jamk

Improving the user experience of a JSON Editor -based user management tool

Lassi Lehtinen

Bachelor’s thesis
April 2023
Information and Communication Technology
Degree Programme in Software Engineering

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Lehtinen, Lassi

Improving the user experience of a JSON Editor -based user management tool

Jyväskylä: JAMK University of Applied Sciences, April 2023, 60 pages.

Information and Communication Technology. Degree Programme in Software Engineering. Bachelor's thesis.

Permission for open access publication: Yes

Language of publication: English

Abstract

Zaibatsu Interactive Oy, a video game and software company from Jyväskylä had issues with the user experience of an in-house user management tool for one of the games they develop. The tool was used by the support team of the game for handling user data and performing administrator tasks. The goal of the project was to improve the user experience of the tool.

The tool utilized JSON Editor, a library that automatically builds web page user interfaces for JSON data. Along with improving the user experience of the in-house tool, the suitability of JSON Editor as a basis for the user interface of such a tool was also evaluated.

The project was started by studying the tool and its user experience. The main body of work then consisted of fixing any issues that were detected and adding new functionality that was determined to be beneficial for the user experience of the tool.

By the end of the development project, the usability and user experience of the tool had both improved noticeably. The usefulness of JSON Editor as a basis for a user interface was evaluated and it was concluded that the library is suitable for the use even though it also has some limitations.

Keywords/tags (subjects)

user experience, user interface, user management tool, development project, JSON Editor, JSON Schema

Miscellaneous (Confidential information)

-

jamk

Kuvailulehti

Lehtinen, Lassi

JSON Editor -pohjaisen käyttäjähallintatyökalun käyttäjäkokemuksen parantaminen

Jyväskylä: Jyväskylän ammattikorkeakoulu. Huhtikuu 2023, 60 sivua.

Tieto- ja viestintätekniikka. Ohjelmistotekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: englanti

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Kehitystutkimuksen asiakasyritys oli Zaibatsu Interactive Oy, jyväskyläläinen videopeli- ja ohjelmistoyritys. Yrityksen kehittämään peliin liittyi sisäinen käyttäjähallintatyökalu, jota pelin tukihenkilöstö käytti pelaajien tietojen hallinnoimiseen ja ylläpitotoimiin. Työkalun käytettävyydessä ja käyttäjäkokemuksessa oli ongelmia, joiden ratkaiseminen oli projektin tavoite.

Työkalun käyttöliittymä oli toteutettu käyttäen JSON Editor -kirjastoa, joka rakentaa annetulle JSON-datalle automaattisesti editorin web-sivuna. Työkalun käyttäjäkokemuksen parantamisen ohella arvioitiin JSON Editorin soveltuvuutta käyttäjähallintatyökalun käyttöliittymän perustana.

Projekti aloitettiin tarkastelemalla työkalua ja sen käyttäjäkokemusta. Työvaiheessa korjattiin käyttäjäkokemuksessa havaitut ongelmat ja lisättiin työkaluun sitä parantavia uusia ominaisuuksia.

Kehitystutkimuksen tuloksena työkalun käyttäjäkokemus ja käytettävyys paranivat huomattavasti. JSON Editorin hyödyllisyyttä käyttöliittymän perustana arvioitiin ja johtopäätöksenä todettiin, että kirjasto soveltuu mainittuun käyttöön tietyin rajoituksin.

Avainsanat (asiasanat)

käyttäjäkokemus, käyttöliittymä, käyttäjähallintatyökalu, kehitystutkimus, JSON Editor, JSON Schema

Muut tiedot (salassa pidettävät liitteet)

-

Contents

1 Introduction 5
1.1 Client 5
1.2 The task 5
1.2.1 Description and motivation 5
1.2.2 Clarifications concerning the vocabulary used in the report 5
1.2.3 Security concerns 6
2 Research approach 6
2.1 Development project 6
2.2 Research questions 7
2.2.1 Nature of research questions in a development project 7
2.2.2 The research questions 7
3 Overview of Player editor 8
3.1 Description and main usage 8
3.2 The basic operation of the tool 8
3.3 Technologies used in Player editor 8
3.3.1 JSON Editor 8
3.3.2 JSON Schema 9
3.3.3 JSONdiffpatch 9
3.3.4 Bootstrap 10
4 Problems in Player editor 10
4.1 Assessing the user experience of the editor at the start of the project 10
4.2 Issues in user experience 11
4.2.1 Some actions required “hidden knowledge” 11
4.2.2 Confusing workflow for enabling and disabling a feature for a player 11
4.2.3 Logically connected pieces of data were separated from each other in the UI 12
4.2.4 Suboptimal sizing of UI elements 13
4.2.5 Poor responsivity of the main editor view 13
4.2.6 Unintuitive UI control element usage 13
4.2.7 Missing tooltips and explanations 14
4.2.8 Breaking common UI layout conventions 14
4.2.9 Lack of a clear indication of current operation environment 14
4.3 Problems with game terminology 14
4.4 Errors in the behavior of Player editor 15

5 Improving Player editor ... 16
5.1 Selecting the approach for improving the editor ... 16
5.2 Improving the user interface and the user experience ... 17
5.2.1 Categorizing the editor contents and moving to tabbed layout ... 17
5.2.2 Removed “hidden knowledge” requirement for gifting upgrades ... 20
5.2.3 Improved workflow for managing the status of a limited time benefit feature ... 22
5.2.4 Fixed terminology to match the naming used in the game ... 22
5.2.5 More intuitive UI control element usage ... 23
5.2.6 Visual and layout improvements ... 24
5.2.7 Adjusted the responsive behavior of the editor ... 25
5.2.8 Adjusted input field widths ... 27
5.3 Fixing errors in the editor functionality ... 28
6 Implementing a major new feature: Change logging ... 29
6.1 Overview of the change logging feature ... 29
6.1.1 High-level description ... 29
6.1.2 Background and rationale for adding change logging to Player editor ... 29
6.1.3 Technical overview of the designed solution ... 30
6.1.4 Challenges in implementing the feature ... 31
6.2 Detecting and handling changes to different types of data ... 32
6.2.1 The problem with handling changes to different types of data ... 32
6.2.2 The concept of a change object ... 32
6.2.3 The concept of a change batch ... 32
6.2.4 The concept of a diffing configuration ... 33
6.2.5 The relation between a change batch and a diffing configuration ... 34
6.2.6 Details of the components of a diffing configuration ... 35
6.3 Filtering array modification false detections ... 37
6.4 Change log user interface ... 38
6.4.1 Overview ... 38
6.4.2 Technical implementation ... 38
7 Results ... 39
7.1 Difficulties identified in using JSON Editor as a basis for a user management tool ... 39
7.2 Fixing JSON Editor’s shortcomings ... 40
7.3 New feature showcase ... 41
7.3.1 Change log ... 41
7.3.2 Unsaved changes view ... 42
7.3.3 Automatic change detection ... 42

7.4 Comparisons between old and improved version 43
7.4.1 Player search view 43
7.4.2 Editor main view 44
7.4.3 Inventory-like view 47
7.4.4 Upgrades view 48
7.4.5 Enhanced workflow for enabling and disabling a feature for a player 49
7.4.6 Grouping related features 50
7.5 Feasibility of JSON Editor as a foundation for a user data management tool 51
7.5.1 Overview 51
7.5.2 Useful qualities of JSON Editor as a basis for a user interface 52
7.5.3 Shortcomings of JSON Editor-generated user interfaces 52
7.5.4 Summary 53
8 Discussion 53
8.1 Remaining issues 53
8.1.1 Player search feature limitations 53
8.1.2 Some in-game items are still only represented as “magic numbers” in the editor54
8.1.3 Change log history has an arbitrary length limit 54
8.2 Ideas for further development 55
8.2.1 Overview 55
8.2.2 Better workflow for gifting in-game items 55
8.2.3 IAP bundle gifting 55
8.2.4 Add more in-editor explanations and tooltips for help 55
8.2.5 Further simplification of tabular data views 56
8.2.6 Refactoring the change log system implementation 56
8.3 Conclusion 56
References 59

Figures

Figure 1. Simple example JSON schema and an example extension schema with two main categories and one subcategory 18
Figure 2. Example data before and after restructuring according to the extension schema shown previously in Figure 1 19
Figure 3. Each category of properties has its own tab (the row at the top). 20
Figure 4. The upgrades array schema with the "Give upgrade" button for each upgrade 21
Figure 5. The callback function for the "give upgrade" button 22
Figure 6. Manual configuration of the width of an editor component 27

Figure 7. A simplified example of the main object that holds all change objects 33

Figure 8. Visualization of the relationships between the player schema, change batches and diffing configurations 35

Figure 9. Visualization of an issue in the process of diffing arrays using the jsondiffpatch diff function without unique identifiers for array items 36

Figure 10. A view of the change log with two changes and a comment 41

Figure 11. The unsaved changes view with two changes and a comment..... 42

Figure 12. Automatic change detection comment prompt as it appears on Firefox 43

Figure 13. Player search view, original implementation 43

Figure 14. Player search view, improved 44

Figure 15. The old editor main view 45

Figure 16. The new editor main view 46

Figure 17. Comparison of the lengths of the main editor views. The red rectangles divide the editor views to screenfuls for illustration purposes. 47

Figure 18. Inventory-like view, previous implementation..... 48

Figure 19. Inventory-like view, improved version 48

Figure 20. Old upgrades view..... 49

Figure 21. Upgrades view, improved 49

Figure 22. Limited time benefits toggle view, old 50

Figure 23. Limited time benefits toggle view, new..... 50

Figure 24. Illustration of poor placements of related features 51

Figure 25. Related features have been grouped together under a common heading..... 51

Tables

Table 1. Fictive example of conflicts between terminology used in the game and legacy naming used in Player editor. 15

Table 2. Fictive example of the importance of the name callback for change log entries concerning the inventory-like view. 36

1 Introduction

1.1 Client

The client of this development project thesis was Zaibatsu Interactive Oy, a video game and software company from Jyväskylä, founded in the spring of 2014. In addition to video game development, Zaibatsu does subcontract projects in the field of video games and software and hires labor force to other companies. The chief executive officer of the company is Jussi Ultima.

1.2 The task

1.2.1 Description and motivation

The objective of the practical side of this bachelor's thesis was to improve the user management tool for a mobile game that Zaibatsu develops. The in-house tool, also referred to as the *Player editor*, is a utility used by the support team of the game for managing the player account data and for performing administrative actions.

The software had multiple issues in its user experience (UX), hindering the work of the support team. That is why enhancing the UX was the highest priority objective for the development project with the visual appearance of the tool also being improved upon.

1.2.2 Clarifications concerning the vocabulary used in the report

It is important to make a clear distinction between discussion concerning the users of the tool and the users of the game. For the sake of clarity, the word *user* is reserved for referring to a member of the game support team as a user of the tool and the word *player* for referring to a person playing the game. The only exception to this assignment of terms is when the concept or the details of the *user management tool* are being discussed – the word *user* in the name of the tool refers to the players of the game.

Furthermore, two different names are used to refer to the software that was being improved. The term *user management tool* is used to refer to the application from the perspective of its purpose

and the name *Player editor* refers to the tool as a product. These two terms are used interchangeably.

1.2.3 Security concerns

Based on a request made by the client, some information about the development project was not included in this report. Among the things left out were the name of the game associated with the user management tool as well as mentions of any names, terms, and features of the tool that were tightly associated with the game. To still be able to discuss the details of the tool as well as the actions taken for improving it, all such terms associated with the game were replaced with terms with separate, unrelated meanings that were deemed suitable to illustrate the aspect of the tool that was being discussed.

Additionally, it is worth noting that all data displayed in the illustrative screenshots is randomly generated and does not reflect actual game values. All game content related terms used are also fictitious. Text elements that are irrelevant to what is being illustrated are censored to be just randomly sized solid blocks the color of the text.

2 Research approach

2.1 Development project

The research approach chosen for this thesis is called a *development project*. On a general level, the objective of a development project is to improve or design the working conventions of the client, which is often a company. What follows is that one of the distinguishing features of a development project is that it is done in collaboration with the client. A development project is presented in a thesis as a documentation of the stages of work in the project and an overview of the results. (Latva-Reinikka 2022.)

2.2 Research questions

2.2.1 Nature of research questions in a development project

Because the development target of this development project type thesis was a piece of software, the research questions concerned mainly practical issues. Among them were things such as problems in the software, decisions regarding the implementation and design of the software and the choices of software libraries and frameworks.

2.2.2 The research questions

What are the main contributing factors for the perceived difficulty of and discontent in the use of a JSON Editor based user management tool?

The first question served as a starting point for the development project. Analyzing the editor and identifying the main problems in its implementation and user experience provided a useful understanding of the task.

What can be done to remedy the problems identified in using JSON Editor as a foundation for a user management tool?

After having identified the most notable issues in the editor, solutions for the recognized problems were sought and implemented. The main body of work in the development project was centered on this question.

Is JSON Editor a suitable foundation for a user data management tool?

The final question remaining after finishing the development project was the evaluation of the usefulness of JSON Editor as a groundwork component for building a user management tool like Player editor. The experience and knowledge gained from working with JSON Editor during the project allowed forming an informed opinion regarding the question.

