Project Outline

civ is a nation simulator which allows users to create and rule over a civilisation, handling the day-to-day economic tasks of a nation and completing to be the largest nation in the land.

Design Choices

Language/Framework - Python w/ Django

My choice of language came down to PHP with the CodeIgniter framework or Python with Django framework. I chose django due to it's built-in webserver which would make it painless to fulfil the requirement of the website being able to be hosted on DICE. I also chose it due to it allowing me the opportunity to strengthen my python skills in a development environment. PHP also is now a rather dated language, so I think using this as an opportunity to adopt a more modern development environment would be a good choice.

Database - sqlite3

I first started on mySQL, as I already had that installed and ready to go in my development environment. I later migrated to postgreSQL in order for easy movement to DICE. However, during my project I encountered issues moving between computers and purely for ease of development I switched to the single file sqlite3. Due to how django handles databases, this can be swapped for postgreSQL, mySQL or any other production-level database system for is this application was actually used in production.

Web Server - Django's built-in

I chose to use Django's built in Web Server due to the fact it would be extremely easy to use on DICE. In a production environment, it of course would not be scalable and thus would need to be swapped with something like nginx with uWSGI, gunicorn or another production-level web server but on DICE is makes things extremely easy.

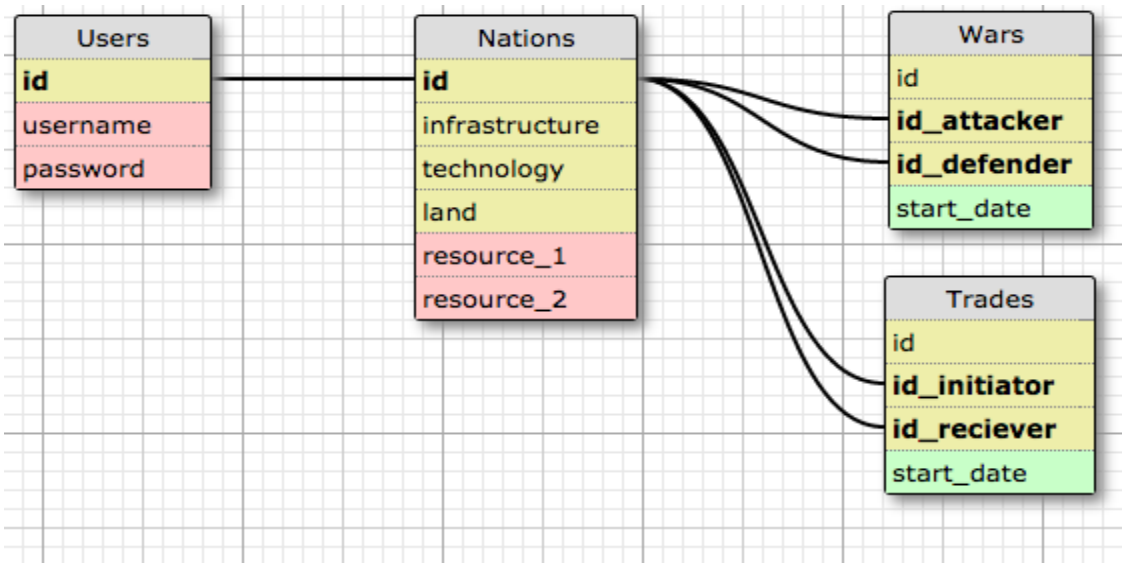
HTML/CSS/JS framework - Twitter Bootstrap

I chose Bootstrap due to it's very comprehensive implementation of a unified visual style while requiring minimal effort from me other than css classes I would normally use anyway. However, bootstrap is extremely popular, so I would run the risk my website looking very similar to others and thus to visually stand out I have used a custom theme that keeps a unified visual style while having less rounded corners in the visual style as well as a slightly different colour scheme. I think these changes combine to give the application and more distinctive look and feel.

Design

Database

The original design of the database was to have four tables: a table of users, a table of nations, a table of wars, and a table of trades. I believed these to be the four crucial areas that would require their own tables. My reasons for separating the tables was to allow for the efficient querying of information. For example, I would have no use for war data if I was merely looking to display the current statistics of the nationView view and I would have no need of general nation information when I merely wanted to check which wars are currently being fought. I separated users and nations to allow for a persistent identity across multiple nations if in the future multi-nation accounts or such features were implemented.



Yellow: Integer, Red: String, Green: Date, Bolded: Keys

By the end of the project, through difficulties outlined in *deficiencies* below, the application was redesigned to implement user accounts and the core nation itself as a priority and left the ability for wars, trading and further complexities to be added later. The segmented nature of database definitely helped during this process.

Page layout

Nations would be displayed on one page, editable on a second, extendable on a third, and with bills and taxes making up another two pages. There would then be a final page to display rankings. Separating the application into multiple pages should allow for the user to not become overwhelmed with information in what is a very heavily data-displaying application as well as simplify each page programmatically allowing me to set clear goals for myself in finishing one page before moving onto another.

