HARVESTER TEST GUIDE

This document is meant to be an explanation about how people can test certain functionality within the harvester and/or related frontends, before releasing new features to production. The guide tries to be practical and is not a detailed explanation of how the harvester works. See the datatypes.02.pdf document for a thorough explanation of the harvester system. If you're trying to troubleshoot a problem on production, read the document-troubleshoot.02.pdf instead. There you can find a step-by-step plan to debug what might have gone wrong during harvesting on production servers when processing metadata. This guide is solely focused on how to setup (mock) test situations to see if the system is functioning under those test conditions.

Types of test situations

There is not only one way to setup a test situation. Exactly which test situation you want to setup depends on your testing goals. Below is a table with all possible test scenario's. The links in this table will jump to relevant sections of this guide.

Name	**Description**	**Docs**
Frontend check	To test if the output from the harvester is used by a frontend.	link
Sharekit integration	To test new Sharekit features.	link
Harvester logic	To test how the harvester processes data.	link
Source integration	To test how external metadata providers work with the harvester.	link N/A

Frontend check tests

This test is most easily executed with Sharekit. There shouldn't be a difference between output for different sources. However if you do want to test how harvester output is used by frontend, without taking Sharekit as the source, then it's possible to adjust data for a source as done when testing harvester logic.

Edusources (through dev)

For Edusources it is possible to edit acc.sharekit and see the changes on dev.edusources. These changes propagate through webhooks and should be visible within 30 seconds of an edit.

Publinova or Edusources (through acc)

Publinova doesn't have a frontend that connects to the dev.harvester environment or sometimes the code is only deployed to acc.frontend. In these cases you can work with the following Sharekit organizations on production:

- Publinova test
- SURF edusources test

Please note that these organizations are hidden by default. You have to log into the backend of Sharekit to make them visisble. Currently it's not possible to make these organizations visible to staff only. Therefor they are hidden until we need them for testing. It would be nice if Sharekit can split the visibility of these organizations, so they are visible in the admin and invisible on public pages.

Any publications that are made for these organizations will propagate in the acc.harvester environment while prod.harvester will silently ignore them. This makes these organizations suitable as testing organizations.

Sharekit integration tests

Sometimes you want to test how new features for Sharekit will impact the harvester or frontends before deploying Sharekit to production.

For Edusources you can best test this using the same guide for testing development frontend changes. That way you both test the harvester and frontend with a a single test.

For Publinova these tests are a bit more involved. You can only test how the harvester will change its output based on the new Sharekit features. From this it logically follows how the frontend would react it it consumed that output. Testing this way is far from ideal. You can see the output that the Publinova frontend uses in the development harvester API documentation.

Harvester logic tests

The harvesting process consists of a few phases that can be described like this:

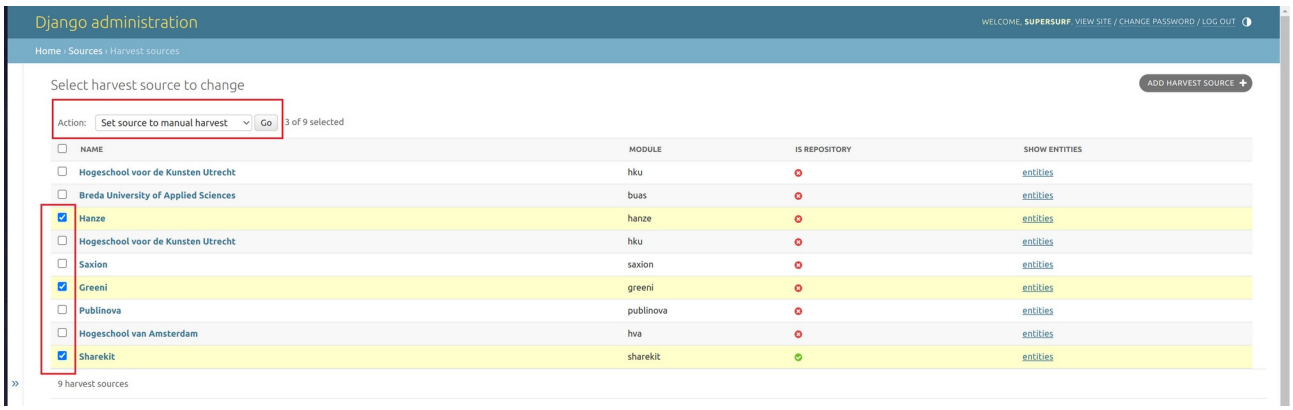
1. Extract (fetch data from sources)
2. Transform (change data format to a (internal) standard format)
3. Load (save the data into Django models, using a Postgres backend)
4. Background tasks (execute computations based on the loaded data)
5. Output (export data to search or the Publinova importer)

The first three steps (the ETL pipeline) are very source specific and therefore need to be tested using source integration tests. The last two steps are very often best tested using Sharekit integration testing. For instance if you want to test how the harvester background task for normalizing publisher years is working. Then the easiest way to test this is to adjust the publication date in Sharekit a few times and to see how normalization changes in the frontend or the harvester API output.