3 Overview of Player editor

3.1 Description and main usage

Player editor is an internal tool that is used for managing the data associated with individual player accounts in the game. In its essence, the tool is an interface for interacting with the user database of the game. It is implemented as a web app that can be accessed by a regular internet browser in the local area network of the company. The types of actions performed using the tool include investigating any problems a player might encounter with in-app purchases as well as warning and sanctioning players that violate the rules of the game.

3.2 The basic operation of the tool

A session in using Player editor begins with searching for the player account to manage. After the user has found the specific player account, the basic operation of the tool on a high level can be described as follows:

1. The editor sends a request to the backend of the game for the player data.
2. Backend performs a database query for the given player data.
3. Backend sends a response to the editor with the player data in JSON format as the payload.
4. JSON Editor builds the HTML form based on the player schema and populates it with the fetched player data.
5. The user inspects and / or modifies the player data.
6. If modifications were made, the user should press “Save”.
7. If the user chose to save their changes, the editor sends the modified data to the backend.
8. If modified data was sent to the backend, the backend performs necessary operations for the data and does a database query to update the player data accordingly.

Essentially, the editor is an HTML form generated by a tool called JSON Editor based on a JSON schema representing the player data structure.

3.3 Technologies used in Player editor

3.3.1 JSON Editor

JSON Editor is an open-source tool designed to facilitate inspecting and modifying JSON data. Given a JSON object and an associated JSON schema describing the data in the object, JSON Editor

automatically generates an HTML form displaying the data in editable format. (json-editor: JSON Schema Based Editor 2022.)

The tool was originally developed by Jeremy Dorn, starting in 2013, but the development was officially transferred over to a community-maintained fork of the repository in 2018 (Official-ify this fork 2018).

JSON Editor version 2.6.1 provides the basic user interface for Player editor. The user management tool gets the user data from the database via the backend as a JSON object. A custom JSON schema representing a player data object has been configured to provide JSON Editor with the required meta data of the user data object, namely the names of the properties that are to be displayed in the editor as well as their data type and some JSON Editor specific configuration options for controlling the way the properties are displayed. JSON Editor then uses this data and the given player data to build and populate the HTML form for inspecting and modifying the player data. After modifying the data, the user must save the changes they have made to push the updated data to the database via the backend of the game.

3.3.2 JSON Schema

JSON Schema is a system for describing the type and structure of data stored in a JSON object. It also allows defining rules for validating the data. (JSON Schema 2022.)

JSON Editor uses JSON Schema for both, building the web form for editing the data as well as a rule set for validating the data. JSON Editor adds some new properties to the schema to allow controlling the behavior and appearance of the editor on a per-property basis.

3.3.3 JSONdiffpatch

JSONdiffpatch is a JavaScript library for comparing and patching data stored in JavaScript objects. Given two objects, it generates a new JSON object containing information of the differences between the two objects. It can also generate patch objects which can be used to revert or apply changes to data. (jsondiffpatch: Diff & patch JavaScript objects 2022.)

Player editor only uses the diffing functionality of JSONdiffpatch. The usage of JSONdiffpatch in Player editor is to determine exactly what changed in the data when JSON Editor notifies the system of a change in the data.

3.3.4 Bootstrap

Bootstrap is a CSS framework designed to speed up and simplify the development of visually pleasing and responsive web pages and applications. It offers tools for creating responsive content layouts, customizing element appearances, and coloring as well as an extensive list of custom-made components to further aid in building interactive web pages quickly. (bootstrap: The most popular HTML, CSS, and JavaScript framework for developing responsive, mobile first projects on the web 2022.)

JSON Editor comes with a few built-in themes for the look of the editor and one of them is built using Bootstrap 4. The Bootstrap 4 theme is the one that is used in Player editor.

4 Problems in Player editor

4.1 Assessing the user experience of the editor at the start of the project

The first step in improving the usability of the editor was to assess the then current user experience of the tool. Analyzing the UX started with gathering information about the usage of the tool and the problems experienced by its users. To achieve this, a request for free-form ideas, wishes and thoughts about Player editor was made to the support team. To give some direction for answering, an optional, short questionnaire was added to the request:

- What is/are the action(s) you most commonly use the tool for?
- What is/are the most common and/or frustrating problems you face while using the tool?
- How does it feel to use the tool? (Comfortable, easy, fun, scary, confusing, risky, tedious, …?)
    - Can you identify the source of these feelings?

A list of welcome additional features and improvements was also compiled based on a meeting with the developers who were in response of updating and maintaining the tool. Additionally, the tool was also manually tested and studied to find any usability problems or technical errors in it. All found issues were taken note of.

Unfortunately, the response rate to the free-form questionnaire was very low. This meant that the work was started based mostly on the discussions with the maintainers of Player editor as well as the findings from testing the tool manually. The rest of chapter 4 details the issues that were found in Player editor.

4.2 Issues in user experience

4.2.1 Some actions required "hidden knowledge"

Some actions in Player editor required detailed knowledge of the implementation of the game. For instance, the game contained upgrades that were unlocked when the player had collected a specific number of collectables linked to that unlockable. The problem was that if a member of the support team were to manually gift such an upgrade to a player, the way to do so would require them to set the player's number of the associated collectables collected to that specific value. The main issue with this approach was that it meant that to give certain upgrades to players, the members of the support team had to have knowledge of arbitrary numeric values that were specific to those features in the game and varied between different upgrades.

Additionally, some other items were only referred to by a numerical ID, once again requiring the operator of the Player editor to know which number corresponds to the item they needed to give or remove. A list of items of this type belonging to a player was just an array of these numeric IDs.

These problems were also not limited to knowing these "hidden" pieces of information – it also included remembering them. As Johnson (2014) reminds, relying on users having to remember arbitrary pieces of information is in contradiction with one of the primary uses of computers: the devices retain information to allow people to concentrate on tasks other than memorization. Such an elementary function of computers should be utilized in the design of user operated systems, such as Player editor. (Johnson 2014, 102-103.)

4.2.2 Confusing workflow for enabling and disabling a feature for a player

The game contained a feature that, once activated, gave the player extra benefits for a limited amount of time. After the given time had passed the feature would reset and the player would

need to activate it again to continue enjoying the benefits. Player editor had a setting that allowed manually controlling the state of the feature for a player account.

The workflow for enabling or disabling the feature was highly unintuitive. The editor only showed the expiration time for the active status of the feature. Gifting or removing the activation of the feature for a player was done by modifying the expiration time. If the time was set to any point in time in the past, the backend of the game interpreted that as a signal to deactivate the feature for the player. Correspondingly, setting the time to future meant that the feature should be activated for the player in question.

If the backend received a value representing a point of time in the future, it adjusted the expiration time according to the intended duration, meaning that the user of the tool did not need to worry about setting the *correct* expiration time – any future time was valid for activating the feature. The same was true for deactivating the feature, any time value in the past was adjusted by the backend to the value of the Unix epoch, January 1st, 1970 at 00:00:00 UTC.

The described workflow was explained in the label for the control as an additional, parenthesized explanation after the property name. Even with the instructions, such a workflow was unusual and cumbersome, especially considering that the control basically acted as a checkbox toggle, only a particularly difficult-to-use one.

Another problem with this implementation was that to determine whether the feature was currently active or inactive for the player, the user had to check the time value currently shown to see if it was set in the future or the past. Doing so was relatively straightforward as the time value representing the inactive state was always set to the rather easily distinguishable value of Unix epoch by the backend. But despite the relative ease of identifying the inactive state, having to interpret the value of a date time string was far from an optimal way to check the status of the feature.

4.2.3 Logically connected pieces of data were separated from each other in the UI

Some pieces of data, such as a player's IAPs (in-app purchase) and the receipts for those purchases were separated from each other in the layout of Player editor. Based on "The Gestalt principle of

Proximity", distance between related elements can lead to users having difficulties in forming the mental connection between the related elements:

> *The Gestalt principle of Proximity is that the relative distance between objects in a display affects our perception of whether and how the objects are organized into sub-groups. Objects that are near each other (relative to other objects) appear grouped, while those that are farther apart do not. (Johnson 2014, 13-14.)*

When applied correctly, the principle can be a useful design tool to help the user understand the user interface. In some cases, the layout of related objects in Player editor failed to correctly utilize the proximity principle. As a result, the connections between some elements, like the player's IAPs and the respective receipts, may have been difficult to understand. Additionally, some pieces of data that were not related may have seemed related, which also may have contributed negatively to the UX of the tool.

4.2.4 Suboptimal sizing of UI elements

Some numeric value fields in Player editor were considerably wider in size than what was required for showing any sensible values for the variables. One should "match the size of the input fields to the expected length of the answer" (Bargas-Avila, Brenzikofer, Roth, Tuch, Orsini & Opwis 2010, 6). Doing so may help the user to be more certain of the type and format of the desired input.

4.2.5 Poor responsivity of the main editor view

The responsive behavior of the editor view was lacking. Making the browser window narrower made editor elements narrower to the point where some of them would overlap and others become so narrow that the data they were meant to show was only partially visible. Only when the window was made even narrower it adapted to use a different, vertically stacked layout, one that was likely meant for mobile devices.

4.2.6 Unintuitive UI control element usage

Parts of the Player editor misused UI control elements, utilizing them for purposes they were not designed for. An example of this was using disabled dropdown controls as labels for data fields.

4.2.7 Missing tooltips and explanations

Numerous complex parts of the Player editor included no tooltips or explanations for what they were and how they should be operated.

4.2.8 Breaking common UI layout conventions

Some parts of Player editor defied generally accepted UI practices regarding laying out data. There were cases where the values of variables were shown on the left and the labels for the values were on the right.

4.2.9 Lack of a clear indication of current operation environment

The editor operates in two different environments, them being *test* and *production*. While in the test environment, the editor operates only on data that affects the test servers where the only players are the developers and quality assurance personnel. And as indicated by the name, in the production environment the editor operates with real player data. These two environments are used for different purposes. Changes in the test environment data don't affect real players of the game and therefore developers may use the test environment for performing actions that would be destructive if accidentally performed on a production environment player account.

Considering the risks mentioned, the operation environment was not indicated clearly enough in the tool. The only way to differentiate the two environments from each other was to read the address bar of the browser – if the address contained "/test/", the editor was operating in the test environment, otherwise the active environment was production.

4.3 Problems with game terminology

Player editor had several issues with the terminology it used to reference different aspects of the game. Many of the terms used did not match the names of the features in-game but instead were either synonyms of the terms or words with slightly different meanings. An imaginary example of the former problem could be a game in which the player collects *gearwheels* and the term used in the tool would be *cogwheels*. And an example of the latter problem could be if the game featured *lions* and the user management tool called them *tigers*. It is recommended to maintain consistency

in the vocabulary used in a product to help the user be more certain of what part of the product they are dealing with and to lessen confusion (Grant 2018, 320).

Another problem with the terminology used was that Player editor still used some legacy names for things in the game, and the old names conflicted with the current names that were in use in the game itself. A name previously used for thing *X* was being used for another thing *Y* and thing *X* had a completely new name. Table 1 presents an example scenario of this problem. Notice how the word *Mug* is used for one thing in the game and for another thing in Player editor and how some names used in Player editor are no longer in use in the game at all.

Table 1. Fictive example of conflicts between terminology used in the game and legacy naming used in Player editor.

Item name used in the game	**Item name used in Player editor**
Glass	Glass
Cup	Mug
Mug	Beaker
Pint	Jar

Using unintuitive and misleading terminology in data management software may increase confusion and create a risk of misunderstandings and human errors as a result.

4.4 Errors in the behavior of Player editor

Saving changes breaks the editor

When the user chose to save the changes they had made, several parts of the editor changed unintentionally. Some elements disappeared, also causing the general page layout to change, and some controls that were disabled to prevent editing became editable. The problem with some

fields becoming editable was not risking allowing modifications to said data, because the backend would ignore such changes, but it was an additional potential cause of confusion.

5 Improving Player editor

5.1 Selecting the approach for improving the editor

At the beginning of the task there was an important choice that had to be made. One of the following two approaches had to be chosen for improving the user management tool:

1. Keep the existing implementation of Player editor and improve its usability.
2. Choose suitable technologies and use them to completely rebuild the editor.

The first option had the advantage of not having to discard work that had been done for the editor previously, instead being able to benefit from it. Not having to start over would most likely result in a more complete product as development efforts could be focused on improving the existing editor and on adding new features to it. The most notable downside of this option would be having to work with the technologies chosen previously, even if they proved difficult or limiting to work with.

The second option had the advantage of allowing the developer to choose the most suitable technologies to aid in the development of the tool, likely allowing far more flexibility in implementing the editor. The downside would be having to rebuild what already existed, which could easily take a large amount of the time available for the project. This could risk the feasibility of the task because a thesis project has a limited development time—in the worst case, the resulting product could have less supported features than the legacy version.

With these advantages and disadvantages of both approaches in mind, the first approach was selected for the project. Choosing to improve the existing implementation of Player editor meant that development efforts could be focused on improving the user experience of the tool right from the start of the project. The choice also meant that there would be no threat of ending the project with a less useful product than the original one. These advantages were deemed important enough to justify the choice.

5.2 Improving the user interface and the user experience

5.2.1 Categorizing the editor contents and moving to tabbed layout

Originally, Player editor was one, long web form with all editable data scattered around by JSON Editor according to its default layout without any configuration. JSON Editor has a feature for dividing the data to multiple pages or tabs based on categories derived automatically from the structure of the data. The categorization logic used by JSON Editor is rudimentary. It creates a new category for each array property that is a direct child of the root object and for each object property that is a direct child of the root object, and one category for all direct non-array, non-object child properties of the root object.

