Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Catedra de Calculatoare

LUCRARE DE DIPLOMĂ

Proiectarea unui framework online pentru EVE Online

Conducător Științific:
Conf. Dr. Ing. Răzvan Rughiniș

Autor:
Andrei Picuș

București, 2013

University POLITEHNICA of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

Building an online fitting framework for EVE Online

Scientific Adviser:
Conf. Dr. Ing. Răzvan Rughiniș

Author:
Andrei Picuș

Bucharest, 2013

Abstract

The purpose of this thesis is to create an online framework that helps players of the popular scifi video game, EVE Online, with one of the most crucial aspects of the game: designing ship fittings. Building on top of a community-developed fitting engine, this project will support the simulation of numerous combinations of ships and modules. Users will be able to use the framework to create and share fits, and 3rd party developers will be able to use the provided APIs to enhance their services.

Contents

Chapter 1

Introduction

EVE Online [1] is a Massively Multiplayer Online [2] game with Role Playing Game [3] elements, developed by CCP Games [4] and published in 2003. Set in a futuristic environment, it has a player base of over 500,000 active players and a vast community of 3rd party developers. Certain tools have been developed that make the life of players easier by letting them monitor their characters [5], simulate fittings offline [6] and study the market [7].

1.1 Objective

Our main objective is to build an online framework for EVE Online that allows players to observe and analyze the interactions between entities within it. These entities consist of ships and the modules you can fit on them. Different modules affect the ship in different ways and each has a well suited purpose. Finding out the right combination of modules can be both time and resource consuming as you can only fit your ship with items that you own.

Every item in the EVE Online universe is stored in a table in the game's database. CCP provides these tables as part of their community toolkit [8], but does not provide documentation for them. Thus, these tables need to be first analyzed to extract useful information. With about half a million rows, some of these taking part in relations involving up to 5 foreign keys, and tables with in excess of 20 columns, this will not be an easy task.

Moreover, the effects of modules are specified using a language called Dogma which uses an expression tree to describe how the module works and how it affects the ship and even other modules. Like the tables, Dogma is undocumented.

After a proper research phase, we will start building an engine that parses the database and can then simulate a ship fitting (the ship and the modules fitted on it). This engine needs to be highly scalable and efficient as it will be used to power an online framework that will host thousands of simultaneous connections. Not only that, but the layers of abstraction need to support EVE Online's development cycle which states that, every 6 months, a new expansion is released, which brings new modules and updates existing ones.

On top of that engine, we will build a high-level wrapper that will be closer to the mechanics of EVE Online. It will provide methods for querying the most important attributes of a fit and act as a middle layer between Dogma and a GUI. While still providing access to the raw engine, it will augment it with higher-level logic, whilst not slowing it down in any way.

After all of this is done, an online framework is to be designed that will be feature rich, fast, scalable and sleek. It will provide cross-browser means for creating and sharing fits using a user-friendly interface capable of supporting thousands of simultaneous users 24/7. While, currently, there is no way to push data into the game, CCP provides some Javascript hooks in their ingame web browser that developers can use, along with a special data format, to allow users to save fits. The online framework needs to be compatible with the ingame browser and allow exporting fits to that special format.

Most importantly, this platform needs to provide a fast and easy update process, which will consist of getting the newest version of the database, feeding it to the engine and redeploying the online framework.

1.2 Motivation

As an EVE player you are presented with a harsh environment where your mistakes can have dire consequences. Losing your ship in combat means losing time and money. As a result, one must plan ahead before entering combat. Finding a suitable fitting for your ship can make the difference between walking away victoriously, or losing considerable amounts of resources.

Thus, EVE players need an environment where they can safely experiment with different fittings and scenarios so that they can find the optimal setups for their ships. Since you can only fit your ship with items that you own, having an environment where you can do so without spending resources may prove to be invaluable.

With a community of over half a million players, this project will make a huge impact not only among the players, but among 3rd party developers alike that can build upon it and improve their services.

1.3 Impact

As fitting ships is at the core of the game, a tool that allows easy and risk free creation of fittings is necessary. We foresee that a vast majority of EVE players will use our framework, directly or indirectly.

Currently, alliances and corporations need to maintain a wiki for all their common fleet doctrines in text only format that players need to constantly check and import in any offline tool they have available. We believe that moving the process of managing fits online will make it easier for everyone to remain updated.

Alliances can range from tens of players to even thousands, while coalitions can even consist of tens of thousands of players. All these players need quick and easy access to the repertoire of fittings, and what better way to do it than through a central service that is both easy to manage and highly secure. Fits can be made public for anyone to see, or they can be restricted to the players of the alliance or corporation the owner is part of. Thus, opposing alliances will not have access to each other's fittings. Nowadays, this is achieved by using authentication on the wikis and sharing the passwords amongst the alliance, which can be very easily exploited.

Moreover, there are certain services, like Battleclinic [9], and killboards, such as eve-kill [10] and zKillboard [11], who require ways of rendering fits, that will greatly benefit from this project. We plan to release an open API [1] that anyone can use to retrieve public fits and embed them on their websites. The API will not only provide renders of the fits, but also statistics calculated with our engine. Currently, these services only provide a visual representation of the fitting, leaving the statistics part in the hands of the players which have to paste them in text-only format from their own tools. This is not only cumbersome, but prone to errors as different tools have different methods of calculating stats.

In the end, our plan is to create a central source for everything related to EVE fittings, that anyone can use and that anyone can take advantage of in building their own projects. A highly scalable and flexible architecture is the key in achieving this, while an easy to use interface will keep users coming back.

1.4 EVE Online

EVE Online is a player-driven, persistent-world MMORPG set in a single shard science fiction universe. Players pilot customizable ships through a galaxy of over 7,500 star systems connected by means of stargates. Character advancement is done through training skills [12] in real-time.

While EVE shares many aspects with most of the current MMO games, it differs radically through its player driven market. Every item in the game is manufactured by the players themselves through the use of blueprints [13].

Another important difference is that once a player is killed in combat, his or her ship, along with the modules fitted on it, are lost. This makes EVE considerably more difficult than other games in the genre as death can have extreme consequences for players. Items must be restocked after every combat loss and, in special cases [14], a player can even lose skill points, thus having to retrain the lost skill.

EVE Online is hosted on a single massive server called Tranquility (or simply, TQ). At the time of writing, the London based server sports 4TB of RAM, 2.5 THz of CPU power and military grade equipment.

In February 2013, EVE Online reached over 500,000 subscribers [15], making it the single video game in history to have a steady growth of the player base over the course of 10 years. CCP

[1] Application Programming Interface

already has a road plan for the next decade, so this is a game that's here to stay and that will continue to evolve, along with its player base.

Chapter 2

State of the art

2.1 Technologies

2.1.1 Python

Python [16] is a multi-paradigm high-level programming language whose design philosophy emphasizes code readability. It allows programmers to write clear programs faster and easier than it would be possible in other languages such as C.

We have chosen python as our language due to a number of reasons. Firstly, some of the tools that we will use are written in python. While it would have been possible to write the rest of the project in a different language, it would have introduced communication bottlenecks and performance issues. EVE Online is also developed using python, so there's a close correlation between the game itself, the databases and our toolset.

Another selling point was the community of developers behind the language. There are a plethora of modules written for python ranging from archive managers, all the way to scientifical frameworks. Taking advantage of them will prove much easier than having to write new libraries ourselves.

Moreover, from a programming perspective, python provides functionalities like list comprehension and elements of functional programming that make it easy to write an engine that works with expression trees and lists of heterogeneous entities. There is also Django, a framework that makes writing web apps a total breeze.

