry.apps.core.models.Assignment.max_points` automatically. If this is `True`, the plugin must implement `get_max_points()`.

`get_passing_grade_min_points``()`
: Get the value for `devilry.apps.core.models.Assignment.passing_grade_min_points` for this assignment.

  MUST be implemented when `sets_passing_grade_min_points_automatically` is `True`.

`get_max_points``()`
: Get the value for `devilry.apps.core.models.Assignment.max_points` for this assignment.

  MUST be implemented when `sets_max_points_automatically` is `True`.

`is_configured``()`
: Is the plugins configured in a manner that makes it ready for use on this assignment.

  MUST be implemented if `requires_configuration` is `True`.

`get_configuration_url``()`
: Get the configuration URL for this plugin for this assignment.

  MUST be implemented if `requires_configuration` is `True`.

`get_edit_feedback_url``(`*deliveryid*`)`
: Get the feedback editing URL for this plugin for the given `deliveryid`.

  Parameters **deliveryid** – The ID of the delivery to provide feedback for.

`get_bulkedit_feedback_url``(`*assignmentid*`)`
: Get the feedback editing URL for this plugin for the given `assignmentid`.

  Parameters **assignmentid** – The ID of the delivery to provide feedback for.

exception `devilry.devilry_gradingsystem.pluginregistry.`**`GradingSystemPluginNotInRegistryError`**
: Bases: `exceptions.Exception`

  Raised by `GradingSystemPluginRegistry.get()` when a plugin that is not in the registry is requested.

exception `devilry.devilry_gradingsystem.pluginregistry.`**`NotGradingSystemPluginError`**
: Bases: `exceptions.Exception`

Raised by `GradingSystemPluginRegistry.add()` when adding a plugin that is not a subclass of `GradingSystemPluginInterface`.

class `devilry.devilry_gradingsystem.pluginregistry.`**`GradingSystemPluginRegistry`**
Bases: `object`

Global Registry for grading system plugins.

This registry holds information on each grading system plugin the current setup uses.

The registry is used to decouple providing points for grades from the rest of the grading framework.

`add`(*registryitemcls*)
Add a plugin to the registry.

Parameters **registryitemcls** – A subclass of `GradingSystemPluginInterface`.

Raises NotGradingSystemPluginError If `registryitemcls` is not a subclass of `GradingSystemPluginInterface`.

`get`(*id*)
Get a grading plugin API class by its ID.

Raises GradingSystemPluginNotInRegistryError If the plugin is not found in the registry.

`iter_with_assignment`(*assignment*)
Returns an iterator over instances of all the plugins in the registry. Each instance is constructed with the given `assignment` as their first and only argument.

`devilry.devilry_gradingsystem.pluginregistry.`**`gradingsystempluginregistry`** **= <devilry.devilry_gradin**
The grading system plugin registry. An instance of `GradingSystemPluginRegistry`. Plugins register themselves through this instance.

`devilry.devilry_search` — Search for Devilry

This app provides a search API for Devilry.

How we handle object level permissions

We maintain a list of `admin_ids` on Node, Subject, Period, Assignment and AssignmentGroup. On AssignmentGroup, we also maintain a list of `examiner_ids` and `student_ids`. When we perform a search, we filter on these ids (the requesting user must be in an id-list). I.E:

> When we search for assignments, we first filter on `admin_ids=request.user.id`, then we perform the search.

Protection of anonymous data

We do not include any sensitive data in the main search index:

- No student names on anonymous assignments — Examiners should not be able to search for these because they are only supposed to know the candidate ID.
- No examiner names on anonymous assignments — Students should not be able to know who their examiner is.
- Tags — Only examiners and admins are supposted to see tags.

This is handled in the `devilry.apps.core.search_indexes.AssignmentGroupIndex`, and the exclusions is handled by the text-template in the `search/indexes/core/assignmentgroup_text.txt` template (located in `devilry/apps/core/templates/`).

We include the excluded data in their own fields in `AssignmentGroupIndexes`. The fields, `examiners`, `tags` and `candidates`, may be used to search for the excluded terms.

Limitations

We do not currently use the excluded fields mentioned in the previous section in the search API. This means that it is:

- not possible to search for AssignmentGroups by username or examiner on anonymous assignments.
- not possible to search for AssignmentGroups by tags.

`devilry.devilry_theme` — The Devilry theme

Warning: The `devilry.devilry_theme` app is deprecated. Use `devilry.devilry_theme2`.

