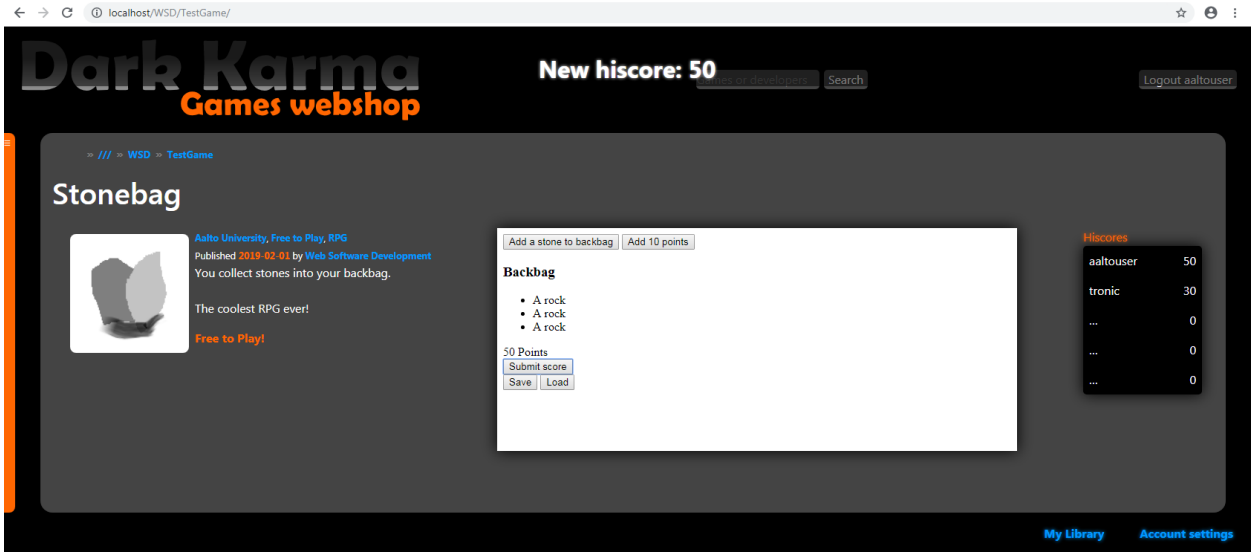


DKshop Project Report

- Lasse Kärkkäinen (61352B)

I was the sole member of this "group", as agreed beforehand with course personnel. The schedule set in the project plan was tightly followed and after about 60 hours of work on the project I have finished it one week prior to deadline, with more features than originally planned. Only minor changes to URLs and models were done to the ones specified in the project plan.

Nothing was particularly difficult, quite the contrary-- Django, modern Javascript and CSS Flex layouts were a pleasure to work with. I only had some minor annoyances, for instance with the definition of my custom upper and lower case slug types, which required investigating Django source code and required some duplication of effort with slight differences (e.g. one slug regex containing ^$ and the other without, for the same goal). Another nuisance was the incompatibility of date formats between Python, JSON and the Highchart JS library.

Set up

Check out `dkproject/settings.py` for possible deployment-related options, and in particular the settings for the payment system that need to be altered for payments to work.

Initialize the database, collect static files and run a development server

```
python manage.py migrate
python manage.py collectstatic --no-input
python manage.py runserver
```

Unless you have enabled debug mode, you need to start nginx frontend as well

```
cd nginx
nginx -c conf/nginx.conf
```

This serves the static and media files from *www* and proxies other requests to the development server. Note: uwsgi or other options might be available that I couldn't use on this Windows box. Modify nginx.conf if you cannot listen on port 80, or if the development server is on port other than 8000.

Now you may register on site and start adding games. Optionally one or more fixtures may be imported for a quick start:

```
python manage.py loaddata 1-genres  # A set of useful genres (required for other f
ixtures)
python manage.py loaddata 2-aalto   # Test games and minimal test data
python manage.py loaddata 3-bulk    # A lot of games for usability/layout testing
python manage.py loaddata 4-sales   # Dummy sales for aalto games (statistics test
ing)
```

Navigate to http://localhost/aalto/ (http://localhost/aalto/) for games of particular relevance to this course.

Accounts and permissions

Create a superuser account to modify genres, assign staff permissions etc.

```
python manage.py createsuperuser
```

Users **aaltouser** and **aaltodev** are created from **2-aalto**. Login with admin user to access the admin site and set passwords for the test accounts.