Changing the editor layout setting to the categorized view without any modifications to the structure of the data made JSON Editor create nine categories, but this result was suboptimal. One problem was that nine pages in the editor was simply too much for the amount of data present. Another problem was that some arrays logically belonged together so splitting them to different tabs was not an improvement over the previous setup. The automatic categorization made by JSON Editor was not adequate, so a manual categorization was needed.

Property categorization using a custom-made extension schema

To manually categorize the user data, a system for manipulating the structure of the data was required. The idea for the implementation of such a system was to add container objects as direct children of the root object and copy all properties belonging to a specific category to the container object representing that category. Such data manipulation can be laborious and error prone though, so a refined solution for making the manipulations to the data was required.

The way that was chosen for the data restructuring was to create another JSON schema that described the custom categories that grouped multiple properties together. The schema definitions of the properties were then moved from the main schema to the extension schema and placed as child properties of the correct category. An example is shown in Figure 1.

```
var originalSchema = {
    "type": "object",
    "properties": {
        "Id": { "type": "integer" },
        "Name": { "type": "string" },
        "Age":  { "type": "integer" },
        "Address": { "type": "string" },
        "Town":    { "type": "string" },
        "FriendStatus": { "type": "string" },
        "Profession":   { "type": "string" }
    }
};

var extensionSchema = {
    "basicInfoContainer": {
        "type": "object",
        "properties": {
            "Name": { "type": "string" },
            "Age":  { "type": "integer" },
            "addressContainer": {
                "type": "object",
                "properties": {
                    "Address":  { "type": "string" },
                    "Town":     { "type": "string" }
                }
            }
        }
    },
    "socialInfoContainer": {
        "FriendStatus": { "type": "string" },
        "Profession":   { "type": "string" }
    }
};
```

Figure 1. Simple example JSON schema and an example extension schema with two main categories and one subcategory

The actual data structure modification then became a simple task of iterating through the properties of the extension schema (and recursively for any object type child properties) and copying the associated data for each child property to the new data object that had an additional object type property for each category to contain the properties. Figure 2 presents an example result of data restructuring according to the schemas shown in Figure 1. The property *Id* is not mentioned in the extension schema and so stays on the root level of the restructured data object.

```
// original data object
{
    "Id": 15,
    "Name": "LassiL",
    "Age":  97,
    "Address": "Wizardstreet 77",
    "Town":    "Gandalfia",
    "FriendStatus": "Acquaintance",
    "Profession":   "Mage"
}

// restructured data object
{
    "Id": 15,
    "basicInfoContainer": {
        "Name": "LassiL",
        "Age":  97,
        "addressContainer": {
            "Address": "Wizardstreet 77",
            "Town":    "Gandalfia"
        }
    },
    "socialInfoContainer": {
        "FriendStatus": "Acquaintance",
        "Profession":   "Mage"
    }
}
```

Figure 2. Example data before and after restructuring according to the extension schema shown previously in Figure 1

For saving the changes made to the data in the editor, this process had to be reversed – the data had to be copied to an object with the original data structure. The extension schema was used for this as well, in a similar way as when restructuring the data, only inversed.

The system also allowed further organizing properties on a single “page” to separate subcategories to improve the readability of the UI by establishing visual hierarchy. As a result of these changes the editor already looked a lot cleaner and easier to understand. The tab-based UI for the categories can be seen in Figure 3.

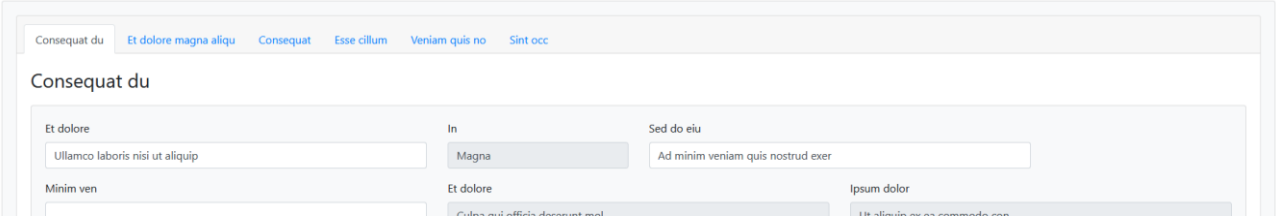

Figure 3. Each category of properties has its own tab (the row at the top).

5.2.2 Removed “hidden knowledge” requirement for gifting upgrades

The problem with having to know a “magic number” of collectibles for unlocking an upgrade for the player (problem described in chapter 4.2.1) was solved by adding a “give” button for each upgrade that, when clicked, would automatically set the correct number of collectibles for the player to unlock the upgrade.

The button was added by utilizing the button editor component of JSON Editor. A new property was added to the upgrade schema defined inside the *upgrades* array property of the player schema. The whole setup is displayed in Figure 4.

```
{
    // ...
    "upgrades": {
        "type": "array",
        "title": "Upgrades",
        "items": {
            "type": "object",
            "id": "upgrade",
            "properties": {
                // for the sake of brevity, other upgrade properties,
                // such as name, have been excluded from this example
                "collectablesCollected": {
                    "type": "integer",
                    "title": "Collectables"
                },
                "giveUpgrade": {
                    "type": "button",
                    "title": "Unlock upgrade",
                    "watch": {
                        "collected": "upgrade.collectablesCollected"
                    },
                    "options": {
                        "button": {
                            "action": "giveUpgradeCallback"
                        }
                    }
                },
            }
        }
    }
    // ...
}
```

Figure 4. The upgrades array schema with the "Give upgrade" button for each upgrade

The property *watch* is a JSON Editor specific extension to the JSON schema that ties into the JSON Editor watcher system that allows making properties aware of changes to the values of other properties. The use of the property here is to provide a method for the callback function of the *giveUpgrade* property of each of the upgrades in the *upgrades* array to access the associated *collectablesCollected* numeric input field to be able to edit its value. The JSON Editor setting property *action* references a button callback function defined in *JSONEditor.defaults.callbacks* as seen in Figure 5. The variable *data* is a global object containing the different configuration values in the game.

```
JSONEditor.defaults.callbacks = {
    "button": {
        "giveUpgradeCallback": function(button, e) {
            let required = data.Upgrades[button.parent.key].requiredCollectables;
            let collectableCount =
                    button.jsoneditor.getEditor(button.watched.collected);

            if (collectableCount) {
                collectableCount.setValue(required);
            }
        }
    },
    // ...
}
```

Figure 5. The callback function for the "give upgrade" button

5.2.3 Improved workflow for managing the status of a limited time benefit feature

A problematic workflow for enabling and disabling a limited time feature for a player was amended to be cleaner and simpler to work with. The old implementation required the user to manually modify a date time field, and the way to know if the feature was enabled or disabled was to check whether the date was set to a “magic value” of Unix epoch to indicate “disabled” or not.

The improved solution was to add a checkbox labelled as “Has feature X enabled” that would be automatically selected if the feature was active and that could be used to toggle the feature on or off for the user. The date field was also preserved to allow troubleshooting any possible problems with the expiry time. The two elements were grouped together in a properly named sub-container (similar to an HTML fieldset) in the form to make it easy to see that they were related to the same feature in the game. The grouping was achieved by utilizing the newly added custom property categorization method (described in chapter 5.2.1) and its support for categorizing properties further into sub-categories in a recursive manner.

5.2.4 Fixed terminology to match the naming used in the game

Two cases where the editor used synonyms of terms used in the game were solved by using the JSON Schema keyword *title* to explicitly define the names shown for the properties in the editor.

The keyword is a standard JSON Schema feature which JSON Editor uses for the purposes of building the editor UI.

Additionally, a group of items that, in the editor, were referred to with their backend identifiers were updated to use the in-game names. The issue was solved by using a hard-coded dictionary for converting the identifiers to the in-game names as the in-game names were not included in the data available to the Player editor. Generally, resorting to a solution as such is undesirable due to requiring manual maintenance. In this case though, the choice was deemed justified since the associated data was not likely to change often and because a reliable fallback behavior was easily implemented for cases where a new item was added but had not yet had its in-game name added to the dictionary. In such cases the code would just fall back to using the backend identifier in the UI.

5.2.5 More intuitive UI control element usage

Some of the data in the editor was previously labelled using disabled dropdown elements. The reason for the use of dropdowns was that it allowed setting the value of the dropdown according to a value of an enum in the data JSON object, which allowed using enum values as titles for a collection of similar, but separate data values. No similar feature was available for plain text elements, so the original developers of Player editor had used what was available.

Misusing UI control elements may have a negative impact on the readability and understandability of a web form. The problem was solved by using JSON Editor's built-in property type "*info*" which displays the given contents as an HTML label element. But with this approach there was an issue that had to be worked around: as previously stated, the "*info*" editor type in JSON Editor, being meant to display static explanation text content, did not support matching with data in the JSON object, the enums associated with the data values, for example. The solution was to keep the old implementation but just set it to be hidden from the UI and then use the JSON Editor's watch system and "*headerTemplate*" feature in the info editor to read and inject the value from the hidden and disabled dropdown to the info editor label element.

The *headerTemplate* keyword is a JSON Editor feature that allows dynamically constructing header strings that will then be displayed as HTML labels for individual property editors in the main

editor. A template specified using the keyword can access the values of all properties specified to be watched by the current property using the watch system provided by JSON Editor, as well as the value of the current property itself.

To simplify the addition of these semantically correct textual label elements, a function returning a JSON Editor compatible JSON Schema object with its properties automatically set as needed was created. The function "*createTextLabelFromSource*" takes up to three arguments:

1. *contentSource*, which is the JSON Schema path to the property to use as the value for the text to display
2. *title*, which is the title for the label element (used as the table column heading for cases where the text label created is used with a different value for each row in a table of data)
3. *inputWidth*, which is the width to use for the label (optional)

The way the function was used was to add a new property name to the schema but instead of manually specifying the configuration object for the new property, a call to the function was placed in the schema.

5.2.6 Visual and layout improvements

More intuitive label placements

Some of the data in the editor had the label – value layout order backwards: the value was presented first, and its label was listed after it. All such cases were remedied by swapping the placements of the values and their labels. An example case of this problem was the inventory-like view. The old and new versions of the inventory-like view can be seen in the comparison screenshots in chapter 7.4.3.

Added whitespace

CSS padding and margin was used to improve the look and feel of the editor. For example, the main editor view was given some horizontal and vertical margin to add some space between the HTML elements and the edges of the window, which helped make the layout feel lighter and easier to read.

Improved player search view

The player search view was a crude HTML page with no custom CSS styling applied and with elements positioned to the top left corner of the browser window by default. Selected Bootstrap utility classes were applied to the player search form to move it to the center of the window and enclose it in a rectangular visual container. The game logo was also added to the player search view to make it easy to quickly grasp what game the tool is associated with. The form controls were automatically upgraded to the default Bootstrap visuals with the addition of Bootstrap to the web page. The old and improved implementations of the player search view are illustrated with screenshots and compared in chapter 7.4.1.

5.2.7 Adjusted the responsive behavior of the editor

The support team members expressed the need to be able to have two editor views open side-by-side for cases where comparing two player accounts is required. Naturally, the basic level of such functionality is supported by any modern web browser, but the request also placed some requirements on the implementation of the editor. Firstly, the editor had to support viewing multiple player accounts at the same time in different browser tabs or windows. This was made trivial by the fact that JSON Editor operates without the concept of user sessions, which meant that each instance of Player editor operated in complete separation from each other. Secondly, the editor layout had to be responsive so its usability would be maintained when operating within a narrower viewport.

Problems

There were several problems in the responsivity of the editor when using the default layout of JSON Editor. When the viewport was made narrower, editor elements only reacted by shrinking horizontally, often to such extents that either the overflowing content became hidden or text elements would be split to multiple rows, even mid-word, noticeably impacting readability. Eventually several elements would begin to overlap with each other, and the layout would break in other ways too. Upon investigating the problems, it turned out that the issues traced back to the way the selected JSON Editor theme utilized Bootstrap for the automatic layout of the editor, and more specifically to how it employed the Bootstrap grid layout concept.

Bootstrap grid layout

Bootstrap's grid layout system is based on the idea of dividing the available horizontal space into 12 columns. Each element on a row of a Bootstrap grid layout can be specified to use a given number of the columns available, as long as the total number of columns used on a row does not exceed 12, which would cause the overflowing elements to wrap to a new row. (Grid system n.d.)

Problems in JSON Editor usage of Bootstrap grid layout

JSON Editor offers a selection of editor layout options for the developer to choose from. As already mentioned in chapter 5.2.1, the top-level layout option that was chosen was the "categories" format. JSON Editor also supports using different layout formats for nested objects, such as the container objects that were used to group the player data properties to meaningful units. The layout format chosen for the container objects was a grid-based layout. It utilized Bootstrap's grid layout system.

The Bootstrap grid layout as applied by the built-in theme in JSON Editor was not very responsive. Making the browser window narrower downsized the editor fields according to the column settings they were given, which, in most cases meant that eventually the fields would just be very small, hiding parts of the data they were meant to display. This was a direct result of how JSON Editor assigned columns to different editor components. Many of the editor components were assigned a static number of columns. The system did not take advantage of the responsive design options provided by Bootstrap, which would have allowed allocating a varying number of columns for a given element depending on the width of the viewport. As a result, the UI controls were excessively wide in some situations and too narrow in others.