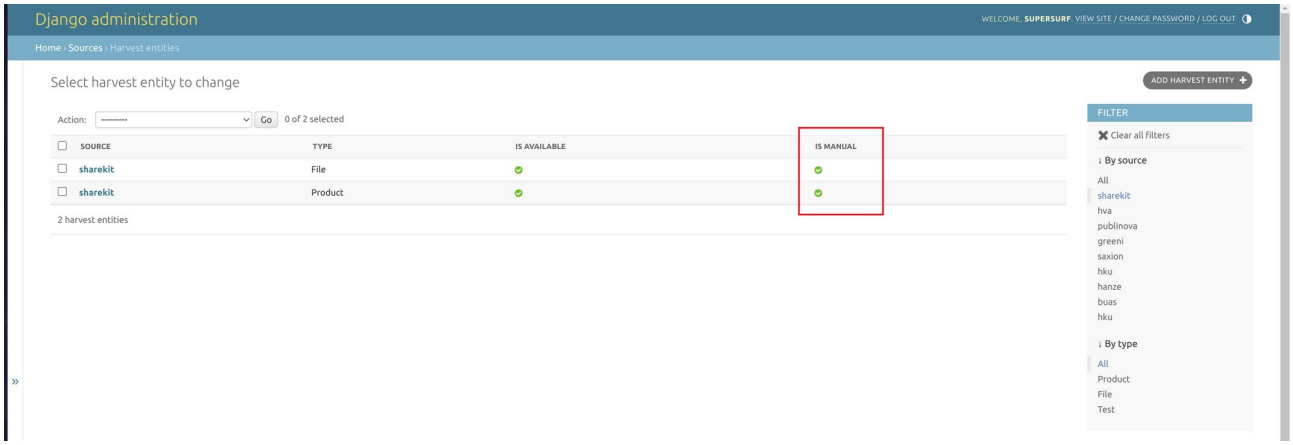
However sometimes you want to test with data that comes from other sources than Sharekit and you want to be able to change the data arbitrarily to test how data variation will impact the system. This kind of testing can be done with the “testing” app. And involves a few steps. Please note that these steps will not work for the production environment. You’ll get warnings or errors instead, because it’s not safe to manually adjust data on production.

Set sources into manual mode

By going to the “source” models in the Django admin (Publinova or Edusources). It’s possible to mark specific sources as manual. See screenshots for an example:



After clicking “go” in the screenshot above. You can click the entities link in the “show entities” column and see that the relevant entities have been set into manual mode.

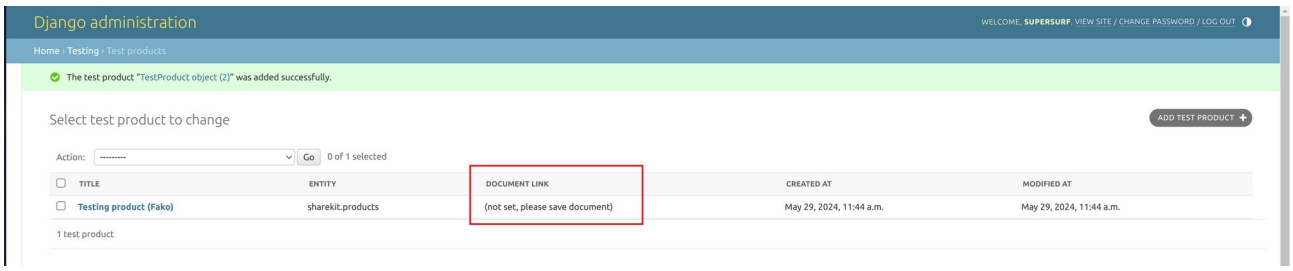


Do not forget to undo these changes by setting the sources into “automatic harvest” mode, when you’re done. However keep in mind that switching to automatic mode will delete all manual changes in the harvester data.

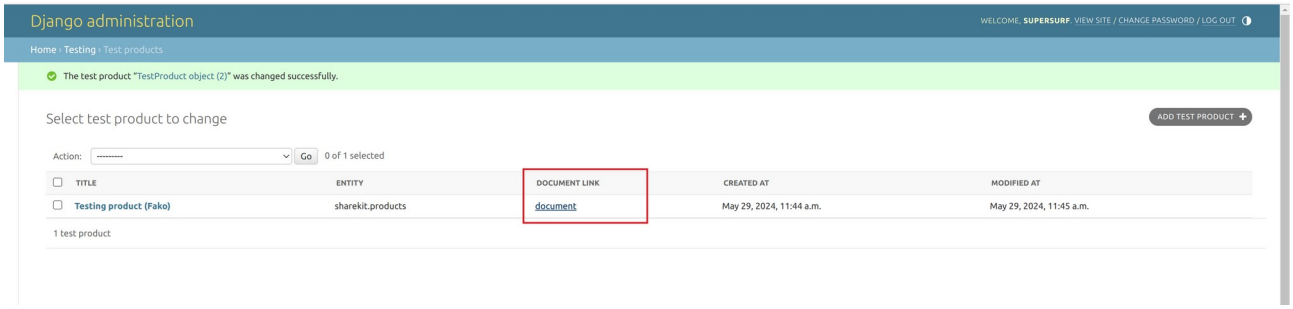
Add a TestProduct

Next you can go the the testing app in the admin (Publinova or Edusources). There you can add a product which lets you fill in a few details. Notice that you can’t change most (properties) data initially.