Do note that superusers and anyone with the `webshop.admin` permission (webshop|Developer|Full access to all games) have complete developer access on the web shop but only staffers can access the admin site where Django's permissions can be used for fine-grained access control.

Registering as developer and editing one's own games and account settings does not require using any admin accounts or the admin site.

Security and error handling

The site relies on iframe sandboxing, Django CSRF protection and in browsers' same-origin restrictions. Everything is escaped automatically by HTML5Tagger and by various Django facilities. No user-originated HTML or scripts are permitted on site (HTML tags display as text). The topic is too broad to cover here in more detail but there are many comments in source code where relevent.

I have not done extensive security testi but I do *dare you* to find any flaws.

No effort is done to catch all exceptions, so 500 errors will appear with deliberately malformed input. Further, Http404 is often raised where another code (403, in particular) would be more appropriate, as Django lacks suitable shortcuts. Forms are expected to do comprehensive error handling, including help messages, for any invalid data the user might accidentally input.

HTML5Tagger

Django templates and generic views were not used at all. Instead, I found it pleasant to work with a HTML5-generating library that I implemented in December. Although it was planned that possibly templating support would be implemented (or further developed from the stub that already exists), there was little need, and thus only minor changes were done to the HTML5Tagger module.

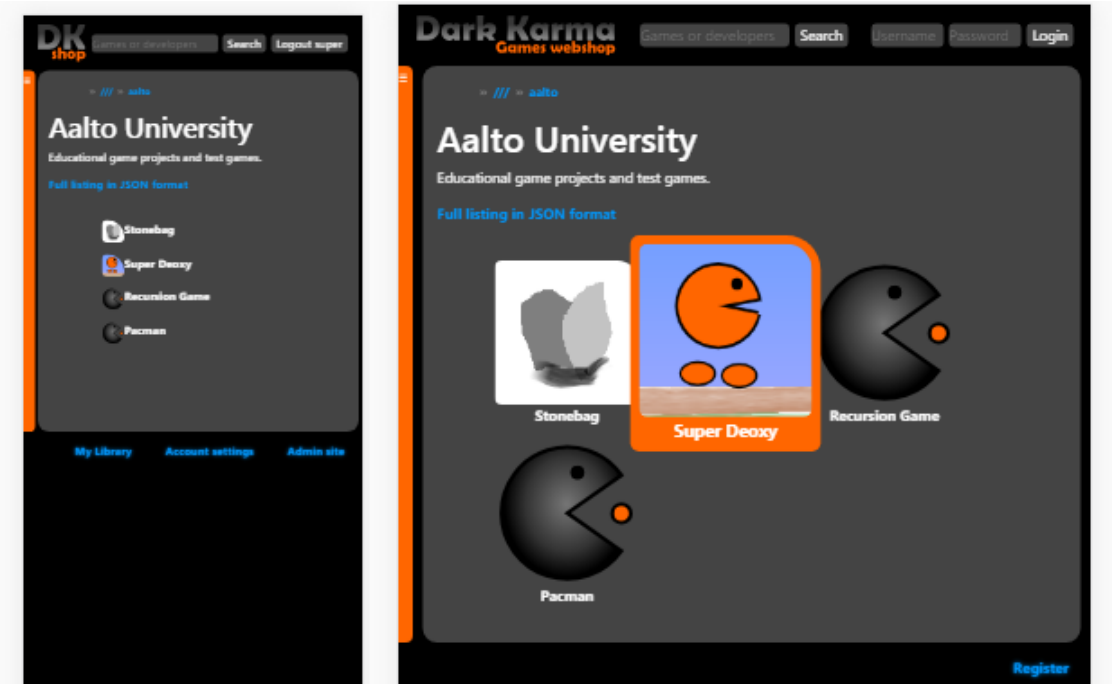
The library is designed to take full advantage of HTML5 syntax while taking distance to DOM. In particular, all optional tags are omitted, allowing code to be written without accurately specifying the nesting of elements, thus allowing for code like `doc.li.a("Link text", href="url")` , that disregards whether the *a* element goes within the li or not (in DOM it goes inside, and a HTML5 parser knows what to do with it). Further, no "pretty printing" is used because extra whitespace would appear as TextNodes on DOM, interfering with data. User browser developer tools to view pretty-printed source code.

See Documentation/HTML5Tagger.pdf for more information.

URL layout

The major URL layout was setup to put content first, so the structure is defined by games, developers and genres. Only very minor changes and a few extensions have been done to the original plan. There is no separate root for API but instead .json URLs are offered within the main hierarchy. A trailing slash was used by default, with the meaning that the URL refers to contents within. The exceptions are "+" for adding a new game, and the JSON files, where the target is directly referred to.