In the end, python was a clear choice because it's a powerful language that suited our design philosophy and has a great community to support the developers.

However, while writing our engine in pure C might prove very difficult, the performance boost might outweigh the effort. In the future we will look on porting it to C and maybe even Javascript to allow client-only fitting tools.

2.1.2 MySQL

MySQL [17] [18] is one of the best-known relational database management systems [1] out there. It is a popular choice of database for use in web applications and a central component of the widely used LAMP [2] open source web application software stack.

MySQL has support for independent storage engines like MyISAM for read speed, InnoDB for transactions and referential integrity and MySQL Archive for storing historical data in little space. MyISAM also supports full text indexing and searching which proves highly valuable when performing keyword searches.

Early development of the project used MySQL as the database backend because of its high performance and ease of deployment. However, due to the fact that the python binding hasn't yet been ported to python 3, we were forced to switch to SQLite, which has a proper python 3 implementation.

2.1.3 SQLite

SQLite [19] is a popular database backend, favored by many due to its simplicity and performance. Unlike other backends, SQLite maintains all information in a single file. This makes migration an extremely easy process since there are no servers that have to be configured and restarted. All you have to do is copy the database file and point your frameworks to it.

It implements this simple design by locking the entire database file during writing. SQLite read operations can be multitasked, though writes can only be performed sequentially. This makes for great performance when reading the database, but writing to it can be very slow. Moreover, INSERT operations are issued in a transaction, making them even slower as the transaction overhead can be far greater than the operation itself.

Using python 3 restricted us to use SQLite, which in the long run proved favorable due to the ease of migration. While performance suffered a bit, we might stick with this backend as CCP will soon provide sqlite dumps of their database.

2.1.4 Javascript

Javascript [20] is a highly versatile interpreted language used to power today's dynamic websites. Through it, websites can interact with the user, control the browser, communicate asynchronously and alter document content. Recently, it has gained popularity in game development, as well as in desktop applications.

Javascript is a prototype based scripting language with a syntax influenced by C and key design principles borrowed from Scheme. Like Python, it is a multi-paradigm language, supporting object-oriented, imperative and functional programming styles.

[1] http://en.wikipedia.org/wiki/Relational_database_management_system
[2] http://en.wikipedia.org/wiki/LAMP_(software_bundle)

The UI of the project is written entirely in Javascript using libraries like jQuery [21] and jQuery UI [22] which enabled us to provide cross-platform features that don't require any extra plugins to work.

2.1.5 Django

One of the most well-known python packages, Django [23] is a framework for building web applications in python that comes with its own Object Relational Mapping [1] framework, along with a caching framework, internationalization support and a template engine.

The template engine allows a clear separation between code and design. The framework supports template files in which one writes the design part of the application. These templates are then fed with data coming from the application which is inserted in the templates through the use of tags.

One big advantage of having the design and code separated is that you can have separate programmers maintain each of them. Web developers can handle the application logic while web designers can produce the layout.

With its multitude of frameworks, Django allowed us to focus on code rather than deal with how to build a web platform. This made it possible to start writing code on day 1 without having to worry about implementing proper session management.

2.1.6 jQuery

jQuery [21] is a cross-browser Javascript library designed to simplify client-side scripting. It makes it easier to navigate a document, manipulate the DOM[2] tree, create animations, handle events and develop Ajax applications. It also takes on a modular approach, developers being able to easily write plugins that can take advantage of the library's power.

The UI uses jQuery to provide users with a sleek interface that is easy to navigate and understand, but rich in features to satisfy any hardcore user. All DOM manipulations are handled through the library, while certain plugins are used to implement features like persistence through cookies, sortable lists and draggable elements.

2.1.7 HTML5

HTML5 [24] is the fifth revision of the HTML standard that aims to improve the language with support for the latest multimedia while keeping it easily readable by humans and consistently understood by computers and devices. It extends and improves the markup available for documents and introduces APIs for complex web applications.

[1] https://en.wikipedia.org/wiki/Object-relational_mapping
[2] Document Object Model

We make use of HTML5 through the new data attributes [25] that allow arbitrary data to be stored for any element. This allows state information to be attached to their containers which can be retrieved easily without having to keep it separate in Javascript objects. Another useful feature of HTML5 is the ability to change the current URL without a page refresh. We do this every time a user browses fits so he/she can easily copy and share the link for it without having to use a separate form.

2.1.8 CSS3

Cascading Style Sheets [26] is a style sheet language used for describing the presentation semantics (the look and formatting) of a document written in a markup language. CSS is designed to separate content from presentation (layout, colors, fonts etc.).

The UI has builtin support for themes which can change the look and feel of the interface. These themes consist of a collection of CSS files which style different parts of the UI.

Some notable CSS3 features that are used in the UI are animations [27] and box sizing [28]. Animations allow transitions between two states, while box sizing sets how size calculations are performed in the box layout.

2.2 Research

2.2.1 Items database

As you can see in diagram 2.1, all inventory types (including ships, modules and skills) are stored in a table called inv_item. Every item has a unique identifier called typeID. Also, each item belongs to a market group as illustrated by the foreign key marketGroupID. Market groups are the ones that appear in the market browser. They can be nested, while the leaf groups have the column hasTypes set to True, which indicates that the respective market group is the holder for the items.

Besides market groups, items also have a separate inventory group, as shown by the foreign key groupID. These groups do not show in the game and they don't have a 1-to-1 correspondence with the market groups. They are more abstract groups and often, items that would fall under multiple market groups, reside in a single inventory group.

Inventory groups have associated an inventory category using the foreign key categoryID. These are top level abstract categories that contain many items. For instance, any ship, regardless of its market group, falls under the Ship category. The same goes for modules which reside in the Modules category. These categories can be used in JOINs to easily fetch all ships or all items in the market. This functionality will prove very helpful when performing item or ship searches.

Some simple attributes, like mass and volume, reside in the inventory items table. The rest of the attributes are found in the dogma attributes table, illustrated in diagram 2.2.

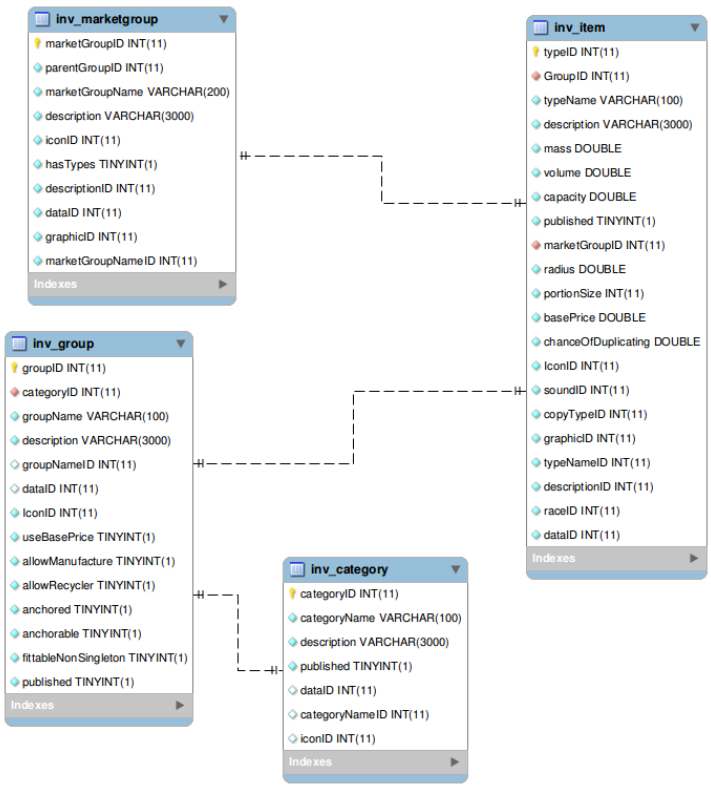

Figure 2.1: Types and groups

Each attribute has an attributeID and an attributeName used for identification. The displayName column is the name that is shown ingame. The foreign key unitID points to the dogma unit table that contains measurement units such as kilogram for mass and second for time. The stackable column specifies whether that attribute has a stacking penalty [30] that applies when more than one module that affects the same attribute are fitted on a ship. More on that in the Dogma section.