Solution overview

The problem was solved by making a customized version of the selected built-in JSON Editor theme that was made with Bootstrap. In the customized version, four preset "classes" for responsive element widths were defined: three size settings and a value allowing setting a custom column number manually if required. The size "class" could then be set for each property in the schema via the "*options*" configuration property.

The size classes were implemented by repurposing an existing configuration option in JSON Editor. The option that was overridden, "*grid_offset*" was itself also related to the Bootstrap grid system, which meant that it was a good candidate for customizing the theme behavior as the modifications would be made only to code that already was working with the grid system. The original usage of the option was to specify the number of columns that should be left empty on a row before the element in question. The original use was deemed to be such a rarely used feature that the editor could be implemented without having it available.

Technical implementation of the solution

As previously stated, the solution was implemented as a customized version of a JSON Editor built-in theme. The only difference between the original theme and the customized version is that the implementation of a method called *setGridColumnSize* was altered. The purpose of the function is to apply the required Bootstrap classes to the given editor element to utilize the Bootstrap grid layout system.

5.2.8 Adjusted input field widths

Multiple input fields were much wider than what was required for displaying any realistic values for the associated properties, making the form needlessly difficult to read. The root of the issue was that the default layout of JSON Editor was simplistic, often extending elements horizontally to fill all the space available. Luckily, the tool offered the option to manually configure the width of the editor for any property in the schema via the option property "*input_width*". Example usage of the configuration property is presented in Figure 6.

```
{
    "ExampleProperty": {
        "type": "integer",
        "options": {
            "input_width": "7em"
        }
    }
}
```

Figure 6. Manual configuration of the width of an editor component

The downside to the above approach is that it clutters the schema with presentation-related information. After evaluating the situation, it was decided that, given the relatively low number of instances where the option was needed, it was preferable to prioritize the improvements in the look and feel of the editor over maintaining the strict data-specificity of the schema. The choice was aided by the fact that all the cases where the option was needed concerned data that was displayed in a table format. Thus, adding only one instance of the option to the schema allowed controlling the width of an entire table column, which meant that the benefits of using the option expanded to multiple rows of data. The results of using the option can be seen in the comparison screenshots shown in chapter 7.4.3.

5.3 Fixing errors in the editor functionality

Fixing the problem with saving changes causing the editor to break

There were two issues that occurred upon editor reload after saving changes to user data. The first problem was that some of the user data that was originally shown in the editor was left out on reload. And the second issue was that all data fields that were meant to be read-only became editable.

The disappearance of previously shown data was caused by a mistake in the editor backend code that resulted in the backend sending a different configuration of player data back with the response to the save action. The configuration was incorrectly set to leave out some of the data that was present in the initial player data response that was used when first opening the Player editor view. When JSON Editor then rebuilt the editor based on this new player data object, the excluded data fields were not rendered at all. The problem was solved by simply setting the correct configuration for the player data response in this case.

The problem with multiple read-only fields becoming editable was a result of the fact that the associated form elements were manually disabled upon the initial editor load, and this procedure was not repeated after the editor was reloaded. The issue was solved by a combination of two actions: firstly, a JSON Editor schema keyword was used to declare data fields as read-only, and secondly, where the previous was not possible, the procedure of disabling the fields manually was repeated. Using the JSON Editor keyword was the preferred way to mitigate the issue as it was the

intended way of marking data as read-only that also made JSON Editor render the associated form elements as disabled. Unfortunately, the keyword was not usable for any data in array format, because it would also prevent adding new items to the array. For all array format data, the custom script that was used to disable editing specific fields in the editor was extracted from the editor initialization function to a new function which could then be called separately in both cases, the initial editor load and the reload after the save action.

6 Implementing a major new feature: Change logging

6.1 Overview of the change logging feature

6.1.1 High-level description

The change logging feature is the most notable new feature that was added to Player editor in the development project. The idea of the feature is simple: it maintains and displays a textual history of support actions performed for a player account. An optional explanatory comment can be added to each change log entry to allow stating intention behind the change. In addition to the change log entries, the system also allows adding and storing plain support comments. A plain support comment is, in essence, just a special kind of change log entry not associated with any specific change.

6.1.2 Background and rationale for adding change logging to Player editor

The feature was added by request from the support team of the game. In the old Player editor, there was no action log at all and the only feature that could be used for adding explanations or notes for other members of the support team was *SupportComments*, a text-type property field in the player data, where a support team member could write a short note for other support personnel.

The old support comments feature was very limited in practice however, mainly because it was a one-line text input field which meant that the message written in it had to be very short if it was to be easily readable. Reading longer messages was cumbersome because it required horizontally scrolling the text in the input field. For the same reason this old implementation did not really

support having multiple comments because adding a new comment meant just adding more text to the one-line message.

6.1.3 Technical overview of the designed solution

The change logging feature was custom-made for Player editor. It was written in JavaScript and utilizes the watcher system of JSON Editor to detect changes to the data shown in the editor web form and the diffing functionality of jsondiffpatch to analyze the change that happened.

The watcher system is set to watch for changes in the player data and to fire a given callback function when it detects a change. The callback function is given a copy of the original data of the changed property as well as the updated data and it compares them to detect precisely what part of the data was changed. It then creates a textual representation of the change, shows it to the user and offers the user the possibility to write an optional explanatory message for the change. The change is then added to a list of unsaved changes that is displayed to the user at the top of the editor view. When the user saves their work on the player data, all the changes listed in the unsaved changes -view are added to the change log which is then saved together with the rest of the player data.

Change log save system

The previously mentioned *SupportComments* property was repurposed to be a hidden property containing the entirety of the change log history. When the user saves the changes they have made, the unsaved changes are added to the change log. The log data is then converted to a condensed text format and written to the *SupportComments* property with a special syntax that allows parsing the individual change log entries for building the change log user interface when the player data is viewed next time. The special syntax is explained in chapter 6.4.2. A downside of this quick-to-implement change log storage solution was that the property had a limit in how many characters of text it could contain.

To prevent the limitation from causing problems, the system detects if there is not enough space for new change log entries and offers to start removing the oldest entries from the log to make space for new ones. Alternatively, the user can remove any changes from the change log manually

before saving the new changes. This solution for handling the case where the limit was reached was rather poor from a user experience point of view as it stopped the user and made them decide on a question that ideally should not be of their concern at all. The reasoning for choosing to use this approach despite the downsides is presented in chapter 8.1.3 when discussing the remaining issues in the tool.

Automatic detection of change reversal

The system automatically detects if an unsaved change is reversed and removes the associated change from the list of unsaved changes. For example, if an originally unchecked checkbox was first checked and later unchecked without saving the player data after the first action, the system would remove the first change from the list instead of adding a new one stating that the checkbox was unchecked again. This way the only changes that are written to the change log are ones that accurately reflect differences in the player data before and after using Player editor to modify it.

6.1.4 Challenges in implementing the feature

The task of implementing a system that, as described, would automatically detect, analyze, and log changes to user data was rather complex. A large portion of the complexity arose from the fact that the editor handled many different types of data that also differed in what kind of changes could be made to them. For example, basic value properties only could have their value changed, but an array of items could have items removed or added. These and other such variances in property behaviors meant that changes to properties had to be handled in specific ways based on their type. This problem and its solution are discussed in more detail in chapter 6.2.

The JSON Editor-provided watcher system that was used for detecting changes and calling callback functions when a change was made also introduced some additional difficulties into implementing the automatic change logging feature. The main issue with the watcher system was that in certain cases it would repeatedly fire the callback function, creating false change detections. The way the problem was mitigated is described in chapter 6.3.

6.2 Detecting and handling changes to different types of data

6.2.1 The problem with handling changes to different types of data

There was three points in the change detection and logging procedure that required customized logic for handling some of the player data properties. First such point was the change detection step, also referred to as diffing, where the ways of comparing old and new data differed depending on the type of data being compared. The other two points concerned the automated generation of a human-readable description of the change being made.

The solution used for handling the different types of data was to implement a way to define customized behavior for the parts of the player data that required it. Given that there were only three steps in the procedure that required such fine-tuning of implementation logic it was decided that the main change detection and logging procedure would be shared between all types of data. The adjusted parts of the procedure logic would be implemented as callback functions that would be automatically called based on the data being handled.

The designed solution required specifying which parts of the player data required customized logic as well as linking each implemented type of customized logic to the data that requires it. Three new concepts were introduced for these purposes. The first new concept was "*a change object*", the second one "*a change batch*" and the third one "*a diffing configuration*".

6.2.2 The concept of a change object

A change object is a self-explanatory concept: it is a simple JavaScript object that contains information about a single change to the player data. It stores the information of what property was changed and how, when the change was made, a textual representation of the change and an optional, user-provided message about why the change was made. Each change object represents a single change to the user data.

6.2.3 The concept of a change batch

A set of one or more player data properties that require the same special treatment in the change detection and logging procedure is considered a change batch. An individual change batch is

defined as any select property in the player schema. If said property is an object that is used as a categorizing container for other properties, all the sub-properties are also included in the change batch. The categorizing system was explained in chapter 5.2.1.

It is worth noting that this general concept of a change batch is merely an abstract construct – the player data properties that form a change batch are never collected to a new data structure. The purpose of the concept was to act as a term that described a part of the design of the change logging system.

The system stores all change objects into a single JavaScript object, where each key–value pair has the name of a change batch as the key and an array of change objects belonging in the associated change batch as the value. So, the object contains an array of change objects for each change batch. Such an array of change objects is also referred to as a “change batch” but in the meaning of “a batch of change objects” instead of “a batch of player data properties being monitored for changes”. An example of the structure of the main object that stores all the change objects is shown in Figure 7.

```
{
    "playerBasicData": [
        { /* change object for a change in basic player data */ },
        { /* change object for a change in basic player data */ },
        { /* change object for a change in basic player data */ }
    ],
    "playerInventory": [
        { /* change object for a change in player inventory */ },
        { /* change object for a change in player inventory */ }
    ]
}
```

Figure 7. A simplified example of the main object that holds all change objects

6.2.4 The concept of a diffing configuration

A diffing configuration is a JavaScript object that contains the special change detection and logging logic to use for the properties in its associated change batch. A diffing configuration object has

three function-type properties, *diff*, *nameCallback* and *getChangeText*. A function must be assigned or defined for each of these function properties.

Each change batch is associated with a single diffing configuration. All diffing configurations are defined as properties of a single JavaScript object used to hold them, the object being called "*diffConfigs*". The property name of each diffing configuration is the name of the associated player schema property. This way, the name given to a diffing configuration also declares the change batch associated with it.

6.2.5 The relation between a change batch and a diffing configuration

The new concepts and their relations are visualized in Figure 8. The colored dots in the player schema represent properties. The only non-root properties that have a name shown for them (on the right of the dot) in the illustration are the ones that have been chosen to form change batches.

On the right is an incomplete pseudocode example of the main diffing configurations object that defines and stores all the diffing configurations. In this example the diffing configurations define four change batches in total, one of which is defined by the container property "propA" and thus contains all its child properties, recursively. The other three are defined by individual properties in the player schema with "propC" being an array-type property storing string items and "propD" being another array-type property storing objects, as evidenced by the diffing functions selected for their associated diffing configurations.

Figure 8. Visualization of the relationships between the player schema, change batches and diffing configurations

6.2.6 Details of the components of a diffing configuration

The diffing configuration property *diff* was used to set or define the diffing function that should be used for the data in its associated change batch. In total, there were three different diffing functions that were used depending on the type of data that was being compared. The most used diff function was the one provided by JSONdiffpatch, which was fine for most of the properties of the player data. The other two diffing functions, *diffStringArray* and *diffObjectArray* were custom-made to circumvent problems in how the diffing function provided by JSONdiffpatch handled arrays.

The main issue with the JSONdiffpatch diffing function was that in order to create meaningful diffs of arrays, it required unique identifiers for each array item. Due to how JSON Editor operates, no such identifiers were available to the diffing function and because uniqueness of array items was not guaranteed, the values themselves were also not suitable for the use. As a result, the JSONdiffpatch diffing function would fall back to merely comparing the items in corresponding indices in the old and new arrays. The fallback behavior would break if an item was removed from the array, because the indices of the rest of the items would change and the function would be comparing the wrong items, as exemplified by Figure 9.