ExtJS apps

ExtJS apps should use the `extjs4.views.Extjs4AppView` from django_extjs4:

```
from django.utils.translation import ugettext as _
from extjs4.views import Extjs4AppView

class AppView(Extjs4AppView):
    template_name = "devilry_examiner/app.django.html"
    appname = 'devilry_examiner'
    title = _('Myapp') # The initial title until you set one in your app
```

Writing ExtJS apps is out of scope of this guide. The code above will give you a view that you can add to your `urls.py`. You have to put your `app.js` in `static/myapp/app.js`, and it will just work. Take a look at `devilry_student` and `devilry_subjectadmin` for inspiration.

Normal Django apps

Normal Django apps can extend `devilry_theme/nonapptemplate.django.html` template. This will give you access to all of the bootstrap CSS, and the Devilry header at the top of your page. Most parts of the template and its parent-template can be modified by overriding blocks. See their source code for more details.

Example:

```
{% extends "devilry_theme/nonapptemplate.django.html" %}
{% load i18n %}
{% load static %}

{% block title %}{% trans "Select assignments that students must pass to qualify for final exams" %}

{% block head-pre %}
    <script type="text/javascript" src="{% static "myapp/stuff.js" %}"></script>
{% endblock %}

{% block bodyclass%}devilry_subtlebg{% endblock %}

{% block bootstrap-body %}
```

```
    <div style="margin-top: 40px;">

    </div>
{% endblock %}
```

Deprecated APIs and frameworks

devilry.apps.core.testhelper — Create core test data

Deprecated since version 1.4: Use *corebuilder — Setup devilry core data structures for tests* instead.