The dogma type attributes table specifies many-to-many relations between inventory items and their attributes. Each association has a value column that represents the value of that attribute for that module. There are many weird attributes present in the table for historical or unknown reasons. On the other hand, some are missing. For instance, the warpSpeedMultiplier attribute specifies how many times faster the ship's warp speed is compared to the baseWarpSpeed attribute, which is not present in the table, but rather hardcoded to the value 3.

Another oddity is that, while present, the attribute for cargo capacity is never set on any of the ships. Instead, this attribute is set in the items table. Not following this rule is the attribute for the ship's mass, which is set both in the items table and in the attributes table.

CCP's codebase has been constantly evolving so it's quite common to find such inconsistencies. The database also hosts a lot of internal testing data or data restricted to game masters and administrators. This data usually has the published column set to False, so a quick filtering by

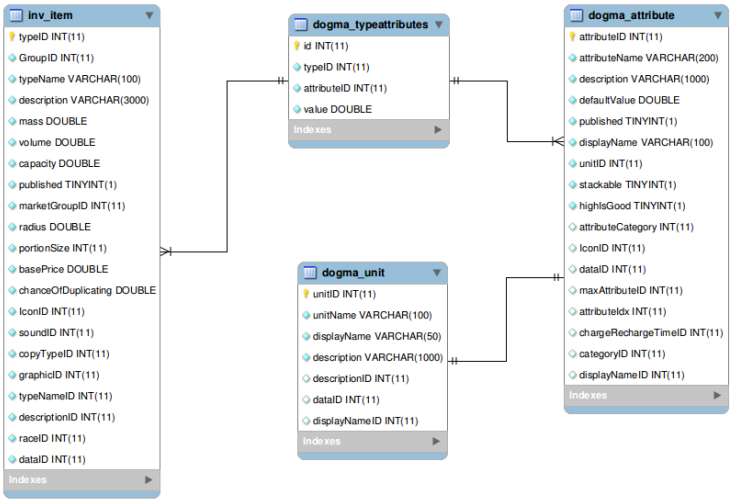

Figure 2.2: Dogma attributes

that column will only return the items that are actually present in the game.

2.2.2 Dogma

The items and attributes table are enough to learn about the modules in the game, but to see how they function we need to look into dogma effects and expressions.

Every item present in the items table has at least one dogma effect attached to it through the dogma type effects table. These effects describe what the item does and how it affects the ship, the character flying the ship and the enemy being targeted by the ship.

The main parts of a dogma effect are the preExpression and postExpression attributes, that both link to a dogma expression. The preExpression specifies what the module does when it becomes active, while the postExpression specifies what happens when the module becomes inactive. The active/inactive state can mean different things for different modules.

Passive modules become active once they are fitted to the ship and onlined. They become inactive when they are offlined or removed from the ship. In the case of active modules, the pre and post expressions refer to the start and end of the module's cycle. For instance, shield rechargers apply their bonus to the ship's shields at the start of the cycle. In this case, the preExpression will specify that, at the start of the cycle, the module will consume capacitor and boost the ship's shield. At the end of the cycle, the module will do nothing. Armor repairers apply their bonus at the end of the cycle. The preExpression in this case will include the effect for consuming capacitor, while the postExpression will apply the repairing bonus.

It is important to differentiate between these two types of behaviour as simulators can only be accurate if the effects are applied at the right time and in the right circumstances.

A dogma expression consists of a left and right argument that are actually expressions them-

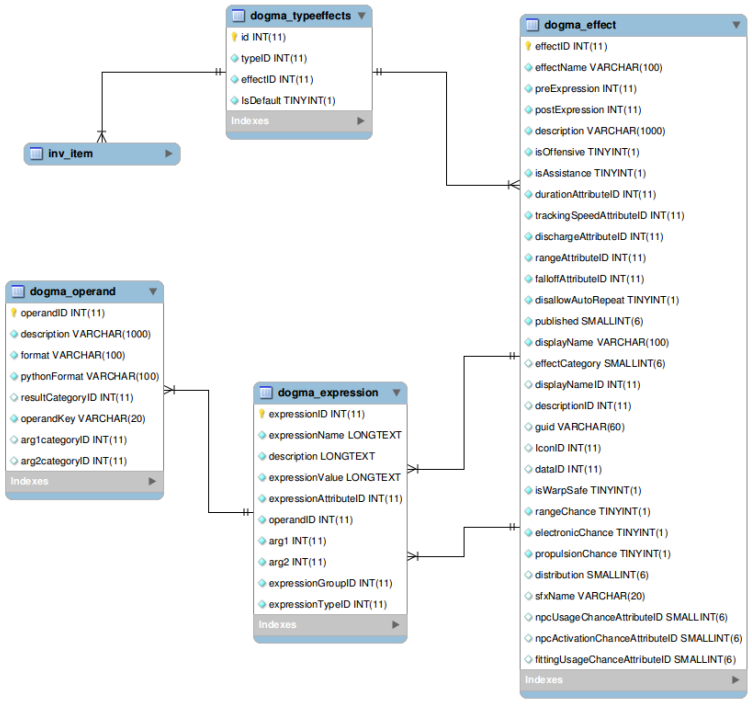

Figure 2.3: Dogma effects and expressions

selves and an operand that is applied to the two arguments. There are a number of special expressions that will actually point to the ship or character entity so that they can be used as part of much larger expressions. These expressions do not have the second argument set to anything.

When an expression needs to refer to an attribute, it sets the operandID to 22 (defAttr) and the attributeID value to the actual id of the attribute. It also ignores the arg1 and arg2 values. If the targeted attribute is stackable, then the effect receives a stacking penalty which is calculated depending on the number of modules affecting it. Thus, if only one module affects an attribute, then it will apply 100% of its bonus. If a second module is fitted that targets the same attribute, then it will only apply its bonus with an 87% efficiency. This efficiency drops exponentially with the number of modules. The following formula is used to calculate the penalty:

$$S(n) = 0.5^{\left(\frac{n-1}{2.22292081}\right)^2} \tag{2.1}$$

Now let's look at a couple of dogma expressions.

Listing 2.1: An example of a dogma expression

```
1 (SkillCheck(OnlineHasSkillPrerequisites)) AND (If(CurrentShip.cpuOutput()>=(
      CurrentShip.cpuLoad())+(CurrentSelf.cpu())), Then (If(CurrentShip.powerOutput()
      >=(CurrentShip.powerLoad())+(CurrentSelf.power())), Then (CurrentSelf->isOnline
      := Int(1));      (((CurrentShip->cpuLoad).(ModAdd)).AddItemModifier (cpu));
      (((CurrentShip->powerLoad).(ModAdd)).AddItemModifier (power)) OR UserError(
      NotEnoughPower)) OR UserError(NotEnoughCpu))
```

The above effect tells us a number of things. First, it checks whether the character has the proper skills needed to use the module. Then, it checks whether the ship has enough CPU and Power to fit the module. If these conditions are met, then the module is onlined and the CPU and Power levels are updated to reflect the newly fitted item. If the ship cannot sustain the module, then an error is thrown.