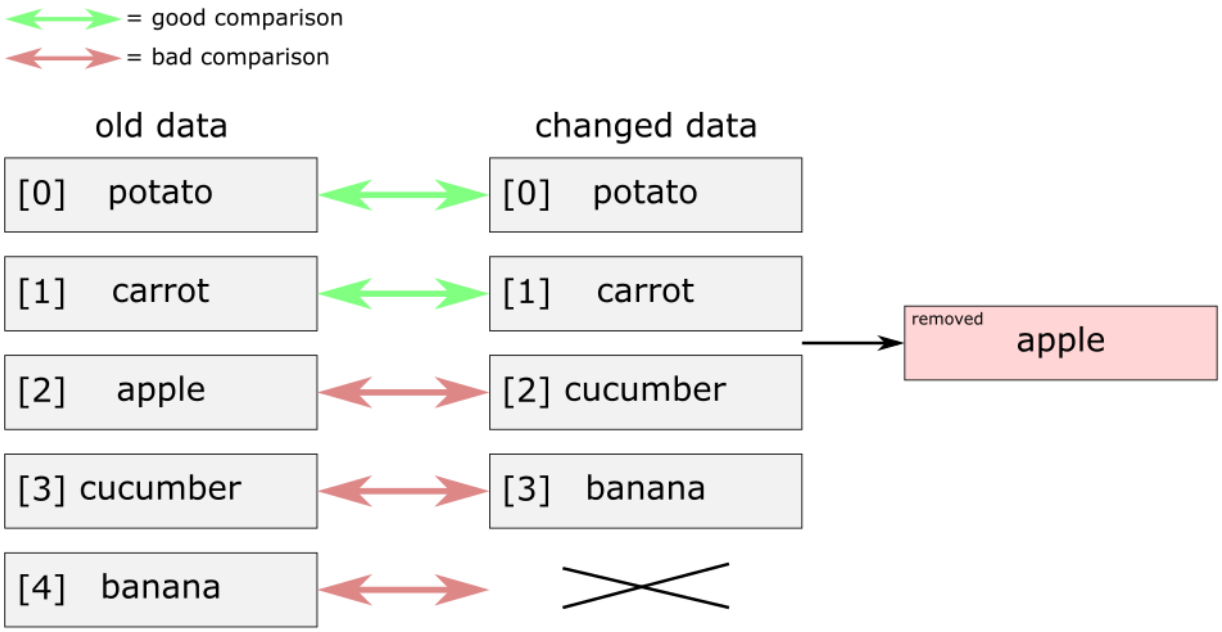

Figure 9. Visualization of an issue in the process of diffing arrays using the jsondiffpatch diff function without unique identifiers for array items

The optional diffing configuration property *nameCallback* allowed defining a callback for constructing a custom name to use in the change log for the property that was changed. Its primary use was to allow using an identifying property of an object to differentiate which object in an array of objects was the one that had its property value adjusted. An example of what this meant for an inventory-like view is shown in Table 2.

Table 2. Fictive example of the importance of the name callback for change log entries concerning the inventory-like view.

Change log text without name callback	**Change log text with name callback**
Amount changed: 5 → 6	Red potion amount changed: 5 → 6
Amount changed: 15 → 20	Green potion amount changed: 15 → 20

The third property, *getChangeText* was for defining a callback function for constructing a human-readable description of a change that was made to player data. Given the diversity of types of properties in the player data, it was necessary to provide a way for the developer to specify suitable change text formats for each change batch. The callback function is called when the change texts such as seen in Table 2 are being created. The callback implementation allows defining a sensible change text format for changes in each change batch.

6.3 Filtering array modification false detections

The JSON Editor watcher system implementation had some problematic qualities in relation to the behavior of arrays in JSON Editor. Mainly, the way JSON Editor handled adding or removing rows from a table representing an array of data did not work together with the watcher system the way one would have expected. The expected case would have been for the given callback function to be called once for each addition or removal from the array. But instead, what happened, was that the callback function was invoked numerous times. This happened because the way JSON Editor removed a row from a table was as follows:

1. Remove the selected row.
2. If there's a row beneath the removed row, remove it and add it back as a new row in the position of the previous row.

JSON Editor would repeat the pattern above until all the remaining table rows were shifted backwards by one place. In other words, when a row was removed from a table, all the remaining rows were shifted upwards by removing them temporarily and inserting them back to the table in a new position. The problem this caused was that the watch system detected each removal or addition operation as a change made to the data and as a result called the change watcher callback function repeatedly, once for each operation performed. This created numerous false change detections reporting for each row below the deleted one that the row was removed and then added back. The false changes had to be filtered from being added to the change log.

A simple yet reliable approach was chosen for filtering the false change detections. Instead of allowing the change detection system to directly call the callback function that added a new change to the list of unsaved changes, an intermediate step with a small delay was introduced. With the delay in place, if another change to the data in the same category was made before the timer had

expired, the old diff was discarded in favor of the new one and the timer was restarted again. This way each involuntary change made to a table representing an array of data got discarded until the only diff remaining was the one that contained the information of what really was removed from – or added to – the table. Each change batch had its own timer instance to ensure that a quick change to a property in a different change batch would not stop a yet unconfirmed change from another change batch from being added to the list of unsaved changes.

6.4 Change log user interface

6.4.1 Overview

The user interface of the change log is a foldable table added to the top of the editor view. For player profiles with no previous support actions performed on their data, this table is not shown to conserve vertical space on the web page. Each row of the table contains a timestamp indicating when the change was made as well as a textual representation of the change and the comment field showing the optional comment in an editable format. Each row also had a button that could be used to toggle the status of the log entry: whether it should be removed or preserved upon saving, with the default choice being to preserve the log entry.

The “unsaved changes” -view is a table of new log entries that will be added when the modifications to the data are saved. The table is similar to the change log table, positioned above it and shown only when there is unsaved changes to show.

6.4.2 Technical implementation

As mentioned in the technical overview of the change log system, the feature was built by repurposing a text type property that was reserved for the support team’s internal comments about each player account. The new use of the property is similar in the sense that it still can be used also for storing comments by the support team, it just now also stores data of the changes made to the player account data.

All the comments and log entries are stored as a single string with a special syntax that makes it easy to parse the data. The structure of both, comments and log entries is as follows:

1. Timestamp of creation as milliseconds elapsed since January 1, 1970 00:00:00 UTC as returned by JavaScript's built-in *Date.now()* method (Date.now() – JavaScript 2023).
2. Change text, which in the case of comments is "*Support comment:*" and in the case of change log entries is a textual representation of the change that was made.
3. An optional comment about the change made, or in the case of a support comment, the comment text itself.

These three parts of a single entry in the change log are separated from each other by a tabulation character "*\t*". And different entries are separated from each other by the vertical bar character "|".

Additional logic was programmed for handling any comments stored in the *SupportComments* field before the inception of the change log feature. The old value, if present, is converted to a support comment with a special change text message: "*Recovered pre- change logging Support-Comments value:*".

7 Results

7.1 Difficulties identified in using JSON Editor as a basis for a user management tool

To answer the first research question, it was imperative to analyze Player editor and identify the problems in it. After doing so it had to be determined which problems were either a direct or an indirect result of using JSON Editor as the basis for the editor. The evaluation was important because to properly answer the research question a distinction had to be made between problems in Player editor that were rooted in JSON Editor and ones that were results of oversights or shortcomings in the implementation of Player editor.

Multiple problems in the user experience of the editor were a result of JSON Editor offering rather limited options for adjusting the layout of the editor. The tool is designed to perform most of the layout work automatically, supposedly to make it quick to set up and use the system for managing JSON data. The unfortunate downside of the approach is that the default layout constructed by the tool is not easily adaptable for different layout requirements in the often-unavoidable cases where the generated layout is suboptimal in terms of readability or usability. The framework supports implementing custom themes, but custom theme creation is not documented in depth (jsoneditor Wiki: Theme 2022), and as a result would require examining the source code of the built-in

themes of the framework for reference. Additionally, even if a custom theme was created, certain problems would not be possible to overcome that way, because some of the automated layout decisions are tied to the structure of the given JSON data itself on the implementation level of the framework. A notable example of the previous being the *categories* format option for the *object* type editor basing the categorization decisions on the structure of the given JSON data.

The issues that were a direct result of basing Player editor on JSON Editor included things such as poor responsivity of the editor, confusing and disorganized editor layout and the previously mentioned problem with the automatic categorization of the data based on its structure.

Among the problems not linked with JSON Editor were conflicts between new and legacy terms for referring to features in the game as well as the need for the user to either memorize or have access to an external reference of a set of “magic number” identifiers to be able to manage a player’s inventory of various in-game items.

7.2 Fixing JSON Editor’s shortcomings

The answer to the second research question is in the solutions used for overcoming the JSON Editor derived problems and limitations in Player editor. The solutions were described in detail in chapter 5. To summarize, the actions taken for solving the problems in Player editor can be divided in two categories.

The first and simplest way that was used to fix smaller problems in the editor was to better utilize JSON Editor’s own editor configuration schema extension keywords for adjusting the look and behavior of the individual components of the editor. Researching JSON Editor documentation for a feature or a set of features that would allow solving a specific problem in a JSON Editor -native way was always the starting point when considering how to amend an issue in Player editor. The reason for preferring this approach was that it was the easiest and the cleanest: native solutions supported by the library can be assumed to be typically the most robust. An example case where this approach was used is adjusting the editor input field widths by utilizing a JSON Editor built-in schema keyword as described in chapter 5.2.8.

The second approach to solving issues in the editor was programming new systems for overcoming the limitations and shortcomings of JSON Editor. This way of fixing issues was less desirable due to the amount of additional work required but was necessary for several cases where no better options were available. The custom-made data restructuring system described in chapter 5.2.1 is an example of this way of overcoming JSON Editor's limitations.

7.3 New feature showcase

7.3.1 Change log

One of the main new features that was implemented is the change log. Any changes to the player data made by the user of the tool are saved to a list, with optional comments attached with them. The list of previous actions is shown whenever a user views the data of the same player again, allowing the user to inspect what operations have been performed for that player previously. Together with the change log feature, a support comment feature was added as well, which allows the user to add general comments about the player. The change log and support comment feature are documented in more detail in chapter 6.

An example view of a change log can be seen in Figure 10. The first row in the table shown below is a support comment, which is a text-only notice written by a user of the tool about the player. Support comments are not associated with any specific change made to the user data. The second table row in the screenshot has been marked to be deleted, which means that once the user saves the changes they have made, that change log entry will be permanently removed from the log. It is worth noting that removing a change log entry does not revert the change itself, only the log entry associated with it is discarded. The "fold" button allows collapsing the table.

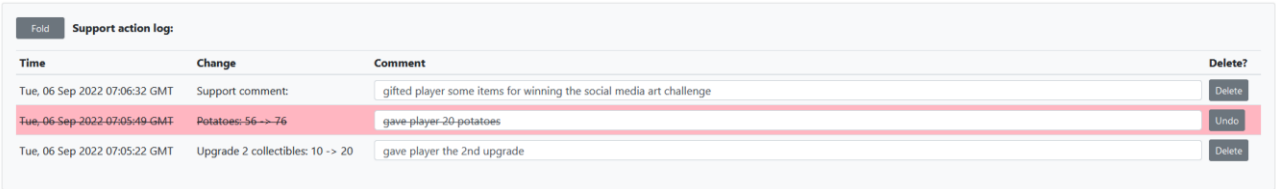

Figure 10. A view of the change log with two changes and a comment

7.3.2 Unsaved changes view

The unsaved changes feature is closely tied to the change log feature. Any unsaved changes made to the player data are shown in a table at the top of the editor view and – unless marked to not be saved – will be stored as new change log entries upon saving the changes made to the player data. The feature and its implementation are discussed in more detail in chapter 6. The “unsaved changes” -view is visually similar to the change log view, but with a different title and coloring. An example of the “unsaved changes” -view can be seen in Figure 11. The same changes are shown in Figure 10 as change log entries after having been saved.

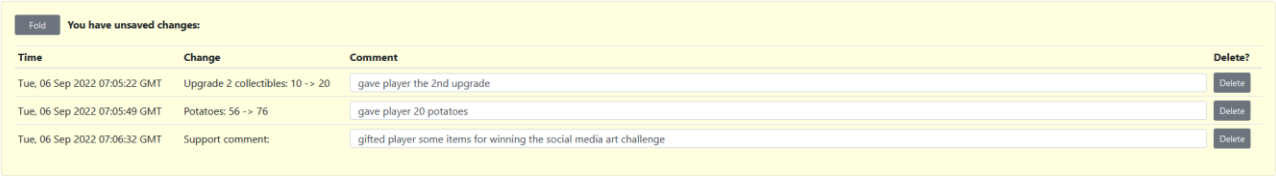

Figure 11. The unsaved changes view with two changes and a comment

7.3.3 Automatic change detection

The automatic change detection feature is also tied to the change log feature. Any time a change is made to the player data, the feature gives the user the option to write a comment explaining the reasoning for making the change. If a comment is given, it will be shown in the “comment” field of the “unsaved changes” -view, and after saving the changes, in the “change log” -view. A more thorough explanation of the change detection system is given in chapter 6.

The comment request is made using the *prompt* instance method of the *Window* interface of the web browser (Window: prompt() method – Web APIs 2023). As a result, the dialog looks different on different web browsers. The dialog as it appears on Firefox can be seen in Figure 12. In the future, a more elegant solution should be considered, such as using the *dialog* HTML element, which would also allow using custom CSS styling to make the visual look of the dialog match the look of the editor (<dialog>: The Dialog element - HTML: HyperText Markup Language 2023). Another point worth considering is if the automatically opening prompt for an explanatory comment may become frustrating or hinder the user accomplishing their task.