Responsive design and accessibility



As seen in the screenshots above (scaled at 50%), the site logo was implemented in *art direction* with two sizes for devices of different sizes. Also, the game listings change from large icons to list views for small devices.

Nearly all scaling is based on em or rem units, so that they scale with font size. For views narrower than 480px the root font size is decreased so that the page width remains at 30rem. For views wider than 1920px the font size is increased keeping the page width at 120rem. For all window sizes between these limits, the default font size of 16px is used. Flex layouts take care of effective use of screen space.

Meta viewport is used to preserve readable font size and intended layout on phones that would otherwise report dimensions much larger than their screen sizes. I could not test the effect of this on HiDPI displays but I suspect that the tag has no effect on them (i.e. one CSS pixel still corresponds to 2x2 or more hardware pixels).

Further, semantic HTML elements and suitable DOM layout (ordering) is used to make the site more accessible to screen readers and other accessibility devices. Although the site makes extensive use of mouse hovering, care is taken to make sure that everything works smoothly without hover, or with keyboard-only navigation (tab traversal).

Payment system

First I tried inserting a HMAC-protected purchase request into the pid field sent to the payment service, with the idea that once a payment is approved, we'd get back the message containing information on what was purchased, securily armored against modification by either the user or by the payment service, without having to store anything about the deal on our server.

Alas, the payment service enforces a 64-character limit on pid length, so this had the be scrapped, and instead the message is now stored in Payments table until a response comes from the payment service.

This approach is resistant against message replay as well as modification, and multiple on-going purchases can be handled because each has UUID pid. The user session is not used, so payment approvals can be processed even if the user has logged out during the payment. The game ownership is added to the user specified in the Payment indicated by pid, rather than a logged-in user.

Two different modes are implemented. If PAYMENTS_GET=True, the user is redirected to payments URL right after clicking the purchase button. Otherwise a separate page for choosing payment provider is shown, with buttons that cause user's browser to POST to the payment service. Apparently only the latter mode works with the Simple Payments version used at payments.webcourse.niksula.fi.

RESTful API

A minimal API is implemented:

- /games.json - a list of all games, hiscores etc.
- /SomeDeveloper/sales.json - sales of all developer's games
- /SomeDeveloper/AnyGame/sales.json - sales of a particular game

Games list is implemented as a single large file that could easily be cached. It doesn't depend on user session, and thus can be safely permitted cross-origin access with no authentication.

The sales API requires authenticated user with access to that developer. A session-based login is used by default, and this is used in practice in the sales statistics at /SomeDeveloper/s/, which fetches sales.json for the sales chart.

Alternatively, cross-origin access is permitted with authentication via GET parameters (yes, this is a bit risky), e.g. http://localhost/WSD/sales.json?username=aaltodev&password=yourpassword (http://localhost/WSD/sales.json?username=aaltodev&password=yourpassword) (404 is used for access denied... don't ask)

The **Access-Control-Allow-Origin** header is set to allow this alternative access from cross-origin scripts. Session-based authentication is limited to same origin (assuming that browsers enforce such security, as is done with the rest of the site as well).