Listing 2.2: Another example of a dogma expression

```
1 ((CurrentShip->shieldCapacity).(ModAdd)).AddItemModifier (capacityBonus)
```

The above effect tells us that the capacityBonus attribute is being added to the current ship's shield capacity.

2.2.3 Capacitor

The capacitor is the ship's main power supply that acts like a rechargeable battery. As modules are activated, power is drained from the capacitor, which then starts to recharge. A capacitor has two attributes: capacity (how much power it can store) and recharge time (how much time is needed for the reactor to fill it).

Capacity

The amount of power that capacitors can store varies, although usually it varies by size of ship: the larger the ship, the larger the capacitor. The capacity can be increased directly by training the Energy Management [1] skill. Also, a huge variety of skills, and in particular those in the Engineering [2] tree, reduce the amount of power needed to activate modules, thus indirectly improving capacity. The capacity can be further increased using specialized modules, such as capacitor batteries [3], power diagnostic systems [4] and rigs [5].

Recharge time

The amount of time to fully recharge a capacitor also varies, in this case usually by the size of the capacitor: smaller capacitors tend to recharge faster. Capacitor recharge time can be reduced directly by training certain skills or fitting certain items such as cap rechargers [6] and flux coils.

[1] http://wiki.eveonline.com/en/wiki/Energy_Management
[2] http://wiki.eveonline.com/en/wiki/Item_Database:Skills:Engineering
[3] http://wiki.eveonline.com/en/wiki/Item_Database:Ship_Equipment:Engineering_Equipment:Capacitor_Batteries
[4] http://wiki.eveonline.com/en/wiki/Power_Diagnostic_System_I
[5] http://wiki.eveonline.com/en/wiki/Large_Capacitor_Control_Circuit_I
[6] http://wiki.eveonline.com/en/wiki/Cap_Recharger_I

Role in combat

If the capacitor becomes fully drained, a ship may be unable to fire its weapons, warp out from combat, or use any of its modules, and may wind up being left with only the ability to launch drones and move around. Running out of capacitor power in the middle of combat can be fatal.

There are modules that can neutralize an opponent's capacitor power, steal it and add it to your own, or allow allies to transfer it among each other. Therefore capacitor management is a notable aspect in warfare.

Role in navigation

Capacitor is also used by the ship's warp drive. The longer the distance is, the more energy is needed to initiate the warp. Sometimes a ship cannot make a very long warp in a single try, thus requiring to make more than one warp and allow the capacitor to recharge between them.

2.2.4 Recharge rate

Many EVE pilots have tried to determine the precise formula for the capacitor recharge rate — the speed at which the ship's capacitor refills itself with energy. Through observation of various experiments, it has been concluded that the recharge rate is not a linear function, but has a more complex shape, as can be seen in Figure 2.4.

By taking various ships and fully depleting their capacitor, we could observe how it recharges to full capacity. Experiments have shown that peak recharge rate is at near 20% of the capacitor's recharge time, as illustrated by Figure 2.4.

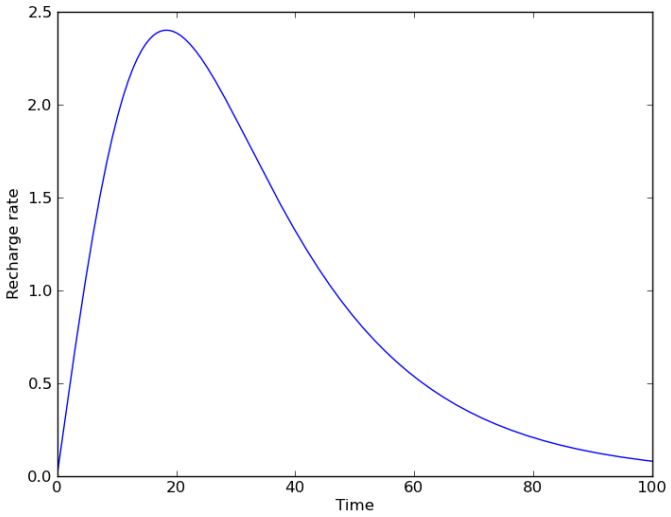

Figure 2.4: Capacitor recharge rate evolution over time

Research [29] done by various members of the community has shown that the actual formula for recharge rate is:

$$C = C_{MAX} * \left(1 - \frac{1}{\cosh(\tau * t)}\right) \tag{2.2}$$

Where:

- C_{MAX} is the maximum charge of the capacitor.
- τ is a time constant equal to $\frac{k}{T}$, where T is the recharge time.

It has been shown that k is approximately 4.8 [29].

Taking the derivative of Equation 2.2, we get the formula for instantaneous cap recharge rate:

$$\frac{dC}{dt} = C_{MAX} * \tau * \frac{\tanh(\tau * t)}{\cosh(\tau * t)} \tag{2.3}$$

We isolate t from Equation 1 and obtain:

$$t = \frac{1}{\tau} * acosh\left(\frac{1}{1 - \frac{C}{C_{MAX}}}\right) \tag{2.4}$$

$$\tanh\left(acosh\left(\frac{1}{1 - \frac{C}{C_{MAX}}}\right)\right) = \sqrt{2 * \frac{C}{C_{MAX}} - \left(\frac{C}{C_{MAX}}\right)^2} \tag{2.5}$$

Using equations 2.2, 2.4 and 2.5 we obtain:

$$R(C) = C_{MAX} * \tau * \left(1 - \frac{C}{C_{MAX}}\right) * \sqrt{2 * \frac{C}{C_{MAX}} - \left(\frac{C}{C_{MAX}}\right)^2} \tag{2.6}$$

The local maximum for Equation 2.3 occurs at $t \cong 18.3619$, which corresponds with the inflection point for Equation 2.2. This means that, at 18% of the recharge time, the recharge rate is at its peak. However, this does not mean that the capacitor is at 18% of its max charge. To find that, we substitute $t = 0.18 * T$ in Equation 2.2 and get $C = 0.284 * C_{MAX}$. So the capacitor recharges fastest at approximately 28% of its max charge. These numbers are backed up by real life observations.

2.2.5 Capacitor stability

Players are interested in the point where the capacitor is stable - at which you can sustain all your active modules and not run out of cap. That situation is commonly referred to be cap stable.

This is an important metric for any ship because it provides invaluable information in combat: how much you can last before having your capacitor fully drained, how much you can keep

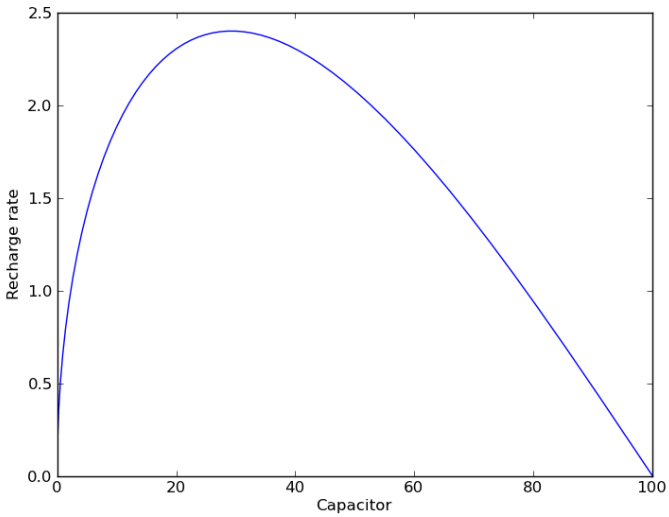

Figure 2.5: Capacitor recharge rate vs capacitor state

certain modules active or how and when you should activate certain cap hungry modules.