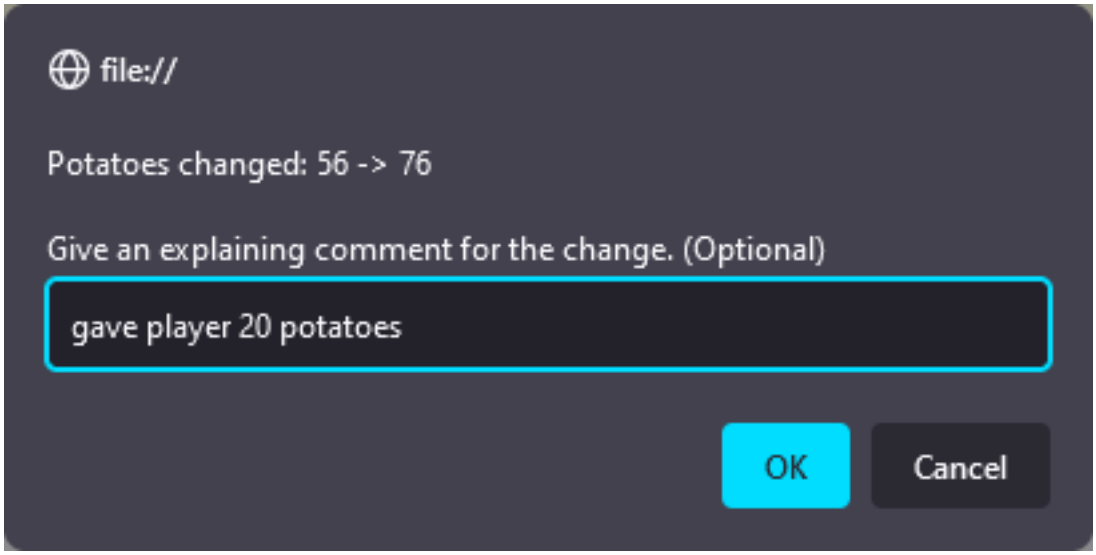

Figure 12. Automatic change detection comment prompt as it appears on Firefox

7.4 Comparisons between old and improved version

7.4.1 Player search view

The original player search view seen in Figure 13 was a crude HTML form with no custom CSS styles applied that was automatically positioned to the top left corner of the browser window.

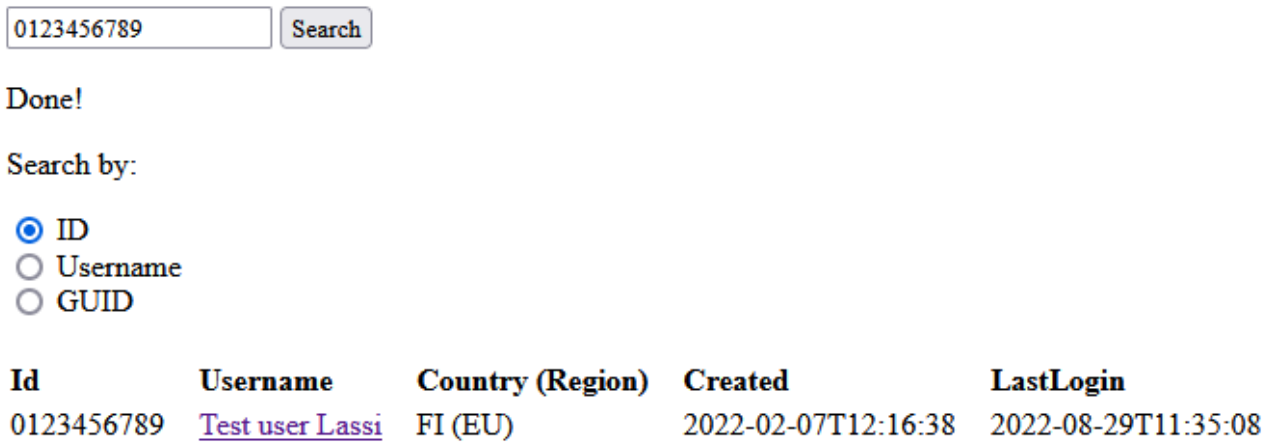

Figure 13. Player search view, original implementation

The view was partly remade and some of the Bootstrap classes used in the main editor were also applied to the player search view. Additionally, an image of the game logo was added to the top of

the search view together with a text displaying the current operation environment. The improved player search view is shown in Figure 14.

Figure 14. Player search view, improved

7.4.2 Editor main view

The Player editor main view was a single, long web page with all the inspectable player data displayed at once. Navigating the extended editor page involved excessive scrolling which in turn made the editor feel unwieldy and tiresome to use. Another issue was that the layout was unorganized and there was no visual hierarchy to help the user understand the connections between different aspects of the player data. And the fact that there was no indication of the current operation environment of the editor in the editor view itself presented a heightened risk of the user mistaking the test environment for the production environment or vice versa. These and other issues of the main editor view were detailed in chapter 4.2. The old editor main view is shown in Figure 15.

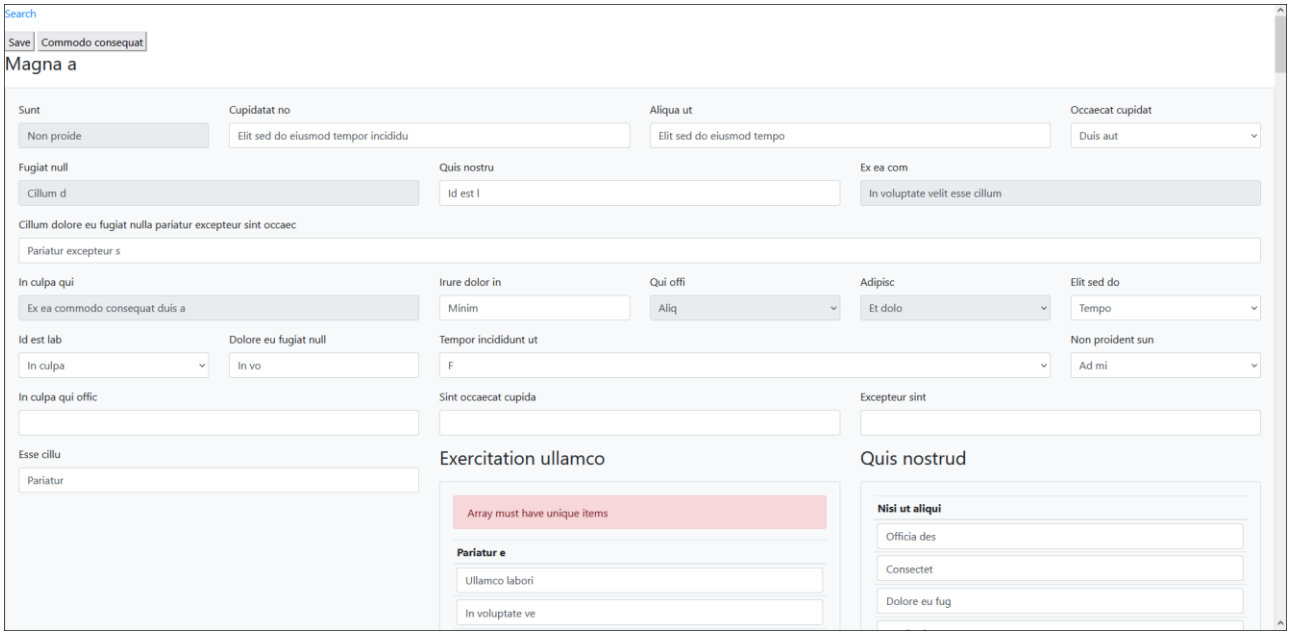

Figure 15. The old editor main view

The page length and the poor readability of the unorganized layout were solved together by sorting the data to multiple subcategories and then using JSON Editor's layout options to display the data on multiple "pages" accessible via tabs on the top of the editor view. The details of the implementation of the solution were presented in chapter 5.2.1. Additionally, a label indicating the current operation environment of the editor was added to the top of the editor view. The improved main editor view is shown in Figure 16.

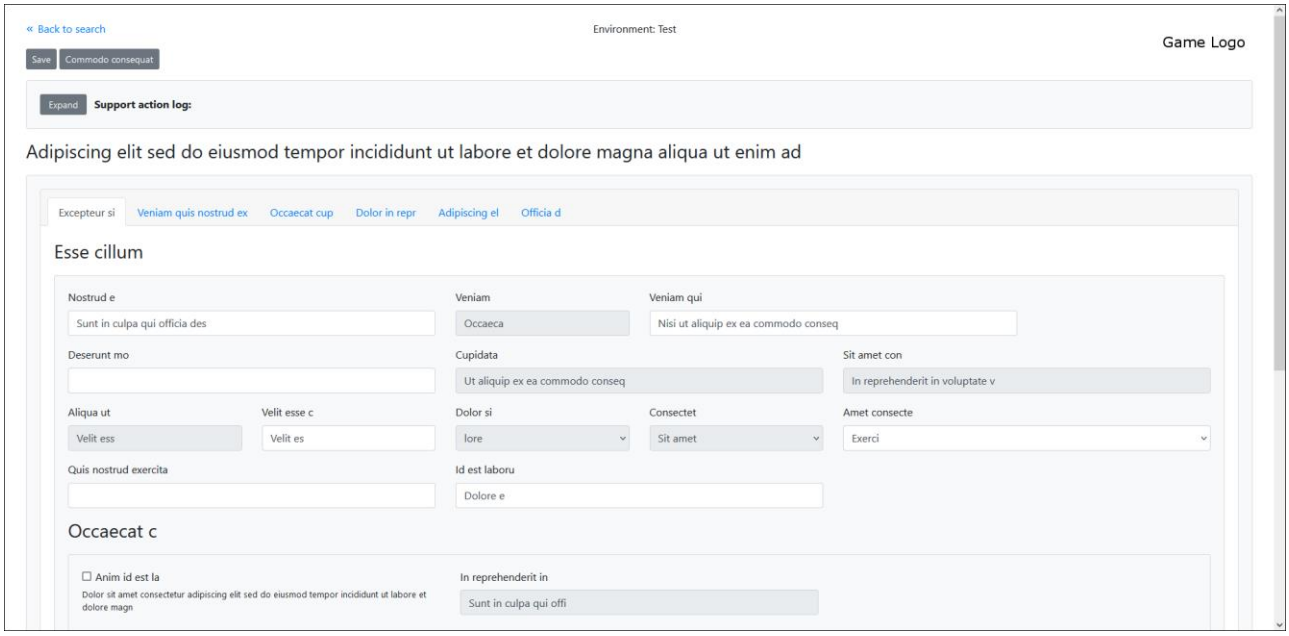

Figure 16. The new editor main view

The move to the categories-based tabbed layout greatly reduced the vertical length of the editor web page, lessening the need to scroll up and down when viewing or editing the player data. This difference is showcased in Figure 17, where the red rectangles represent screenfuls of editor view. In the illustration, the old editor view on the left displays all the data of a player that possesses many in-game items, causing the editor to stretch to over nine screenfuls vertically. Accessing the data at the very bottom of the view required scrolling past all the unrelated information.

The right side of the illustration shows the amount of vertical space taken by the improved editor when viewing only the basic information of a player account. In the new editor view, long lists of items are organized under their own tabs, preventing them from needlessly bloating the length of the editor when handling other types of data. So, while a long list of player-possessed items may still require multiple screenfuls of space to display, it will not hinder accessing other data anymore.

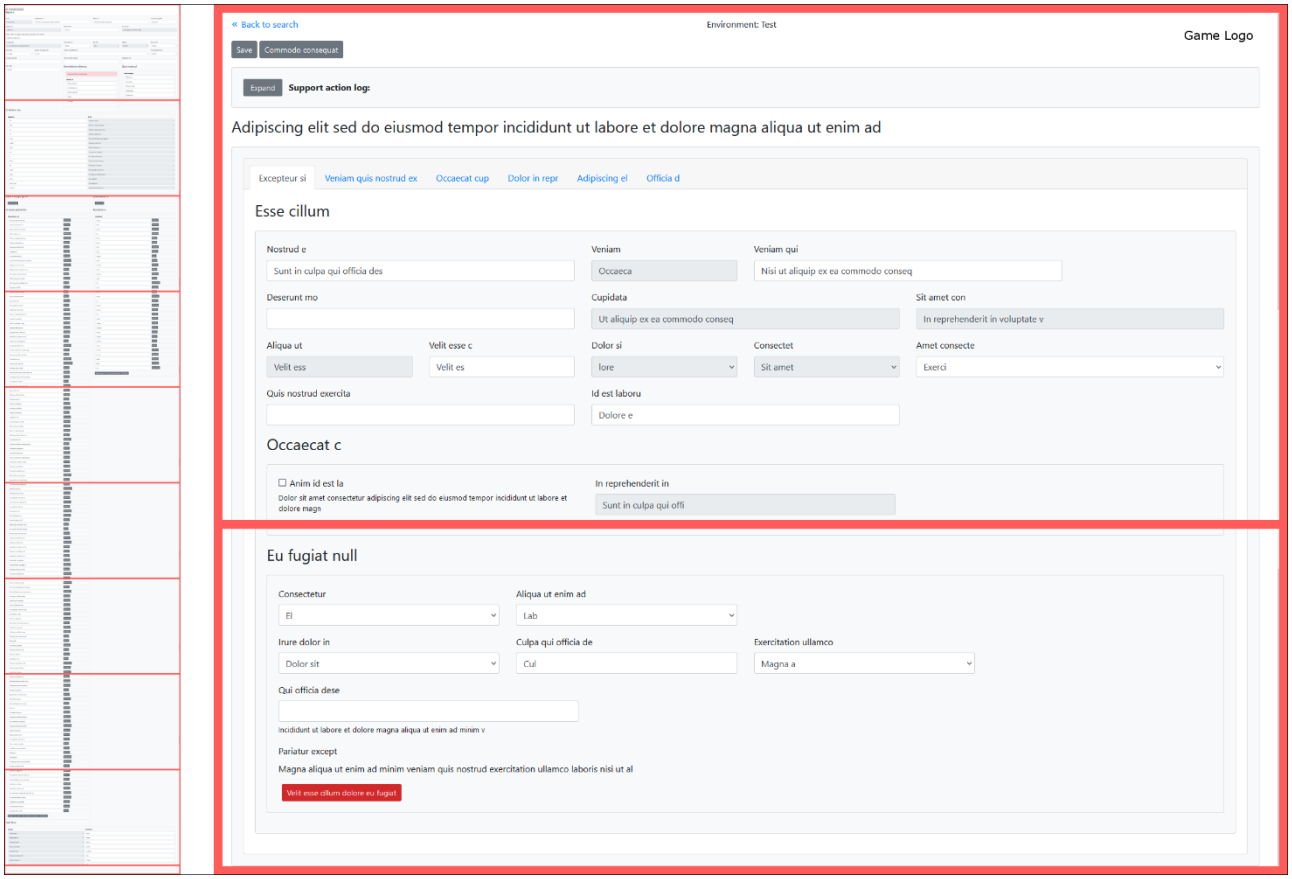

Figure 17. Comparison of the lengths of the main editor views. The red rectangles divide the editor views to screenfuls for illustration purposes.