Example

```
from devilry.apps.core.testhelper import TestHelper

testhelper = TestHelper()

testhelper.add(nodes='uni:admin(mortend)',
         subjects=['cs101:admin(admin1,admin2):ln(Basic OO programming)',
                   'cs110:admin(admin3,admin4):ln(Basic scientific programming)',
                   'cs111:admin(admin1,damin3):ln(Advanced OO programming)'],
         periods=['fall11', 'spring11:begins(6)'])

# add 4 assignments to inf101 and inf110 in fall and spring
testhelper.add(nodes='uni',
         subjects=['cs101', 'cs110'],
         periods=['fall11', 'spring11'],
         assignments=['a1', 'a2'])

# add 12 assignments to inf111 fall and spring.
testhelper.add(nodes='uni',
         subjects=['cs111'],
         periods=['fall11', 'spring11'],
         assignments=['week1', 'week2', 'week3', 'week4'])

# set up some students with descriptive names

# inf101 is so easy, everyone passes
testhelper.add_to_path('uni;cs101.fall11.a1.g1:candidate(goodStud1):examiner(examiner1).dl:ends(5)')
testhelper.add_to_path('uni;cs101.fall11.a1.g2:candidate(goodStud2):examiner(examiner1).dl:ends(5)')
testhelper.add_to_path('uni;cs101.fall11.a1.g3:candidate(badStud3):examiner(examiner2).dl:ends(5)')
testhelper.add_to_path('uni;cs101.fall11.a1.g4:candidate(okStud4):examiner(examiner2).dl:ends(5)')

testhelper.add_to_path('uni;cs101.fall11.a2.g1:candidate(goodStud1):examiner(examiner1).dl:ends(5)')
testhelper.add_to_path('uni;cs101.fall11.a2.g2:candidate(goodStud2):examiner(examiner1).dl:ends(5)')
testhelper.add_to_path('uni;cs101.fall11.a2.g3:candidate(badStud3):examiner(examiner2).dl:ends(5)')
testhelper.add_to_path('uni;cs101.fall11.a2.g4:candidate(okStud4):examiner(examiner2).dl:ends(5)')

# inf110 is an easy group-project, everyone passes
testhelper.add_to_path('uni;cs110.fall11.a1.g1:candidate(goodStud1,goodStud2):examiner(examiner1).dl
testhelper.add_to_path('uni;cs110.fall11.a1.g2:candidate(badStud3,okStud4):examiner(examiner2).dl.en

testhelper.add_to_path('uni;cs110.fall11.a2.g1:candidate(goodStud1,goodStud2):examiner(examiner1).dl
testhelper.add_to_path('uni;cs110.fall11.a2.g2:candidate(badStud3,okStud4):examiner(examiner2).dl.en
# inf111 is hard! Everyone passes week1
testhelper.add_to_path('uni;cs111.fall11.week1.g1:candidate(goodStud1):examiner(examiner3).dl:ends(5)
testhelper.add_to_path('uni;cs111.fall11.week1.g2:candidate(goodStud2):examiner(examiner3).dl:ends(5)
testhelper.add_to_path('uni;cs111.fall11.week1.g3:candidate(badStud3):examiner(examiner3).dl:ends(5)'
testhelper.add_to_path('uni;cs111.fall11.week1.g4:candidate(okStud4):examiner(examiner3).dl:ends(5)')

# and 2
testhelper.add_to_path('uni;cs111.fall11.week2.g1:candidate(goodStud1):examiner(examiner3).dl:ends(5)
testhelper.add_to_path('uni;cs111.fall11.week2.g2:candidate(goodStud2):examiner(examiner3).dl:ends(5)
testhelper.add_to_path('uni;cs111.fall11.week2.g3:candidate(badStud3):examiner(examiner3).dl:ends(5)'
testhelper.add_to_path('uni;cs111.fall11.week2.g4:candidate(okStud4):examiner(examiner3).dl:ends(5)')

# badStud4 fails at week3
testhelper.add_to_path('uni;cs111.fall11.week3.g1:candidate(goodStud1):examiner(examiner3).dl:ends(5)
testhelper.add_to_path('uni;cs111.fall11.week3.g2:candidate(goodStud2):examiner(examiner3).dl:ends(5)
testhelper.add_to_path('uni;cs111.fall11.week3.g4:candidate(okStud2):examiner(examiner3).dl:ends(5)')

# and okStud4 fails at week4
testhelper.add_to_path('uni;cs111.fall11.week4.g1:candidate(goodStud1):examiner(examiner3).dl:ends(5)
testhelper.add_to_path('uni;cs111.fall11.week4.g2:candidate(goodStud2):examiner(examiner3).dl:ends(5)

# deliveries
goodFile = {'good.py': ['print ', 'awesome']}
okFile = {'ok.py': ['print ', 'meh']}
badFile = {'bad.py': ['print ', 'bah']}

# cs101
testhelper.add_delivery('cs101.fall11.a1.g1', goodFile)
testhelper.add_delivery('cs101.fall11.a1.g2', goodFile)
testhelper.add_delivery('cs101.fall11.a1.g3', badFile)
testhelper.add_delivery('cs101.fall11.a1.g4', okFile)
testhelper.add_delivery('cs101.fall11.a2.g1', goodFile)
testhelper.add_delivery('cs101.fall11.a2.g2', goodFile)
testhelper.add_delivery('cs101.fall11.a2.g3', badFile)
testhelper.add_delivery('cs101.fall11.a2.g4', okFile)

# cs110
testhelper.add_delivery('cs110.fall11.a1.g1', goodFile)
testhelper.add_delivery('cs110.fall11.a1.g1', goodFile)
testhelper.add_delivery('cs110.fall11.a2.g2', badFile)
testhelper.add_delivery('cs110.fall11.a2.g2', okFile)

# cs111
testhelper.add_delivery('cs111.fall11.week1.g1', goodFile)
testhelper.add_delivery('cs111.fall11.week1.g2', goodFile)
testhelper.add_delivery('cs111.fall11.week1.g3', badFile)
testhelper.add_delivery('cs111.fall11.week1.g4', okFile)

# g3's delivery fails here
testhelper.add_delivery('cs111.fall11.week2.g1', goodFile)
testhelper.add_delivery('cs111.fall11.week2.g2', goodFile)
testhelper.add_delivery('cs111.fall11.week2.g3', badFile)
testhelper.add_delivery('cs111.fall11.week2.g4', okFile)

# g4's delivery fails here
testhelper.add_delivery('cs111.fall11.week3.g1', goodFile)
testhelper.add_delivery('cs111.fall11.week3.g2', goodFile)
testhelper.add_delivery('cs111.fall11.week3.g4', okFile)
```

```
# g4 fails
testhelper.add_delivery('cs111.fall11.week4.g1', goodFile)
testhelper.add_delivery('cs111.fall11.week4.g2', goodFile)

# feedbacks
#   an empty verdict defaults to max score
goodVerdict = None
okVerdict = {'grade': 'C', 'points': 85, 'is_passing_grade': True}
badVerdict = {'grade': 'E', 'points': 60, 'is_passing_grade': True}
failVerdict = {'grade': 'F', 'points': 30, 'is_passing_grade': False}

testhelper.add_feedback('cs101.fall11.a1.g1', verdict=goodVerdict)
testhelper.add_feedback('cs101.fall11.a1.g2', verdict=goodVerdict)
testhelper.add_feedback('cs101.fall11.a1.g3', verdict=badVerdict)
testhelper.add_feedback('cs101.fall11.a1.g4', verdict=okVerdict)
testhelper.add_feedback('cs101.fall11.a2.g1', verdict=goodVerdict)
testhelper.add_feedback('cs101.fall11.a2.g2', verdict=goodVerdict)
testhelper.add_feedback('cs101.fall11.a2.g3', verdict=badVerdict)
testhelper.add_feedback('cs101.fall11.a2.g4', verdict=okVerdict)

# cs110
testhelper.add_feedback('cs110.fall11.a1.g1', verdict=goodVerdict)
testhelper.add_feedback('cs110.fall11.a1.g1', verdict=badVerdict)
testhelper.add_feedback('cs110.fall11.a2.g2', verdict=goodVerdict)
testhelper.add_feedback('cs110.fall11.a2.g2', verdict=okVerdict)

# cs111
testhelper.add_feedback('cs111.fall11.week1.g1', verdict=goodVerdict)
testhelper.add_feedback('cs111.fall11.week1.g2', verdict=goodVerdict)
testhelper.add_feedback('cs111.fall11.week1.g3', verdict=badVerdict)
testhelper.add_feedback('cs111.fall11.week1.g4', verdict=okVerdict)

# g3's feedback fails here
testhelper.add_feedback('cs111.fall11.week2.g1', verdict=goodVerdict)
testhelper.add_feedback('cs111.fall11.week2.g2', verdict=goodVerdict)
testhelper.add_feedback('cs111.fall11.week2.g3', verdict=failVerdict)
testhelper.add_feedback('cs111.fall11.week2.g4', verdict=okVerdict)

# g4's feedback fails here
testhelper.add_feedback('cs111.fall11.week3.g1', verdict=goodVerdict)
testhelper.add_feedback('cs111.fall11.week3.g2', verdict=goodVerdict)
testhelper.add_feedback('cs111.fall11.week3.g4', verdict=failVerdict)

# g4 fails
testhelper.add_feedback('cs111.fall11.week4.g1', verdict=goodVerdict)
testhelper.add_feedback('cs111.fall11.week4.g2', verdict=goodVerdict)
```