In order to find this point, one must look at the recharge rate function graph illustrated above. If you think of the combined drain of all active modules as a straight line on that graph, parallel to the OX axis[1], then there's 3 possible situations:

- The drain is larger than the peak recharge rate, illustrated by the line being above the curve. In this scenario the capacitor won't be able to sustain the active modules and will fully deplete. The metric in this case is the amount of time till full depletion.
- The drain is equal to the peak recharge rate, illustrated by the line being tangent to the curve. In that case, the capacitor will drop to the point determined by the peak value and then be able to sustain all the active modules. The metric in this case would be that single point.
- The drain is lower than the the peak recharge rate. In that case, there will be two cap stable points (illustrated by the line intersecting the curve in 2 points) at which the capacitor can sustain all active modules. The library will return both points, leaving it up to the next layer to decide which to display.

There are two approaches to finding these points. One would be to assume that the activation cost of an active module is drained over time, thus calculating an average per second of cap drained. All these averages could be added up and divided by the number of active modules, thus ending up with a total average of cap drained per second. This model would result in a straight line represented on the recharge graph that would yield 1 or 2 cap stable points. This approach would be very computationally light weight, but provide a rather inaccurate measurement.

[1] All modules will drain the same amount of cap, regardless of the state of the capacitor.

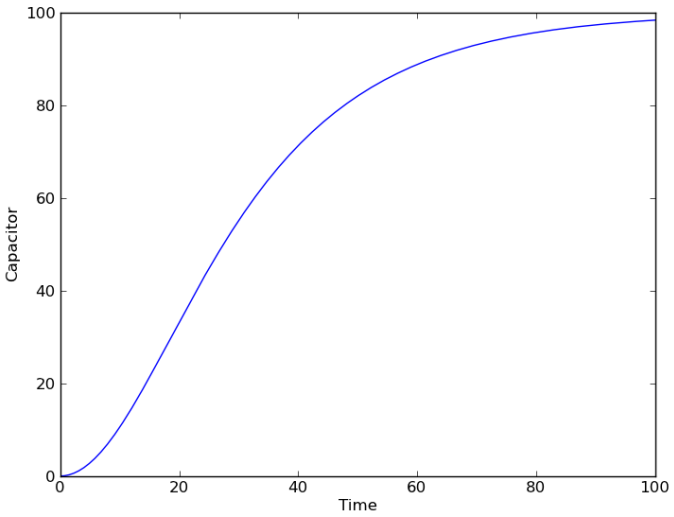

Figure 2.6: Capacitor evolution over time

The other, more CPU intensive method, would be to consider the activation costs as instantaneous bursts of cap drain (which is a correct assumption) and simulate the drain over time in small steps. You start with a full capacitor and at t = 0 all modules are activated resulting in a spike of cap drain as every module bites a portion of the capacitor. A period of time passes in which the capacitor has time to recharge until the module with the shortest cycle activates again. More such intervals pass in which the capacitor tries to sustain the active drain. At a moment in time, determined by the smallest common multiple of the module cycles, all modules will again simultaneously activate resulting in another huge spike. The interval between two such spikes shall be referred to as a drain period.

There can be more than one drain period until the capacitor reaches a cap stable point, or, in case the drain is too much, fully deplete. The simulator will run until the cap level stabilizes around a point - the first stability point. By equating the recharge rate function with the value of it in that point, we obtain the second stability point.

2.2.6 Effective Hit Points and Resistances

While there are so many different ways of sustaining damage, there is one metric that applies to the majority of them and that is the number of Effective Hit Points. This number represents the total incoming damage the ship can sustain until it explodes. It only makes sense when it is associated with an incoming damage pattern.

There are 4 types of resistances than can be applied to shield, armor and hull. These are Electromagnetic, Thermal, Kinetic and Explosive. Each of them signifies how well the ship absorbs incoming damage of that type. For instance, if the shields of a ship have 50% Kinetic resistance, that means that any enemy shooting the ship with Kinetic ammo will only do 50%

of its damage. In the database, these attributes are called resonances[1] and they're actually incoming damage multipliers. So a 0% resistance translates to a 100% resonance.

EHP is calculated by multiplying the base hit points of shields, armor and hull with $\frac{1}{resonance[type]}$. If a ship has 50% Kinetic resistance for shields, 1000 shield hitpoints and the enemy is shooting Kinetic ammo, then that ship will have 2000 effective shield hitpoints.

2.2.7 Shields

A ship's shields are the only kind of defense that regenerates over time. The recharge rate is the same as the capacitor's so the same module used to simulate capacitor drain can be used to simulate incoming damage. The key difference would be that some modules, instead of draining the shields, would add to it. This would not require any change in the simulator since these modules would just have a negative drain.

2.2.8 Damage per second

A ship's main metric for damage output is the amount of damage per second (DPS). Every type of gun hits for a certain amount of damage called alpha damage every cycle. By dividing alpha damage by the cycle time you get the amount of DPS that weapon can output. Adding them all up gives the ship's DPS. This can consist of turrets and missiles damage, drone damage and smartbomb damage.

There are two ways of calculating DPS by either taking account or not of the reload time for guns. Different guns have different reload times so it is important for a player to know the effective damage they can put out over time. While the regular metric would be enough for most fleet battles, since the target dies before the ships have to reload, most solo encounters would require to account for reloading time.

2.2.9 Align time

A ship can only enter warp if two conditions are met:

- The ship is aligned towards the destination.
- The ship has reached at least 75% of its maximum velocity.

There are two attributes which determine how quickly a ship accelerates: Mass and Inertia Modifier[2]. The latter can be reduced with both skills and modules, while mass can never be reduced, only increased. The product of Mass and the Inertia Modifier gives the ship's agility which determines how quickly it accelerates (and thus how quickly it turns).

The time to align towards the destination is always the same, regardless of the initial orientation of the ship. Thus, the time to enter warp is constant and is only influenced by mass and agility.

[1] Resonance = 1 - Resistance
[2] http://wiki.eveonline.com/en/wiki/Acceleration

The equation used to find out the align time is as follows:

$$Aligntime = \frac{-\ln(0.25) * Mass * Agility}{1,000,000} \tag{2.7}$$

The ln(0.25) element means this equation is actually finding how long it takes to reach 75% of the ship's maximum velocity.

Implementing this is rather easy as the engine only needs a method which returns a floating point number that's a function of two of the ship's attributes.

Chapter 3

Architecture

The architecture of the project involves 3 layers:

- **Eos** - A low level layer that handles effects and attributes using DOGMA. This layer should be very abstract and not care about EVE at all. This is already implemented as part of a community effort.
- **Raven** - A middle layer that augments the first one by exposing methods that return high-level information that only makes sense in the world of EVE Online.
- **Tengu** - A service layer and a GUI, that will serve visual information to users and handle their requests.

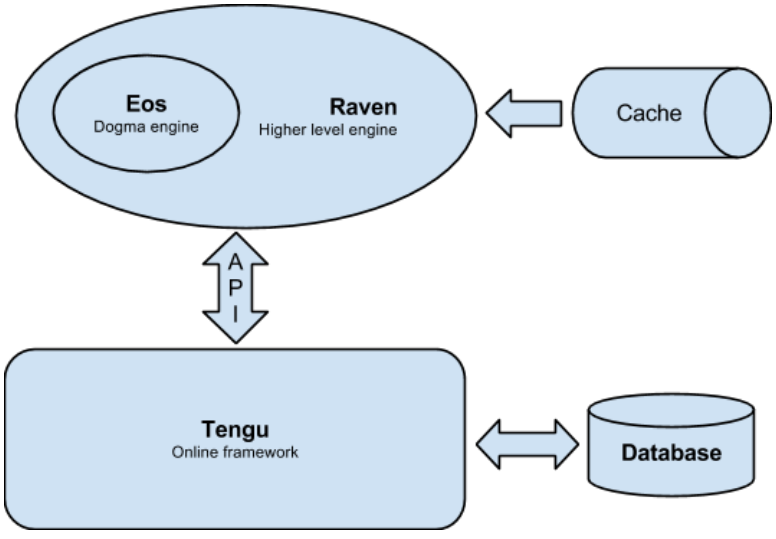

Figure 3.1: Architecture of the project

The first layer, represented by Eos, should be as decoupled from EVE as possible. Apart from knowing the structure of the database, Eos will only care about abstract entities. It will not differentiate between a weapon module and a defensive one. They will both be represented as modules that have certain effects that can be applied to certain targets.