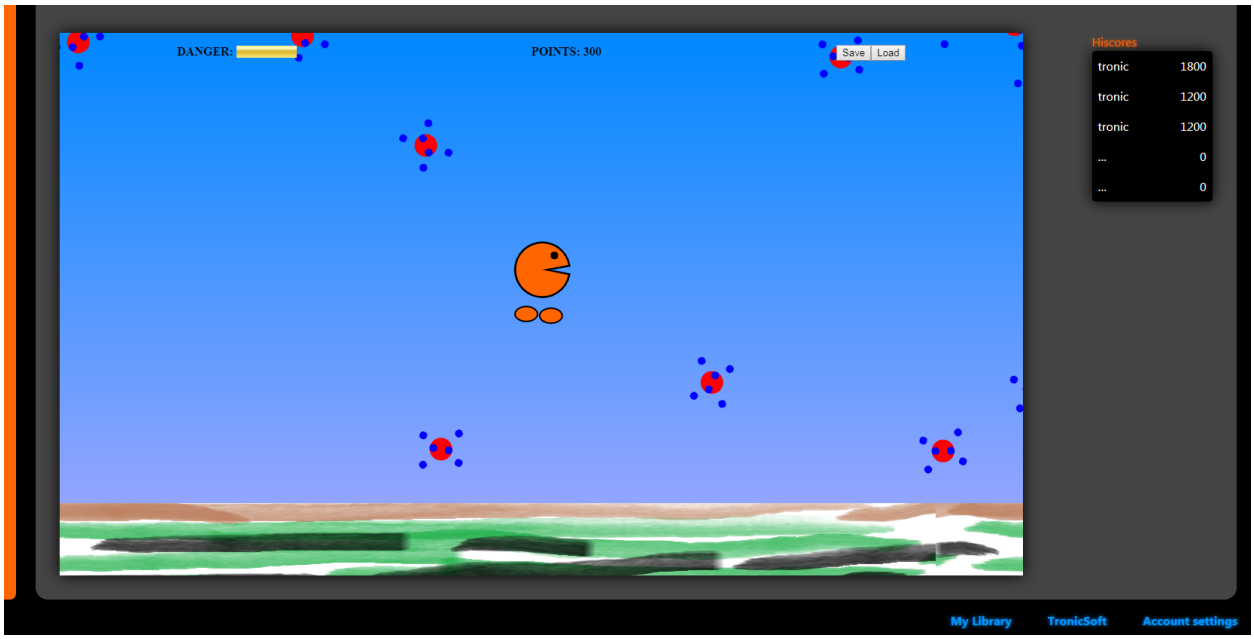
API requests are served with **Content-Disposition** headers, allowing for easy download of data. Sales attachment filenames contain developer and game names, as that is more useful than simply "sales.json".

Game and site JSON APIs

The game-site messaging based on **postMessage** implements all messages in the given specification and nothing more. The site POSTs to the game page using very similar messages as necessary. This API is somewhat different from the rest of the REST API mostly because it was implemented earlier.

Of particular interest is the response given by server to a hiscore submission; it may request the site to refresh a certain element (#hiscores), so that new hiscores and the popup message(s) announcing the score may be displayed without interrupting the game. Since I decided to work without JQuery, this is implemented fully in **webshop/static/webshop/script.js**.

Super Deoxy



I've implemented my own game in Javascript, familiarizing myself with the canvas element. First I planned to use more drawn graphics but being limited to MS Paint and Inkscape led to using character animation created using shapes generated from Javascript. Some math had to be derived for elastic collisions, correct feet animation and to evenly distribute the electrons around their atoms.

To constrain the manhours spent, I had to limit down on the features of the game, and it implements little beyond that of the requirements. However, I've done my best to implement well that what is present, e.g. smooth animation (matching your display refresh). As a part of the said limitations the game was only implemented for a fixed resolution of 1280x720, and it becomes unplayable on very small screens (unless browser scaling can be used). HUD uses default styling which is quite ugly.

Testing and coverage

Full coverage testing was initially planned but I had to settle with 80 % attained via a few test cases. The primary approach is to start at site root and breath-first traverse all links found, which actually covers most of the site and the API. A few additional tests were added for notable special cases, but payments and forms remain untested. Five cases of broken links or misbehaving code were uncovered by these tests over the last few weeks.

Coverage report **Documentation/htmlcov/index.html** was produced with:

```
python -m coverage run manage.py test
python -m coverage html
```

Manual testing of all forms using correct as well as various forms of malformed input guided the implementation of form processing and error handling.

Notably *nothing* related to escaping, injection and such had to be considered. The Django ORM and forms entirely avoid the need for manual escaping, and quite surprisingly HTML5Tagger also automatically played along despite never being tested with or designed for use with Django.

The HTML, CSS and JS were checked with the Nu Validator, jshint and jslint. It needs to be noted that some other validators may give errors for code that is in fact valid HTML5.

Features and grading

- Minimum functional requirements (DONE)
- Authentication (100/200) - no email validation

- Basic player functionalities (300/300) - all listed features and more
- Basic developer functionalities (200/200) - all listed & more
- Game/service interaction (200/200) - implements the described API entirely
- Quality of Work (100/100) - all features, pro quality
- Non-functional requirements (200/200) - clear plan & documentation
- Save/load and resolution feature (100/100) - implemented and used in my own game
- 3rd party login (0/100) - not implemented
- RESTful API (100/100) - JSON API implemented with cross-origin/auth support
- Own game (100/100) - Super Deoxy should suffice for full points
- Mobile Friendly (50/50) - I accidentally a bit overkilled this too
- Social media sharing (0/50) - not implemented

This leads to a total of 1450 points. I have not compensated for having to implement everything by myself, nor penalized the "group work" part for lack of collaboration.