TestHelper API

class `devilry.apps.core.testhelper.`**`TestHelper`**

Bases: `object`

This class helps generate test data.

`create_user`(*name*, *fullname=None*)

Create a user with the given username. Adds the user to `self.<name>`.

Returns The created user object.

reload_from_db(*obj*)
Reload the given `django.db.Model` object from the database using `obj.__class__.get(pk=obj.pk)`. Updates the cache entry on this testhelper object if the object was created using this testhelper.
Returns The object that was re-loaded from the database.

create_superuser(*name*)
Create a superuser with the given username. Adds the user to `self.<name>`.
Returns The created user object.

add_delivery(*assignmentgroup*, *files={}*, *after_last_deadline=False*, *delivered_by=None*, *successful=True*, *time_of_delivery=None*)
Parameters

- **assignmentgroup** – Expects either an AssignmentGroup object or a string path to an assignmentgroup. This is a mandatory parameter.
- **files** – A dictionary with key/values as file name and file content as described in Delivery.add_file()
- **after_last_deadline** – If True, sets time_of_delivery 1 day later than the assignmentgroups active deadline. Effectively the same as setting `time_of_delivery=1`. Ignored i `time_of_delivery` is used.
- **time_of_delivery** – Set time_of_delivery to this number of days after the active deadline. Use a negative number to add a delivery before the active deadline. Can also be a datetime.datetime object that specifies an exact timestamp.

add_feedback(*delivery=None*, *verdict=None*, *examiner=None*, *timestamp=None*, *rendered_view='This is a default static feedback'*)
Parameters

- **delivery** – Either a Delivery object or a string path to an assignmentgroup, where we take the last delivery made. This is the only mandatory parameter
- **verdict** – E dict containing grade, score and passing grade. Defaults to:

```
dict(grade='A', points=100, is_passing_grade=True)
```

- **examiner** – A User object. Defaults to the first examiner for the delivery's assignment group.
- **timestamp** – A datetime object for when the feedback was saved. Defaults to same time the delivery was made
- **rendered_view** – The rendered view of the feedback. Defaults to `"This is a default static feedback"`.

add(*nodes=None*, *subjects=None*, *periods=None*, *assignments=None*, *assignmentgroups=None*, *deadlines=None*)
Smart add.