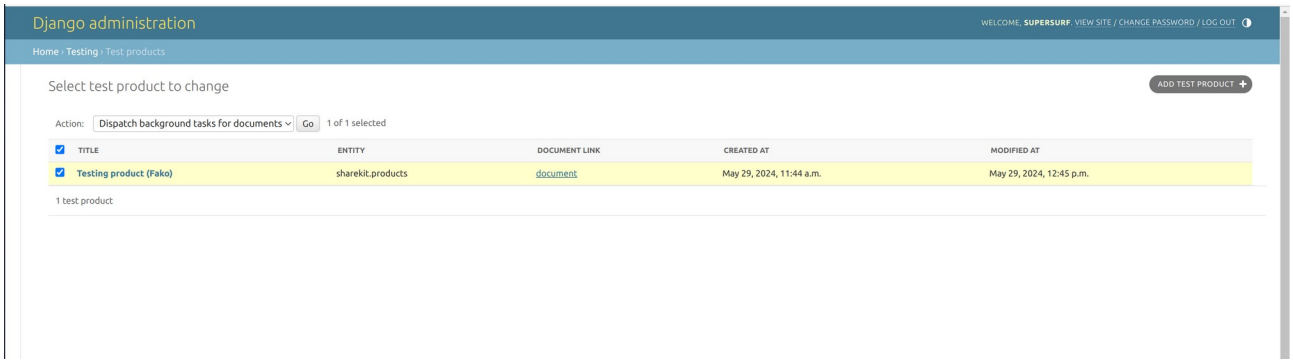
Once you click “save” you’ll see the new TestProduct in the overview. However it will be missing its Document link. The Document is the place where the harvester will actually save data. As long as that link is missing you haven’t inserted the test data yet.



If you click on the “title” you can edit the TestProduct again. This time it will be possible to edit the properties. The properties have to be valid JSON, so editing is quite error prone. Perhaps ChatGPT can help out if you are uncertain about which JSON syntax errors you’ve made and the admin is complaining about requiring “valid JSON”. The second time you save the Document will be created and a link will be available to see that Document.



Please note that these Documents are the same as Documents used in the “real” harvester. The Document troubleshooting PDF explains in more detail how you can debug these Documents. Generally the data should now show up in the frontend and background tasks for the mock Document should be executed if applicable. You can for instance try to change the “publisher_year” in the TestProduct to see that the normalize_publisher_year task executes correctly for the corresponding ProductDocument. Task execution happens whenever you save a TestProduct, but it’s also possible to trigger these from the admin using the dispatch action on the overview page. When the test data disappears (because automatic harvesting was turned back on), you can also bring multiple TestProducts back using the dispatch action (as long as the relevant sources are in manual mode once more).



Apart from TestProducts it’s also possible to add TestFiles. These are a bit more involved. Make sure that you have a TestProduct before you make a TestFile. The “files” property of the TestProduct should contain the identical URL as your TestFile. The TestFile will only show up in the output after you save the TestProduct when the TestFile has been created.

Source integration tests

Sometimes you may want to test how the harvester handles extracting, transforming and loading data from an external source. Unfortunatelly there is no easy manual method to test this. However there are a lot of automated tests around this functionality. The input data for these tests are kept outside of versioning control, because SURF may or may not declare that data to be PII (it contains author names). So this data lives in a S3 repository bucket. Most people won't have access and people who do should think twice before editing this data (it will most certainly effect the automated tests). However if there is specific case that needs testing. We can consider adding that test to the suite as long as we can get the relevant data from the source.

Another approach would be to create mock API endpoints that return the data from the S3 bucket, through regular Django views. If we point the classes responsible for extracting and transforming the data to these endpoints. It would be relatively easy to introduce new data variants. The automated tests would then check if extraction and transformations are correct for all (possibly dozens of) variants.

Final thoughts

This guide is meant for people to manually test features, before they land into production. It would be a good idea to do some of this testing automatically. An earlier document that I've written named test-strategy.pdf goes into system changes that allows for teams other than the data-engineering team to provide test data concerning test cases that these teams want to test. This could be a way for the Edusources team to write automated tests, while their application needs minimal adjustments to work with that test data. However they might prefer to mock their own data in their own way instead of the harvester providing endpoints to mock that data (as is the proposal).