This makes for a robust and flexible engine that can easily be adapted and maintained. If the game changes at some point, Eos should theoretically still work.

Raven, the middle layer, will extend Eos by taking all those low level attributes and outputting information that means something in the game of EVE Online: things like align time, capacitor stability, damage per second and the amount of incoming damage it can sustain. Raven will inherit Eos and will add its new methods on top of it. It will be released under an open-source license so other players can contribute to it, making it faster and more accurate.

The service layer will handle the creation and sharing of fits. User management will also be handled by this layer, along with the API that allows 3rd party developers to make use of the functionalities exposed by the middle layer. The GUI will tie all these things together and present them in a way that is both beautiful and intuitive.

Tengu will allow limitless potential for creating fits with any item in the game and sharing them with other players. Tengu will provide a sleek interface that will resemble what players have been used to ingame. It will be made available to the public, but it will be closed-source as to not expose any critical security features that might be exploited.

Chapter 4

Implementation

4.1 Eos

In order to build anything related to EVE fits, an engine is needed that can parse raw Dogma and spit out ship and item attributes. This is where Eos comes in, an engine developed by the community behind pyfa [6], a popular fitting tool.

The first thing Eos has to do is read the inventory and dogma tables through an abstraction layer that lets you define data handlers which will read these tables. After that, Eos cleans the tables of unnecessary data and performs some transformations on them (hardcoding some missing attributes, moving some rows between tables. etc) and builds a cache which it then stores on disk. Future work will involve making an abstraction layer for cache handling so that users can put in their own handlers to store the cache how they see fit.

The main functionality of Eos is exposed through the Fit class. This class encapsulates the restriction tracker which is used to validate the fit. Some items can impose certain restrictions like not having more than one module of the same kind active at the same time, while others can only be fitted on specific groups of ships. Almost all modules have skill restrictions, which specify that the character must have the appropriate skills trained to a certain level in order to use that module. All these rules are verified by the restriction tracker which will raise validation errors in case any of them is not respected.

The Fit class encapsulates the module racks which are lists of abstract holders that can contain items, along with their attributes and effects. Each attribute has a list of modules that affect it, so tracking who affects what is fairly easy and efficient. When calculating the final value of an attribute, one must take into account the stacking penalty [30], which specifies that if more than one module that affects the same attribute are fitted on a fit, then their efficiency will suffer. This means that a single module will apply 100% of its effect to an attribute, while the second one will have a slightly lower weight in the final result. When calculating the penalties for modules, they are first sorted by the value of their bonus. Modules that apply a higher bonus will do so with a higher efficiency that modules affecting the same attribute, but with a smaller

bonus. Iterating through this list, the stacking penalty is calculated using an exponential based on the module's position in the list. The 12th module and further are ignored, since, after the stacking penalty is applied, their influence on the final result is negligible. To make the calculations faster, a lookup table is used that stores the stacking penalties for up to 12 modules.

After the final values are calculated, they are cached and are only recalculated when the list of affectors is changed, by adding or removing modules. This makes for very efficient updates of the underlying fit, due to only the attributes that are affected by the update being recalculated.

4.2 Raven

Handling raw Dogma is only part of the solution. In order to build a proper fitting tool, one must go a level higher than pure Dogma.

Raven is a high-level library built on top of Eos that's designed to output more meaningful information about ships. To do this, it inherits from the Fit class declared in Eos and it adds a number of useful methods. By doing so, it provides transparency as users can still access the class as they normally would in Eos, but with the added benefit of those new methods.

The difficulty when it comes to using Eos directly is that you have to query the ship's attributes by their ids, which not only makes it difficult, but it can lead to inconsistencies as those ids could change at some point. Raven solves this by providing named properties for the most common attributes.

Most of these properties are just for convenience purposes as they only return the value of a single attribute. Some of them, however, convert and calculate these attributes to provide the final output. For instance, duration attributes are expressed in Dogma using milliseconds as a unit. Raven converts these attributes to seconds, as that is the unit used by players. Users can still get the original attribute by querying Eos directly.

Another use case that Raven simplifies is querying for the ship's shields, armor or hull. Using Eos would require many attribute lookups, but Raven returns a dictionary with all the information one could want.

Another feature, and probably the most important one, is the drain simulator used to simulate capacitor and shield usage. As mentioned in the research chapter, the ship's capacitor and shields share the same non-linear recharge function. This makes it difficult for players to estimate their cap usage and damage sustainability. Raven implements a simulator which takes into account the effects of all the modules on the ship that affect the capacitor or shields and finds the point or points where the capacitor/shields can sustain the drain. If no such point or points exist, then it calculates the time until the capacitor or shields fully deplete.

The simulator doesn't differentiate between shields or capacitor. Instead, it takes a number of arguments that define the resource (shields or capacitor) being simulated and a list of arguments defining the modules that affect that resource.

For the resource being simulated, it only needs to know its capacity and recharge time. As for the modules, it needs to know the activation cost and cycle time for each of them.

A naive solution would be to just divide the resource's capacity by its recharge time, thus obtaining a linear recharge rate. If this rate is higher than the sum of all linear drain rates of the modules, then the resource is stable, otherwise it will fully deplete.

The problem with that solution is that it assumes a linear recharge rate. Thus, if the capacitor can sustain the modules, it will be stable at 100

Another approach would be to assume that the modules have drain over time, rather than instantaneous. Coupled with the real formula for recharge rate described in the research chapter, this approach would lead to close approximations of the stability points. Not only that, but it would also be very computationally light, requiring only to solve an equation.

The most accurate solution would involve treating the module's activation cost as an instantaneous drain. This would result in a discontinuous drain function that cannot be intersected with the recharge function. Thus, Raven would have to apply both functions step by step and calculate the stability points, if they exist. However, this would require a lot of compute time since the simulator has to run until the capacitor stabilizes.

4.3 Tengu

While a fitting engine is useful in its own right, a GUI must be made to leverage its power. Our main objectives with the UI were to provide an elegant interface that is similar to the one used in EVE Online, all the while providing a user experience that attracts all the types of players.

The first step in building the UI was the layout, which had to be functional and elastic[1]. We decided on a three column setup with a footer and a header. While the middle column is used to display the main content, the first and third column are used as sidebars and contain various additional information.

There are a few ways to make this kind of layout elastic, including using percentages and em's [31]. However, when padding, margins and borders are involved, this no longer works[2]. Thus, we resorted to using Javascript to automatically adjust the layout when the window resizes. On page load, the width of the main column is calculated as the difference between the window width and the widths of the two sidebars, which are fixed. When the window resizes, so does the content, while every column is set to stretch to occupy the entire available height.

We tried to follow some semantical guidelines when writing markup so as to support various screen readers and web scrapers. Extra markup, such as containers, was only used when absolutely necessary, like for positioning elements inside the layout. We foresee some rewriting in the future to make the markup even more semantically correct.