Each attribute is normally just a list of names. The names are `short_name`, for nodeish types, and a virtual name for assignmentgroups, and deadlines.

Names can be supplemented by **extras**, which are parameters that tunes the created items. Extras are separated by colon (**:**), and each extra has the following format:

```
<name>(<args>)
```

Parameters

- **nodes** – List of nodes.
  admin Comma-separated list of admins (usernames) to add to the node.
  ln Long name of the period. Defaults to capitalize short name.
- **subjects** – List of subjects. Extras:
  admin Comma-separated list of admins (usernames) to add to the node.

ln Long name of the period. Defaults to capitalize short name.

- **periods** – List of nodes. Extras:

  admin Comma-separated list of admins (usernames) to add to the node.

  ln Long name of the period. Defaults to capitalize short name.

  begins Number of months after *now* that the period begins. Can be a negative number. Defaults to *now*.

  ends Number of months after `begins` that the period ends. Can be a negative number. Defaults to `6`.
- **assignments** – List of assignments.

  admin Comma-separated list of admins (usernames) to add to the node.

  ln Long name of the period. Defaults to capitalize short name.

  anon Should the assignment be anonymous? `true` or `false`, and defaults to `false`.

  pub Number of days after the start time of the period that the assignment should be published. Can be a negative number. Defaults to 0.

  delivery_types `electronic` or `nonelectronic`.

  first_deadline The offset of the `first_deadline` from the `publishing_time` in days. If this is `0`, we automatically add `1` second to the `publishing_time` to ensure that they are not equal.
- **assignmentgroups** – List of assignmentgroups. Extras:

  candidate Comma-separated list of candidates (usernames) to add to the group. Optionally, you can add a candidate_id by suffixing the username with `;<candidate_id>`. Example: `candidate(student0;2345,student1;5673)`

  examiner Comma-separated list of examiners (usernames) to add to the group.
- **deadlines** – List of deadlines. Extras:

  ends Number of days after the publishing_time of the deadline ends. Can be a negative number. Defaults to `10` days.

  text Deadline text.

`add_to_path`(*path*)

Splits up a dot separated path, and calls `add()` with those pieces as arguments.

`get_object_from_path`(*path*)

Get a `Node`, `Subject`, `Period`, `Assignment`, `AssignmentGroup`, `Deadline`, `Delivery` or `Feedback` that was added with `add()`, `add_feedback()`, `add_to_path()`, or `add_delivery()`.

The path does not have to contain the node path (unless you are looking up a node), since subject shortnames are unique.

`set_attributes_from_path`(*path*, ***attributes*)

Shortcut to `get_object_from_path()`, set the given attributes on the object, and call `obj.save()`.

`create_feedbacks`(**args*)

Create feedbacks on groups from the given list of `(group, feedback, delivery)`-tuples.

Parameters args – Each item in the arguments list is a `(group, feedback[, delivery])` tuple where:

`group` is the `devilry.apps.core.models.AssignmentGroup`-object that it to be given feedback

`feedbacks` is a dict with attributes for the `devilry.apps.core.models.StaticFeedback` with the following keys:

`grade` See `devilry.apps.core.models.StaticFeedback.grade`.

`points` See `devilry.apps.core.models.StaticFeedback.points`.

is_passing_grade See `devilry.apps.core.models.StaticFeedback.is_passing`

delivery Is an optional dict of files to make a delivery from. Defaults to:

```
{'test.py': ['print ', 'tst']}
```

A delivery to save the feedback on is created automatically, so all that is needed of the groups is an examiner, a candidate and a deadline.

Example:

```
self.create_feedbacks(
    (group1, {'grade': 'B', 'points': 86, 'is_passing_grade': True}),
    (group2, {'grade': 'A', 'points': 96, 'is_passing_grade': True}, {'hello.txt', ['Hello'
    (group3, {'grade': 'F', 'points': 12, 'is_passing_grade': False})
)
```

Releases

Release notes

Release Notes 1.2.1

Major changes

Semantic changes Administrators are no longer implicitly examiner. They must make themself examiner if they want to provide feedback to students. We have made it easy to make yourself examiner:

- An option when creating an assignment.
- Administrators can edit examiners on the period/semester.
- When browsing a group (student), you get a button to make yourself examiner if you are not already.