7.4.3 Inventory-like view

A view showing different types of items that the player can have and the quantities of each of them was improved to be easier to read and more intuitive to understand. The old implementation is shown in Figure 18. This version had multiple problems that were, among other things, described in chapter 4.2. Most notably, the view expanded to the full width of the web browser window, at worst stretching the input field and item type labels to be remarkably oversized in width, consequentially also separating the value and the label from each other. Another issue was the misuse of the disabled dropdown selection control as a makeshift replacement for actual textual labels. The order of values and their labels was also unusual, with the value being presented first.

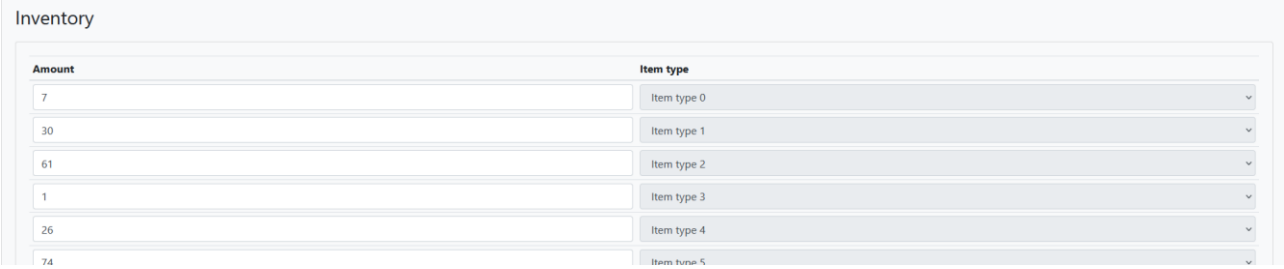

Figure 18. Inventory-like view, previous implementation

The improved version of the view can be seen in Figure 19. All the problems mentioned have been remedied. The inventory view as well as both, the labels and input fields now have a maximum width configured so they don't needlessly expand to fill the whole window. The gap between item types and input fields may still seem exaggerated in this illustration but is explained by the need to be able to show labels that are a little wider than the generic example labels.

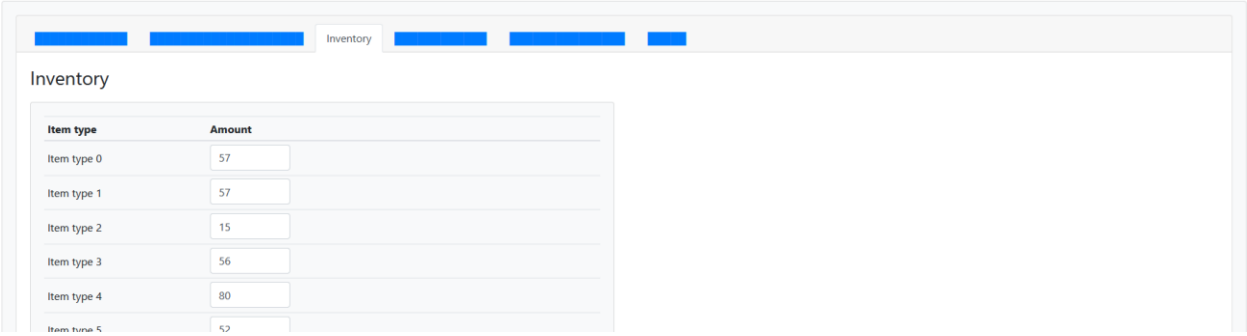

Figure 19. Inventory-like view, improved version

7.4.4 Upgrades view

A view displaying several features that only became available to the player after they had collected a certain number of collectables specific to that feature was confusing in multiple ways. Some of the problems were already discussed in chapters 4.2 and 4.3. One of the most crucial issues was that using the view required the user to have memorized knowledge about the game, information that was not available in the editor itself. This required knowledge was the "magic number" of collectables needed for unlocking each upgrade. Another problem was that the names of the upgrades differed from the names used in the game, quite severely in some cases. This was previously highlighted in Table 1. Additionally, the UI had similar problems as the inventory view

did – unnecessary stretching in width and the usage of disabled dropdowns as labels. The old version of the view with all these issues present is shown in Figure 20.

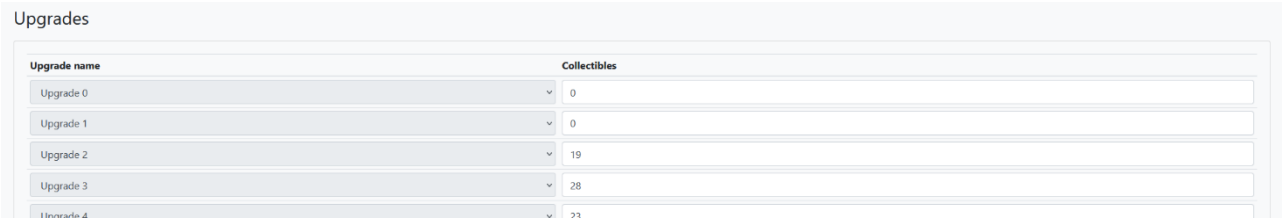

Figure 20. Old upgrades view

In the new version all problems mentioned have been solved. The view now displays the player's progress towards each upgrade, showing how many collectables they have collected out of the required amount. Additionally, a button for gifting the upgrade to the player was added to the view for each upgrade. Clicking the button automatically sets the collectable count to the maximum for that upgrade's collectables and so unlocks the upgrade. The names used for the upgrades were fixed to match the naming used in-game. The names are now also displayed as normal textual labels instead of disabled dropdowns and the view has had its width limited to maintain good readability. The improved version is shown in Figure 21.

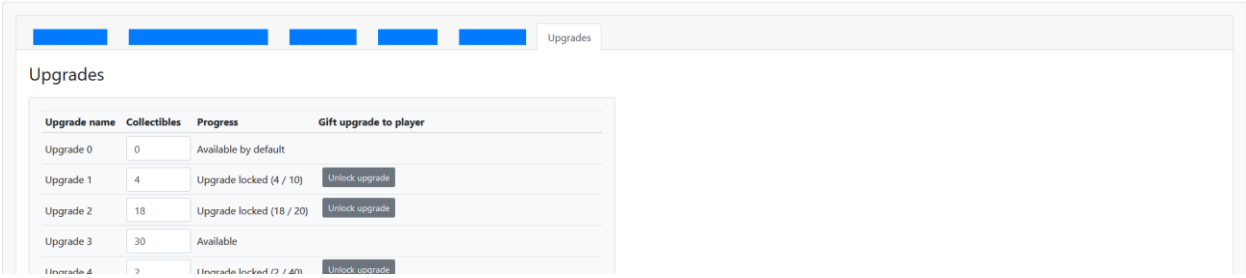

Figure 21. Upgrades view, improved

7.4.5 Enhanced workflow for enabling and disabling a feature for a player

The game had a feature that gave the player some benefits for a limited time. Being able to activate and deactivate this feature for a player was a required feature for the editor. The old workflow for toggling the feature on and off was unintuitive and confusing, as it involved editing a date

time field. Also just checking what the status of the feature was for the player was slow as the only indication was the value of the date time field. This old implementation of the UI for the feature can be seen in Figure 22. The feature and its problems are described in more detail in chapter 4.2.

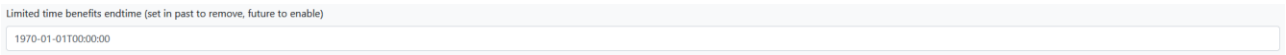

Figure 22. Limited time benefits toggle view, old

In the improved version of the UI for the feature, manual editing of the date time field was disabled, and its functionality replaced with a checkbox to both indicate and control the status of the setting for the player. The date time field is updated automatically when the user saves the changes they have made. Both UI elements were also grouped together into a container with a descriptive heading to clarify that the elements were tightly related. The new UI is shown in Figure 23.

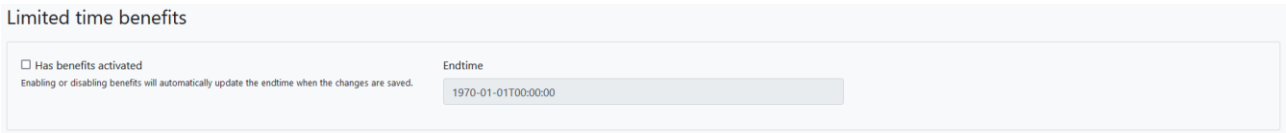

Figure 23. Limited time benefits toggle view, new

7.4.6 Grouping related features

The editor had a problem where some features that were conceptually related were separated from each other in the editor view. This cluttered the editor and made using it needlessly frustrating and error prone. This problem was discussed in more detail in chapter 4.2. Figure 24 showcases the problem, notice how the related features are scattered to every edge of the screenshot. All the censored elements are unrelated to this group of features.

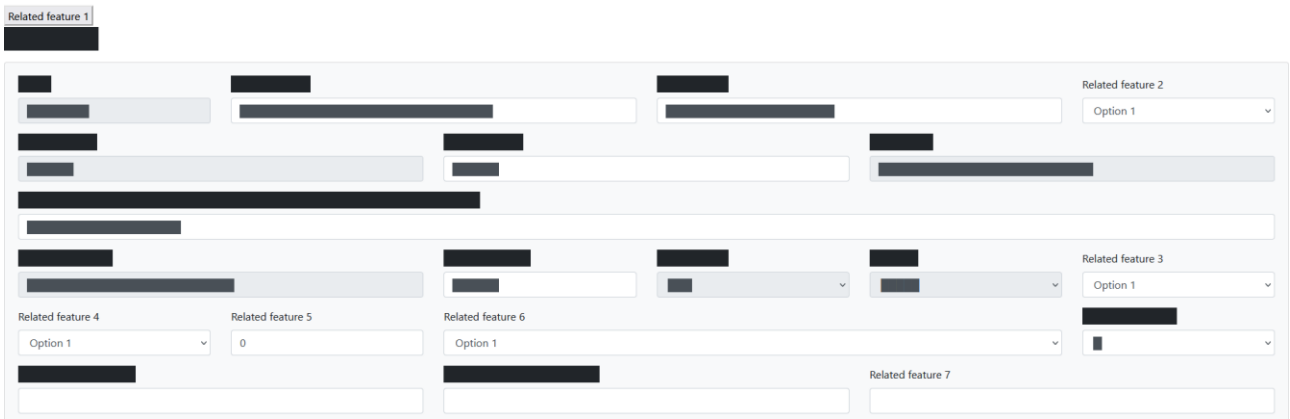

Figure 24. Illustration of poor placements of related features

The problem was remedied by similar measures as used for grouping the UI elements for the time limited feature toggle view. The related features were reorganized into a container UI element that was given a descriptive heading. The improved version can be seen in Figure 25. The censored elements seen in the figure are explanatory info texts.

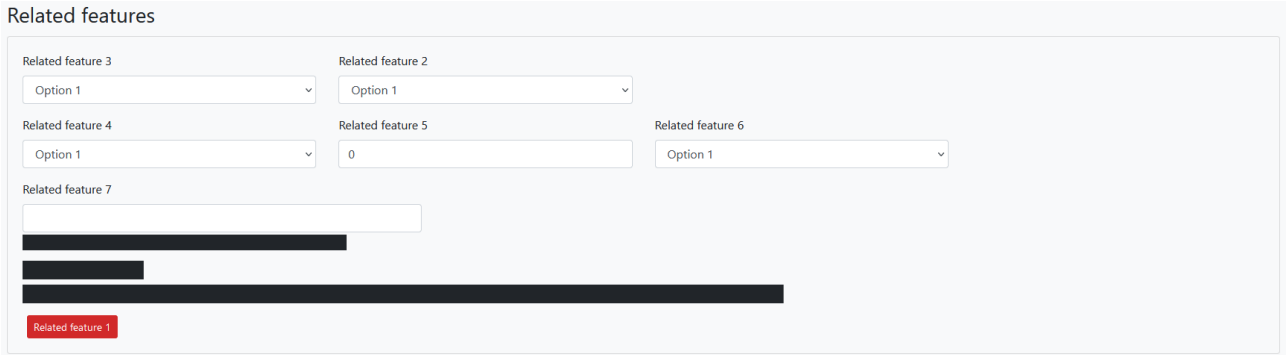

Figure 25. Related features have been grouped together under a common heading

7.5 Feasibility of JSON Editor as a foundation for a user data management tool

7.5.1 Overview

The third research question was about the suitability of JSON Editor for use as a basis for a user data management tool. Answering the question is not straightforward. On one hand JSON Editor does provide a notably quick way of setting up a basic user interface for inspecting and modifying user data that is available in JSON format. On the other hand, the library has some problematic

qualities and shortcomings, most of which concern the automatically generated layout and limitations in its adjustability.