[1] A web page with an elastic layout will adapt to the browser's window dimensions and font preferences.

[2] Specifying the dimensions of an element only affects the inner content and doesn't take into account any padding, margins or borders.

4.3.1 Market browser

After the layout has been designed, we moved on to create the market browser, which would reside in the left sidebar. The upper part would house the browser, while the lower part would accommodate the list of items in the currently selected group. These two parts are separated by a handler which can be used to resize them. At the very top of the left sidebar resides a search box, which is tab-aware, and is used to search through items and fits. The position of the handler (and, thus, the heights of both parts) is persisted through a cookie.

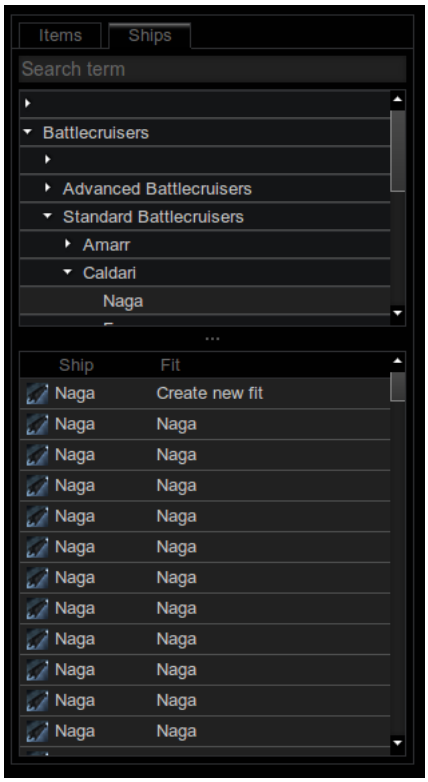

Figure 4.1: The market browser

The market browser has two tabs, one for ships and one for any item that can be fitted to a ship. Both would display a hierarchical view of the market groups available in the game, each treating differently the leaf groups. When clicking on a leaf group in the items tab, the list in the bottom part of the sidebar would load the items available in that group. However, in the ships tab, the leaf groups are the items themselves, which are actually ships. Clicking on one of them would fetch the list of fits for that ship.

Because the items are static, they are cached on the client side using a dictionary so that future requests on the same market group will not result in a new query being sent to the server. Furthermore, the entire market browser is cached on the server side using memcached [32] so as to make the page load extremely fast.

The market browser implements a few user experience features designed to make the life of the user easier. Firstly, when a user clicks on a market group, the height of which is larger than the

height of the container, the container is automatically scrolled to the top of the market group. This allows to fit as much content of that market group as possible while still retaining its name at the very top. Secondly, when a market group is collapsed, all child groups are collapsed as well. In lack of this feature, expanding that group again would present the user with a mess of previously opened groups. Moreover, the list of collapsed market groups is persisted between sessions through a cookie so as to present the user with the same, consistent browser state.

4.3.2 Fitting wheel

The fitting wheel was a lot of work. It is a custom design based on the one available in EVE and it's made completely out of vector shapes, so, if the future requires it, it can be resized without any loss of quality. The wheel is transparent to allow the picture of the ship to be seen beneath it and so as to not require any modifications when the background changes.

Figure 4.2: The fitting wheel on a transparent background.

When a ship is loaded, the number of slots are retrieved from the database and the template associates each rack a CSS class that will load the proper icons. There are 8 icon sets available for each rack in the form of rack_nrslots.png which represents the appropriate number of slots in that specific rack. These are absolutely positioned so as to align perfectly with the rest of the wheel. The initial design would involve styling each slot separately through absolute positioning and CSS3 transformations, but this would have required extra markup and could have caused compatibility issues between browsers (especially the ingame one).

Each slot was manually aligned to make them all equally spaced, while everything was aligned

using guides, resulting in a pixel perfect arrangement. The scale units are actually pipes ("|") typed on a circular path and carefully stretched and aligned to fill the required space. The larger units were individually adjusted to be equally spaced and wide like the rest.

Handling the placement of module icons on the wheel was no easy task. We set out to implement each rack as a sortable list using the jQuery UI Sortable widget. But we soon found out it doesn't support elements that are absolutely positioned, a feature we required in order to align each module with its corresponding slot. We solved this by creating a list of divs that have equal height, but each contain variably padded handles that will contain the actual module icon. The Sortable widget was set to only drag elements by their handles, which would give the appearance of individually positioned elements. Each element was styled according to its position in the list using the nth-type-of CSS3 selector [33].

However, we hit another roadblock. When an element from the list is picked up, the Sortable widget creates a placeholder element which is used to indicate the spot where the element will be dropped. This element is appended after the original element, thus, shifting the indices of all the other elements after it by 1. This was breaking the CSS selectors as they were now counting the placeholder as well. After much tinkering with the Sortable API [34], we attached callbacks to the start and stop events that are triggered when sorting starts and, respectively, stops, that take the element being dragged and move it to the end of the list, leaving the placeholder in its original place. Doing so won't break any selectors as the elements are still in their original slots. When the sorting stops, the original element is placed instead of the placeholder, which is removed by the widget.

This is an innovative approach to this problem and we plan to release a separate, open-source plugin that will create a circular sortable list using the Sortable widget and the tricks outlined above. We hope that it will help many other people in need of such a plugin.

4.4 Stats widgets

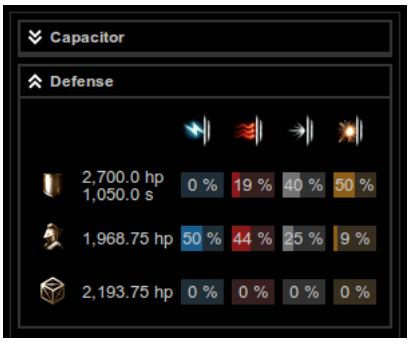

Figure 4.3: Example of a stats widget

After the market browser and fitting wheel were working properly, we moved on to style the widgets responsible with displaying the various statistics. These were made to resemble as

closely as possible the ones present in the game, so as to provide a pleasant and familiar experience to the users.

The widgets are individually collapsable and their state is persisted between sessions through cookies. We went a step further and made their state synchronized between all open tabs. That means that every time a user changes the order of the widgets or collapses or expands one of them, every instance of them is synchronized. This provides a sense of consistency and a level of customization rarely seen in web apps.

4.5 Tabs

In order to support viewing multiple fits at the same time, we had to implement support for tabs. A proper framework was written which supports sortable tabs that can be linked with multiple content panes that will be shown when the linked tab gains focus. These panes can be linked through ids or even directly by object, for scenarios where ids can't be assigned yet (e.g. when content is being loaded).

Figure 4.4: Multiple opened fit tabs

When a tab is closed, the framework automatically places the focus on the rightmost tab, if one exists. Closing tabs also triggers events that can be caught by other scripts which can register callbacks to perform various custom actions. One particular example is when closing the last fit tab, in which case a welcome tab is displayed that instructs the user to use the market browser to create a new fit or edit an existing one. When a new tab is opened, the welcome tab is hidden, only to be revealed again when the last one is closed.

4.6 Service layer

As an integrated part of Tengu, the service layer handles storing and accessing fits from the database, as well as user management and authentication.

Every user has an associated profile, which, among various additional information that are not present in the standard Django user model, stores an API key which is used to retrieve information about the ingame character from the Tranquility server. This information includes the character name, which is displayed next to a fitting so that EVE players can easily recognize the author. Moreover, the character's corporation and alliance names are fetched to allow saving fits that will be made visible to the entire corporation or alliance.