A complete rewrite of the deployment system We have split our deployment scripts (for system administrators) into a separate repository. The repository includes a Chef DevOps setup that should simplify the work of system admins greatly. It also includes a much better setup for those who do not wish to use Chef. See http://devilry-deploy.readthedocs.org/ for more information.

New subject admin UI A completely new user interface for subject (course) administrators. The UI has more or less all of the features of the old UI, but it is far more user-friendly and optimized for common task. Some highlights:

Create new assignment wizard Smart and efficient *create new assignment* wizard. The wizard sets up an assignent, adds students to the assignment and assigns examiners to the students with very little input needed.

The wizard is smart and tries to suggest values when you create assignments. It automatically suggests names of assignments based on previous assignments. So if you name your first assignment *Assignment 1*, it will suggest *Assignment 2* for your next assignment.

The wizard also autodetects regularly repeating assignments, and suggests publishing time and submission dates based on regular intervals. This means that if you have weekly deliveries, you will only have to setup the submission and publishing times on the first 2 weeks, and Devilry will suggest sane defaults for the rest. It even works if you have a break of a week or 2, because Devilry uses the most common interval for all your assignments.

For those who like to set up many assignments, the wizard has a shortcut after each assignment is created, that lets you re-run the wizard using the same settings. Combined with the autodetection described above, this means that you can setup many assignments in a very efficient manner.

Semester/period overview An overview, very similar to the one in the old UI, but it is faster and has some new features. Autodetects problems with missing students. Supports export to Office Open XML (MS Excel), CSV, JSON, XML, YAML and REST API.

Logging of all dangerous actions We log all dangerous actions in the new UI, like deletion, renaming, moving deadline, and so on. The log-records include the action performed, with IDs and names, the user who made the change, and the time the change occurred. We also log failed dangerous actions.

New system for marking qualified for final exam Far more user-friendly and plugin based, so it is easy to extend.

Edit examiners and students on semesters/periods Admins can manage their own students and examiners. They can tag students and examiners, and use those tags to automatch students and examiners on all assignments.

Deadline manager A full featured deadline manager that gives you full control over all your deadlines, and the students on each deadline. Among many other features, it supports moving deadlines (which several users have requested).

Interractive guide system A guide that stays at your right hand side and guides you through the UI. We only have a guide that helps users creating new assignments, but we will add guides when users tell us what they need help understanding.

Smarter statuses Groups (students) are no longer closed or open. Instead they have smarter statuses, like: *Waiting for feedback*, *Waiting for deliveries*, *Corrected*, and so on.

Statistics about examiners on an assignment Charts and numbers that should help admins keep track of their examiners. Please let us know if you have suggestions for more numbers or charts in this view, or if you have ideas for making it better.

New node admin UI Because the old admin interface was for both node and subject administrator, we had to make a new UI for node administrators when we replaced it. This UI is not very powerful in this release, but we plan to improve it gradually in cooperation with its users.

Release Notes 1.2.1.1

Major changes

New qualified for final exam app The qualified for final exam app has been rewritten. It now uses a very user-friendly wizard to guide users through the process, and the entire system in plugin-based, so is is relatively easy to add support for more complex scenarios than the build-in plugins support. In addition to a plugin based architecture, the new app adds some useful new features:

- Support for *almost finished*. This solves the problem that arises when just a couple of the students need more time, but you want to export the rest of the students as ready for exams.

Here is how it works for a subject/course administrator:

- The entire period/semester is marked as *almost ready for export*.
- The administrator gets a message field where they can explain the situation.
- The administrator has to select the students that is not ready for export.

And for a Node/Department administrator:

- Gets a list of un-exported periods/semesters, kind of like the TODO-list for examiners.
- Can mark periods/semesters as exported.
- If the qualified-for-final-exam status on a period/semester is changed, it re-appears in the TODO-list with information about why it has re-appeared.

For systems that want to auto-export from Devilry:

- Can get the same information as admins get via their UI via the REST API, and make smart choices based on metadata they store about the last time they exported. Devilry saves a new status each time an admin makes a change, so it should not be a problem to track changes.

Release Notes 1.3

See also:

Migration guide for sysadmins

Major changes

REST APIs We are working on migrating from `djangorestframework` version 1 to `django-simple-rest`. The reason for this is that `djangorestframework` version 2 is incompatible with version one, but they use the same namespace, which makes it hard to run them side by side. We could hack it to work, but `django-simple-rest` matches our needs better, and it is more in line with the modern Django view API. We have created the `devilry_rest` module where we keep our common REST utilities, and we have implemented public/private authentication that will make it a lot easier to program against Devilry.