7.5.2 Useful qualities of JSON Editor as a basis for a user interface

As said, JSON Editor is good for setting up a simple user interface for editing user data. A backend that can forward user data in JSON format and receive and convert JSON data back to a format that can be written to the user database is the only major pre-requirement for deploying a JSON Editor-based user management tool. Besides that, only a JSON schema that lists the relevant user data properties is required for specifying which properties are to be shown in the editor.

One of the strengths of JSON Editor is that it is customizable and extendable. The system has built-in support for multiple popular CSS frameworks, including Bootstrap 4 and Bootstrap 5, and allows creating custom themes, though there is no comprehensive documentation for custom theme creation. The tool can also be extended by building custom editor components.

Another useful JSON Editor feature is that it fully supports JSON Schema core and validation specifications version 3 and 4, which means that it also supports JSON Schema-based data validation (json-editor: JSON Schema Based Editor 2022). The validation rules can be set on a per-property-basis in the user schema. If validation rules are provided, JSON Editor validates the data in real time when it is being edited and notifies the user of any rule violations. The validation system can help catch human errors such as erroneous edits in time to prevent them from causing any inconveniences to the owner of the user account being handled.

7.5.3 Shortcomings of JSON Editor-generated user interfaces

The structure of a user interface that is automatically generated by JSON Editor is closely tied to the structure of the data being displayed. As a result, there is no visual hierarchy or grouping of related properties if there is no such hierarchy in the data. A “flat” JSON object results in an editor with all properties laid out to a single view. The system has no built-in mechanisms for manually specifying visual hierarchies or property groupings.

A user interface generated by JSON Editor also has a poorly responsive layout. The editor elements do resize in a responsive manner, but the results are not very convincing. In a wide viewport, most elements stretch to unreasonable widths and in a narrower one some elements can shrink so much that any data they are meant to show may become largely hidden or unreadable. There are options for adjusting the sizing of individual editor elements to some degree, but the options are implemented as JSON Editor-specific schema keywords that must be manually entered for each property and quickly bloat the size of the schema document with low-relevancy information if specified for many properties.

7.5.4 Summary

Based on the experience gained from the development project it can be said that JSON Editor is a suitable foundation for a user data management tool user interface, albeit with certain reservations. For the described use case, JSON Editor is probably best suited for simple and relatively small-scale systems where the amount of user data to be displayed is moderate or the data is hierarchical in nature so that JSON Editor also builds a hierarchical UI. If the data is “flat” and sizeable instead, JSON Editor is likely to generate an unorganized and busy-looking user interface that can be difficult to understand and unwieldy to use.

JSON Editor is also customizable and extensible to some extents, though not much can be said about the significancy of these sides of the system as they were not explored in the development project. Nevertheless, it is probably safe to assume that customizability and extensibility of the system are qualities that add value to its use as a basis for a user interface.

8 Discussion

8.1 Remaining issues

8.1.1 Player search feature limitations

The player search feature that is used for finding the player account to inspect and edit is very limited. The feature only supports searching players based on a few player-specific identifiers. The search view still includes search options that are not functional in the production environment and instead only cause the query to time out.

Solving the timeout problems with the player search would have required modifying the backend of the player editor, and for that reason was not done as a part of this development project where the choice was made to focus on the frontend of the tool.

8.1.2 Some in-game items are still only represented as “magic numbers” in the editor

Some types of items obtainable by the player in the game are still represented only as numeric values in the editor. As a result, working with them requires to user of the editor to either have memorized which value represents which item or to have a list of items and their numeric representations at hand for reference. Such a list may be unreliable because any new content added to the game needs to be manually added to the list which is easy to forget and prone to human errors.

8.1.3 Change log history has an arbitrary length limit

The Player editor change log is saved as a property of the user data in the database, and so the length of the change log history is dependent on the maximum size of the field allocated for it in the database. The limitation was acknowledged during development, and after considering the average amount of support operations performed for a single player account it was concluded that the limitation was not severe enough to warrant spending time implementing a more sophisticated change log storage system. A more refined system with unlimited change log size would only help prevent the rare cases where the current, limited change log length is not sufficient. So, instead, a simple system for freeing up space in the change log automatically and/or manually was implemented to handle the situations where the limit would be reached.

But the user experience of the change log in the case where the storage space does run out is less than ideal because it stops the user from saving their work until the issue is solved. The editor provides the option to automatically remove as many of the oldest entries in the change log as needed for having enough space for the new ones or alternatively allows the user to remove any change log entries manually. Either way, stopping and delaying the workflow of the user and forcing them to work around an implementation-dependent issue results in a poor user experience. Improving the user experience is a valid reason for reconsidering a better implementation in the future.

8.2 Ideas for further development

8.2.1 Overview

During the process of improving Player editor, numerous additional feature requests and opportunities for improving the editor were identified. Due to the time-limited nature of a thesis project not every request could be fulfilled but identifying remaining needs and problems and compiling them to a comprehensive list can still provide value for the future development of the tool.

8.2.2 Better workflow for gifting in-game items

There were multiple different types of items in the game, and it would be useful if it was possible to easily gift any of these items to a player via the editor. Most items can already be gifted, but the workflow is cumbersome with the user having to deal with item-specific textual or numeric identifiers. Some item types are not manageable via the editor at all, limiting the capabilities of the support team. A better solution for these problems would be to have an item-gifting UI with search and filter options for finding and gifting specific items. Optimally, the UI should include pictures of the items as seen in the game.

8.2.3 IAP bundle gifting

The game features purchasable item bundles, and occasionally a support team member might want to gift such a bundle to a player. Currently this is not possible via player editor, though the same result can be achieved by gifting a set amount of each item type manually but doing so is much more work than should be necessary, and some item types cannot currently be gifted at all via Player editor. A menu for selecting an IAP bundle to gift from the bundles present in the game would be a welcome addition to the editor.

8.2.4 Add more in-editor explanations and tooltips for help

A few of the more complex parts of the editor could benefit from additional explanatory help texts and tooltips that the user could utilize to better understand the tool and how it is used. For example, some of the item types in the game are quite like each other, so having additional descriptions available for them could help the user be more certain that they are dealing with the intended item type.

8.2.5 Further simplification of tabular data views

When rendering tabular data, JSON Editor automatically generates associated table row operation buttons such as “add”, “remove”, “remove last” and “remove all” for each table. These can be disabled for individual properties via property settings in the JSON schema provided to JSON Editor. The option to disable the buttons is already in use for some of the data views where the number of rows in the table is fixed and the only modifiable data is the value associated with each key.

But there are also other types of data displayed in a tabular format, some of which still have all the automatically generated buttons available. The type of data displayed in each table should be analyzed to determine which of the options are relevant to have present, if any, and the settings should then be adjusted accordingly.

8.2.6 Refactoring the change log system implementation

A lack of depth in previous experience with JavaScript resulted in a somewhat unorganized structure for the codebase of the change logging feature. Another contributing factor for some of the written code being low quality was that the feature proved to be considerably more complex to implement than initially thought. Gradual increases to the complexity of the system provoked the implementation of ad hoc solutions to unexpected problems, often increasing the complexity of the code and decreasing its readability. Some refactoring was performed at the end of the development project, but further refactoring of the code could significantly improve its maintainability.

8.3 Conclusion

The beginning of the project included gathering feedback from the users of the tool as it was. The objective was to form an initial understanding of the then-current user experience. A free-form request for thoughts, ideas and wishes with a couple of directional questions was used to gather the information. Unfortunately, not many people answered the request. In hindsight, the low response rate problem was probably a result of the choice of using a text-only message for approaching the users. A scheduled group video call with the support team would likely have yielded a better and more comprehensive understanding of the team’s experience with using the tool. Video calls were chosen as the primary communication method with the support team for the rest of the development project. A video call meeting was held with the maintainers of Player editor

with the goal of learning about the implementation of the tool. Problems in the editor as well as possible improvements and future features were also discussed.

The editor was thoroughly inspected and any issues in usability and user experience were written down as well as bugs that were encountered. As the feedback questionnaire sent to the support team yielded only a little information, the work on the editor was started based mainly on the discussions with the maintainers of the tool and findings from inspecting the editor.

At the beginning of the practical part of the development project, the choice was made to preserve the existing implementation of the user management tool and aim to improve its usability in favor of the option to build a completely new tool. This choice had the advantage of not having to rebuild something that already existed. Being able to concentrate on improving the existing solution also meant that there was a very good chance that the result would be an improvement over the starting point whereas rebuilding the tool in the limited time available could have meant having less features available in the new tool than in the previous one. This advantage was seen in the result in the way that all existing features were preserved or improved, as well as new ones added. The choice also probably factored in there being enough time for polishing the visual appearance of the editor.

On the other hand, continuing to build on what was already present also meant that any problems or limitations in the implementation of the existing tool could restrict the methods available for improving the tool. This disadvantage was encountered a few times during the project, when implementing some features proved to be more laborious than expected. The most notable example of such a case was the automatic change detection feature, where the development task turned out more complicated because of having to work around some peculiarities in the way JSON Editor handled updates in table contents.

Another problem that stemmed from the decision to retain the JSON Editor base of the application was that the system placed limitations on the choice of HTML elements used for displaying the data. Due to how JSON Editor renders the editor, some workarounds had to be used to have specific types of data be displayed in semantically correct and intuitive HTML elements.

The result of the development project was a visibly improved Player editor with a more organized layout, improved visuals and functionality, and an overall improved user experience. A major addition to the tool was an automated change logging feature that maintains a history of support team actions on a player account.

The source code of the implementation of the change log system ended up being a little unorganized. More comprehensive prior knowledge and experience of JavaScript design patterns could have simplified the process of building the feature.

In conclusion, the main goal of the development project was reached – the user experience of Player editor was significantly improved. During the project, valuable knowledge about JSON Editor was also gathered. The knowledge gained allowed evaluating the suitability of JSON Editor as a basis for a user data management tool user interface, with the conclusion being that JSON Editor is suitable for the use, albeit with a couple of caveats regarding the adjustability of the UIs it generates.

References

<dialog>: The Dialog element - HTML: HyperText Markup Language. 2023. MDN HTML reference. Accessed on 11 April. 2023. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog.

Bargas-Avila, J.A., Brenzikofer, O., Roth, S.P., Tuch, A.N., Orsini, S. & Opwis, K. 2010. Simple but Crucial User Interfaces in the World Wide Web: Introducing 20 Guidelines for Usable Web Form Design. In publication User Interfaces. Ed. Rita Mátrai. Accessed on 27 March 2022. Retrieved from https://www.intechopen.com/chapters/10814.

bootstrap: The most popular HTML, CSS, and JavaScript framework for developing responsive, mobile first projects on the web. 2022. GitHub repository. Referenced on 30 November 2022. Retrieved from https://github.com/twbs/bootstrap/tree/v4-dev.

Date.now() – JavaScript. 2023. MDN JavaScript reference. Accessed on 3 April 2023. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now.

Grant, W. 2018. 101 UX Principles: A Definitive Design Guide. Birmingham: Packt Publishing. Accessed on 12 April 2023. Retrieved from https://janet.finna.fi, ProQuest Ebook Central.

Grid system. n.d. Bootstrap 4.0 documentation. Referenced on 15 February 2023. Retrieved from https://getbootstrap.com/docs/4.0/layout/grid/.

Johnson, J. 2014. Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines. Second edition. Waltham: Morgan Kaufmann. Accessed on 31 March 2022. Retrieved from https://janet.finna.fi, ProQuest Ebook Central.

JSON Schema. 2022. JSON Schema website. Referenced on 8 November 2022. Retrieved from https://json-schema.org/.

jsondiffpatch: Diff & patch JavaScript objects. 2022. GitHub repository. Referenced on 8 November 2022. Retrieved from https://github.com/benjamine/jsondiffpatch.

json-editor Wiki: Theme. 2022. GitHub Wiki. Referenced on 13 February 2023. Retrieved from https://github.com/json-editor/json-editor/wiki#theme-srcthemejs.

json-editor: JSON Schema Based Editor. 2022. GitHub repository. Referenced on 7 November 2022. Retrieved from https://github.com/json-editor/json-editor.

Latva-Reinikka, A. 2022. Instructions for Bachelor’s Thesis – Different forms of the thesis. 2021. Seinäjoki University of Applied Sciences - Academic Library. Accessed on 22 November 2022.

Retrieved from https://seamk.libguides.com/instructionsforbachelorsthesis/goals/different-formsofthethesis.

Official-ify this fork. 2018. Issue on JSON Editor GitHub repository. Referenced on 13 February 2023. Retrieved from https://github.com/json-editor/json-editor/issues/5.

Window: prompt() method – Web APIs. 2023. MDN Web APIs reference. Accessed on 11 April 2023. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Window/prompt.