Testing

My testing regime during the implementation phase consisted of writing a section of code with an expected result, before running the application and attempting to use the newly developed section in the hope it would display the expected result. This was all done visually and I would fixed any errors that I spotted or the application reported then repeat the process until the expected result was achieved and then informally on a few extreme values.

This of course is a far from complete testing method so I combined this with a more formal testing after the complete application was implemented in order to give credence to the reliability of the application.

My plan was to first check the effect of user accounts on each view. Each url should be tested with an anonymous user, the correct logged in user, and a logged in user specifically not having permissions to perform an action (if applicable). My reasoning was that the first piece of code on the execution of a GET or POST function in many of the views was to check for authentication.

Then, I would check that any individual helper functions outwidth the GET or POST functions in a view would give satisfactory results as they are used heavily throughout the logics of the views and having no errors there would give a good chance of a view acting correctly.

Finally, I would check the database itself to make sure that each view has done it's job. This is due to the difficulty in checking the contents of a view with dynamic values. However, due to the non-asynchronous nature of the application the effects of a view can be tested by checking the database after key actions. The graphical nature of the view will have been checked through the implementation process during my manual use of each view checking for expected behaviour which while informal should prevent gamebreaking visual bugs.

Deficiencies

Design

One constant theme throughout my implementation of this project is that I had drastically underestimated simply how long it would take for me to learn django. I assumed that a knowledge of web design as well as previous experience with other web frameworks would result in a short learning time being required. This was a mistake on my part and something I can take forward to future projects. In my proposal I outlined around five hours for learning specifics of django but this was wildly inaccurate and I faced a learning process at nearly every turn. Outside of the very basics, django has it's own way of doing many things compared to my previous experience of PHP (namly abstraction) which while not intrinsically difficult does require time to learn the django specific implementation of many parts of the program.

These difficulties have led to my actual implementation time being quite drastically cut and I made the design decision to cut trading and wars from the implementation. I realise this cuts a decent chunk of functionality from the game in its current state but the base functionality had to be prioritised in order for me to feel the game was functional. With extra time, I feel that trades and wars could be implemented after some learning of django's linking between models.

Overall, I was overzealous in my design of the project and overestimated my ability to produce such a full-featured game in such a short span of time. I should have aimed for a smaller project.

However, I do feel that the parts of the proposal that remained were working and implemented to a reasonable standard with a respectable level of polish. The core aspects of the application: the idea of a user, an associated nation, the tasks of maintaining and building a nation, and competing in rankings of various nation attributes with other nations has been delivered.

Implementation

I believe the error reporting - by using a GET method to pass the fact there is not enough funds to complete purchases in the expandNationView class (views.py line 119) - has far cleaner ways to be implemented and if I had more time I would attempt to refactor it using dictionaries similar to how I outline in the Idiomatic code section below.

My knowledge of django in some cases has limited my implementation, and thus instead of I have had to fall back on more general ways to make things work. For example, in situations where I use a hard-coded form (templates/nations/nation_extend.html line 35 onwards) rather than a django generated one, I do input validation purely on the frontend via HTML5 attributes on the form itself. Obviously best practice would have these checked for true database validity rather just being an integer, but it should hopefully be a good enough solution and cases where the database could be passed errors are hopefully nonexistent.

Things I'm proud of

Idiomatic code

I made use of python's dictionaries in passing the results of the validation stage of both paying bills and collecting taxes to their respective templates. (nations/views.py lines 254-264 and 310-312 respectively). As well as for passing the category rank 1 nations to it's template (nations/views.py lines 245-252).

My reasoning behind using this feature of python was that it allowed the passing of a related error along with the validation result allowing for responsive error reporting to the user in a manner unified with the design of bootstrap - the use of alerts. While I haven't had the time

to implement this everywhere, I believe it efficiently enhances the areas they are implemented and with more time would be the standard way to pass errors in the application.

Unified Design

Using Twitter's Bootstrap project, I was able to present a clear and unified user interface with less effort required compared to creating the interface from scratch. There is use of bootstrap stylised tables in nearly every view giving a clear display of data consistent across nation tasks - something I believe very important in such a spreadsheet-esque project.

Good use of HTML5

I use HTML5 forms throughout the application which allow form validation and multi-platform usability. Declaring types on forms in HTML5 allows for inputs such as integers, dates or emails to be typechecked before being passed to the backend. Mobile devices can use this information to change the keyboard to make it easier for input and it also allows the application to give errors on the frontend rendered by the browser itself - creating better continuity by allowing for the errors to be presented in the design of the host OS itself.

Decision Making

I believe the decision to change the design of the application quite drastically was the correct one to make and was critical in me producing an application to report on. I feel that this could have easily resulted in a black hole of time and that taking the hit of delivered functionality was a good decision that resulted in a delivered product.

Password Security

I override the default password field of the UserForm in the forms.py to use a password widget from django. This encrypts the password with an encryption algorithm plus a salt so that user passwords in the database is safe in the event of intrusion.