Fixed HARD deadlines issues We have moved all constraint checking for HARD deadlines from the core into the only view where it makes sense to check for hard deadlines. This view is, of course, the one where users add deliveries.

This fixes a lot of edge case issues, such as examiners adding a placeholder delivery when the deadline has expired.

More details: https://github.com/devilry/devilry-django/issues/434

Tell students when they are not relatedstudent on a semester/period We add a big red message for students when they are not student on a period. This can happen when students are added to a group, and later removed from the subject/course.

More details and screenshots: https://github.com/devilry/devilry-django/issues/433

MathJaX embedded We have included MathJaX as part of the Devilry repo.

Release Notes 1.3.1

Changes This is a minor bugfix release. It fixes:

- https://github.com/devilry/devilry-django/issues/437
- https://github.com/devilry/devilry-django/issues/444
- https://github.com/devilry/devilry-django/issues/447

Release Notes 1.3.2

Warning: Does not work - see https://github.com/devilry/devilry-django/issues/463. Use 1.3.3 instead.

Release Notes 1.3.3

Features added

- Added possibilities to remove from current selection in the student overview for assignment administrator in addition to existing functionality for replacing and adding student to current selection. This makes it easier to remove student with special tags for example those who have alreade passed the assignment.
- Added an aggregated student info view for Administrators. This will help the managers to be able to easily look up and check for possible wrongly data. Devilry integrates with third-party solutions for the raw student data and flaws and erros in the data are not always automatically ruled out.

Release Notes 1.3.4

Fixes https://github.com/devilry/devilry-django/issues/464.

Release Notes 1.3.6

Fixes https://github.com/devilry/devilry-django/issues/481.

Release Notes 1.4.0

New Examiner Interface The biggest new feature in Devilry 1.4.0 is the completely overhauled and new Examiner interface. Fully responsive and mobile friendly design will make the task of correcting a lot easier.

New Grading System Plugin Architecture Along with the new examiner interface we have added a new plugin architecture for handling grade mapping. The plugins are easily integrated into Devilry with a minimal required setup for external plugins. Grade system provided are:

- Approved / Not Approved
- Points

Release Notes 1.4.10

Bugfixes:

- https://github.com/devilry/devilry-django/issues/587

Improvements/Changes

- Added programming code similarity check support.
- Made examiner overview sortable.
- Added support for more efficient approval/disapproval of one and one group for examiners.

Release Notes 1.4.11

Bugfixes:

- https://github.com/devilry/devilry-django/issues/604
- https://github.com/devilry/devilry-django/issues/605

Release Notes 1.4.12

Nothing new - just a release to fix a bug in the devilry-deploy release with the same version number.

Release Notes 1.4.13

- Fixed https://github.com/devilry/devilry-django/issues/625
- Updated Detektor to 1.1.0-beta.011 (to get the fix for https://github.com/appressoas/detektor/issues/1)

Release Notes 1.4.14

- Fixes issues with 1.4.13.

Release Notes 1.4.2

Bugfix and finetuning after 1.4.0. Fixes:

- Lots of missing translations.
- Bugs with feedback editing in bulk.
- Fixed https://github.com/devilry/devilry-django/issues/520
- Fixed https://github.com/devilry/devilry-django/issues/519
- Add documentation links right in the UI for examiners.
- More useful metadata in listings for examiners. On the frontpage, we show the number of students waiting for feedback. On the assignment overview we have added counter for each menu item and a few other small adjustments.

Release Notes 1.4.3

Bugfixes for 1.4.x. Fixes:

- Fixed https://github.com/devilry/devilry-django/issues/521
- Usable workaround for https://github.com/devilry/devilry-django/issues/520

Release Notes 1.4.4

Bugfixes:

- https://github.com/devilry/devilry-django/issues/528
- https://github.com/devilry/devilry-django/issues/524

Release Notes 1.4.4.1

Bugfixes:

- https://github.com/devilry/devilry-django/issues/531

Release Notes 1.4.4.2

Bugfix Fixed issue #533 https://github.com/devilry/devilry-django/issues/533

Release Notes 1.4.5

Added functionality for students to manage their own group composition.

