getPage function in scg.tsx. this is the core of the panelling system. It returns the current panel, as set in state, and passes it any props it needs. The props are mostly customisations from the ruleset. Where panels are not passed props for their ruleset customisations, these are passed as data or functions via context. 

fetchClasses function in scg.tsx shows format of fetch calls to API. It requests the data from the API. Then on receipt of data, converts the data from the API/JSON format to the format used internally. In this case that involves also fetching class-descriptions since these are linked to the classes. Maps are largely used internally since they are faster than javascript objects, so a helper function converts the json objects to a map.



The object to map helper function:



Helper functions to find objects in a map by id, but if it does not exist, it returns a default object of the type GameObject which all objects inherit from to avoid causing further errors when these objects are read.



Snapshot of 2 types showing inheritance from GameObject:



PanelFrame allows for quick addition of new panels as needed, while keeping same basic layout intact:



Snapshot of equipment panel using the PanelFrame component as a wrapper for its contents. The panel is passed data about the players character, and functions to modify it via the context consumer. It consumes the data given by the context provider in scg.tsx



What is stored in state in the HOC (scg.tsx):



  
Of this, the character and operations are passed into characterContext, while the contents of the superclass, GameObjectsContext is passed to gameObjectsContext so components can access the store of objects fetched from the REST API.

High level view of Character interface. This is passed into characterContext, and used as the savefile format



Operations for modifying the internally stored character are also passed as data via the characterContext. Their format is as follows (many interfaces omitted to save space, but this gives the idea. Even includes so-called meta operations that handle exporting the character, displaying a modal etc.):



Populated by the HOC in its constructor, saved in state alongside data, and passed to components with context:



In the HOC, getPage() call is wrapped in context providers so they have access to all the data (and functions since they are stored as data):



REST API consists of a sequelize model for each object, a controller defining different ways of fetching the objects (including expanding associated objects), and some routes mapping endpoints to the controllers’ functions.

skill.model.js:

  
skill.routes.js:

  
Routes included in server simply by:



skill.controller.js:



Sample of code used to ensure point versatility (taken from fociOperations.removeFocus in context):



And code for always spending the least versatile point first (taken from fociOperations.addFocus in context):