Bugfixes

Release Notes 1.4.6

Bugfixes and improvements: - Cleanup lots of old examiner UI code. - https://github.com/devilry/devilry-django/issues/536 - https://github.com/devilry/devilry-django/issues/538 - https://github.com/devilry/devilry-django/issues/539 - https://github.com/devilry/devilry-django/issues/549 - https://github.com/devilry/devilry-django/issues/547

Release Notes 1.4.7

Bugfixes:

- https://github.com/devilry/devilry-django/issues/560
- https://github.com/devilry/devilry-django/issues/563

Release Notes 1.4.8

Bugfixes:

- https://github.com/devilry/devilry-django/issues/569
- https://github.com/devilry/devilry-django/issues/566

Improvements/Changes

- Making passed as the default choice in Passed/Failed plugin: https://github.com/devilry/devilry-django/issues/523
- Consistent use of "Write Feedback" to avoid confusion when mixed with "Provide Feedback"
- Made the todolist for examiner show the next forward in the list not the previous one
- Fixed timing issues with rendering on safari and chrome when there was no assignments

Release Notes 2.0.3

Note: We skipped 2.0.2 because of a forgotten update to version.json.

- Fix for https://github.com/devilry/devilry-django/issues/746.
- Fix for serialization issues with Qualifies For Exam.

Release Notes 2.0.4

- Fix for https://github.com/devilry/devilry-django/issues/892

Release Notes 2.0.5

- 2.0.x version with json dump scripts for migration to 3.0

Release Notes 2.0.6

- 2.0.x version with json dump scripts for migration to 3.0

Release Notes 2.0.7

- 2.0.x version with json dump scripts for migration to 3.0

Release Notes 2.0.8

- 2.0.x version with json dump scripts for migration to 3.0
- Removed devilry_search app

Release Notes 2.0.9

- 2.0.x version with json dump scripts for migration to 3.0
- Removed HAYSTACK_CONNECTIONS

How to release a new Devilry version

In the devilry-django repo

1. Make sure you build and commit any changed ExtJS apps (see *JavaScript — Libraries and guidelines/code style*). Make sure to test out student, examiner, course admin and department admin roles with the `EXTJS4_DEBUG=False` setting as explained in *JavaScript — Libraries and guidelines/code style*.
2. Update the version number in:

   ```
   devilry/version.json
   ```

3. Add a releasenotes document in `docs/releasenotes-X.Y.Z.rst`, and commit the new file.
4. Commit the version changes.
5. Tag the release:

   ```
   $ git tag vX.Y.Z
   ```

6. Push the changes:

   ```
   $ git push
   $ git push --tags
   ```

7. Push the changes to pypi:

   ```
   $ python setup.py sdist upload
   ```

CHAPTER 2

More help

If this documentation is lacking, or if you have problems, detect bugs, etc...

Forum, issue-tracker and contact information

Note: Devilry is truly open, not just Open Source, but we also try to keep all issues, suggestions and plans in the open. This means that your suggestions, bugs, problems, etc. is handled in the open, and readable by anyone.

Warning: **DO NOT post sensitive information**, like names of students, passwords, etc. **via any of the contact channels** listed below (see the note above for why).

Issue tracker

Visit The Devilry issue tracker.

Anyone can add issues to our issue tracker at our GitHub project page. We use the issue tracker for bugs, problems, suggested improvements, suggested new features, etc.

You need to create a GitHub user to add an issue. You just have to write an understandable title and description. We will then tag your issue, and respond to your via comments on your issue. You should be notified for each new comments on your issues by email unless you disable email notifications on GitHub.

Facebook

Visit our Facebook page.

Question and Answers forum

Visit: The Devilry Help Questions and Answers forum.

We have a Question and Answer forum on Google Groups named Devilry Help. You can post anything on this forum, including:

- help understanding Devilry — no problem is too small for this forum
- suggest improvements — even minor improvements

- report problems
- report bugs
- suggest new features

Note: We recommend that you use the issue tracker instead of this forum if you have a well defined problem or suggestion. Even very small improvements or issues belong in the issue tracker, and they end up there even if you post them in the Q&A forum. The only difference is that someone else have to put them in the issue tracker, which may delay fixing the issue.

Your local Devilry support

The local Devilry support typically adds a link to a page with their contact information in the help page. Click the question mark in the upper right corner when logged in to Devilry, and look for a link to internal/organization specific devilry documentation.

Contact email — only for special cases

Warning: This is only for contact requests that does not belong in the open contact channels, like the issue tracker, or in the Q&A forums. This mailinglist is typically for those that need private and direct contact with the Developers, and requests belonging in the other contact channels is ignored.
The warning above about sensitive information is also for this list.

devilry-contact@googlegroups.com

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Symbols