Permission checking is performed before fetching the fits so as to not let users view private fits, or fits belonging to a different corporation or alliance. Because Django's filter() method [35] only supports simple queries with "ANDed" arguments, we use Q objects [36] to perform more

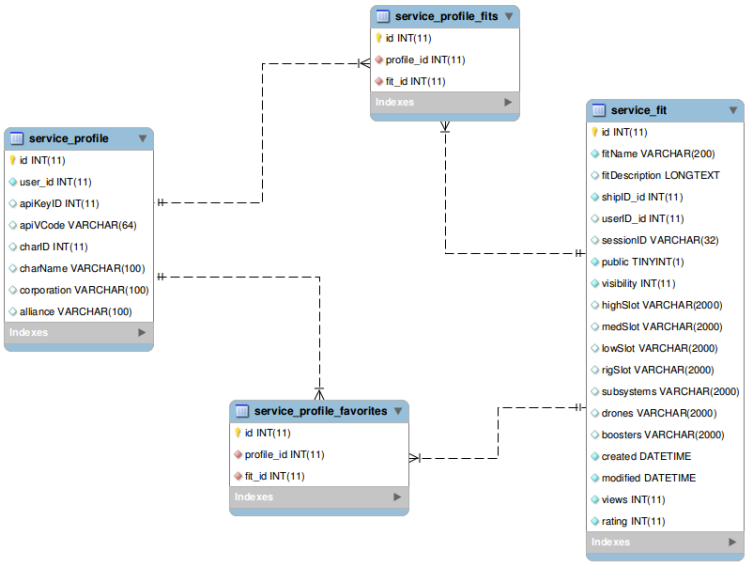

Figure 4.5: Service tables

complex searches. Q objects can be combined using the & and | operators, resulting in a new Q object.

Listing 4.1: An example of using Q objects

```
1 if request.user.pk:
2   return Q(userID = request.user.pk) | Q(public=True)
3
4 return Q(sessionID = request.session.session_key) | Q(public=True)
```

Requesting the corporation and alliance name from the EVE API servers every time a permission check is performed would make queries take a very long time to complete. To solve this issue, we only query the API once per day, for all the users, and store the needed information in our database. That way, a search query would need not interrogate the API servers, as the information would already be available offline. A cron job is setup to handle fetching and updating the information for all the users.

The service layer also exposes an API that 3rd party developers can use to interact with Tengu. The API provides calls for retrieving fittings in various formats, which can include JSON, HTML and text-only.

Chapter 5

Testing and validation

5.1 Unittets

An exhaustive test suite has been written for Eos and Raven that loads various fittings and tests that the engine is correctly calculating restrictions and attributes. This suite is ever growing so these engines will become more accurate as time passes. The results are taken from observations made ingame and are then compared to the output of Eos and Raven.

5.2 Database performance

Event	MySQL	SQLite
Page load	1.5s	1.3s
Get fits (Naga)	35ms	25ms
Get items (Civilian modules)	40ms	40ms
Search items (%drone%)	160ms	440ms
Search fits (%naga%)	50ms	180ms
Create fit (Naga)	30ms	180ms

As we can see, sqlite3 is faster than MySQL at fetching data. However, it performs horribly when doing LIKE queries with preceding wildcards. Also, sqlite3 performance degrades when Django is set to save the session on every request, thus negatively impacting every AJAX request.

5.3 Tengu performance

To increase the loading times for Tengu, we used django-compressor [37] which gathers all script files and stylesheets, concatenates them into a single file and minifies it. Thus, not only do we save on bandwidth, but on number of requests as well.

Metric	Original	Compressed
Load time	1.55s	1.45s
Bandwidth	1.1MB	547KB
Number of requests	45	27
Javascript files size	334KB	325KB
CSS files size	15.5KB	13.5KB

As you can see, compressing Tengu improved the bandwidth that was consumed on every page load and the number of requests being issued by the browser to fetch all the necessary files. The load time is limited by the image server provided by CCP, from which the browser has to download the items and ship icons.

Chapter 6

Conclusions

We set out to design a complete online framework for building fits for EVE Online. We achieved much more than that.

Eos and Raven are two powerful and accurate engines that provide simple, yet powerful APIs that anyone can use to build their own fitting simulators and not be tied to a particular GUI. Also, the documentation we wrote for these engines can help other developers write engines in other languages, or to suit their own needs.

Tengu provides a sleek interface that allows users to create, view and modify fittings using any available item in the game, while providing an unparalleled user experience. Not only that, but, due to its many abstraction layers, it's very easily to update when a new EVE expansion is released.

We foresee that many 3rd party services will use the Tengu API to fetch and display fittings using our fitting wheel template. Our future plans include writing a little Javascript library that wraps our API and allows developers to easily embed fittings into their websites.

6.1 Future work

Detailed view

While the wheel view for fits should be more than enough for anyone trying to lookup a certain fit, there are certain details that cannot be shown there. Things like optimal range and falloff for guns, tracking speed for turrets and drone range should be presented using a more detailed, list based view. This new view should provide theorycrafters with the information they need to concoct the perfect fits. The wheel view will be the default one, users being able to switch to the list view by a small icon in the top right corner.

Internationalization

EVE Online hosts players from all corners of the globe. As such, Tengu must be properly localized. Django provides an i18n[1] framework that gives you the ability to specify translation strings and provides methods to get the proper translation according to the user's locale.

Initially, Tengu will be launched in English. After that, we will seek out help to translate it in Russian, German, French, Japanese and Chinese, the most popular languages among EVE players.

Themes

We plan to create more visual themes that will suit any player looking to spruce up his or her fitting environment. Currently, a light theme is in the works to complement the default dark one.

CREST API

The developers of EVE Online announced a new API called CREST that will provide means of writing back to the Tranquility cluster. Using this new API, Tengu would be able to directly interact with the fittings stored on the TQ server on behalf of the user.

Notifications

Future work could involve adding notifications for when a player from your alliance or corporation adds a new fitting or edits an existing one. This would help players remain updated with the fleet doctrines and never end up in the situation where they would field the wrong setup.

Importing and exporting fits

As there are multiple offline solutions for fitting ships out there, we plan to support importing fits in various formats, as well as exporting them.

Creating fits through the API

Killboards are centered around fittings and, as such, we plan to provide them with an API that will allow them to use our fitting engine and HTML renderer to create fits on the fly. These fits will be stored under a user created specifically for that respective killboard and will be set to private, so only the killboard can fetch them.

[1] Internationalization

Porting Raven to Javascript

A Javascript implementation of Raven, along with Eos, would allow client-side fitting simulations, thus offloading the workload from the server to the user's browser.

Bibliography

[1] *EVE Online*. http://www.eveonline.com/, 2013.

[2] *Massively Multiplayer Online*. http://en.wikipedia.org/wiki/Massively_multiplayer_online_game, 2013.

[3] *Role Playing Game*. http://en.wikipedia.org/wiki/Role_playing_game, 2013.

[4] *CCP Games*. http://www.ccpgames.com/en/home, 2013.

[5] *EVEMon*. http://evemon.battleclinic.com/, 2013.

[6] *Python Fitting Assistant*. https://github.com/DarkFenX/Pyfa, 2013.

[7] *EVE Central*. http://eve-central.com/, 2013.

[8] *EVE Online fansite toolkit*. http://community.eveonline.com/community/fansites/toolkit/, 2013.

[9] *Battleclinic*. http://battleclinic.com/, 2013.

[10] *eve-kill*. http://eve-kill.net/, 2013.

[11] *zKillboard*. http://zkillboard.com/, 2013.

[12] *EVE Online skills*. http://wiki.eveonline.com/en/wiki/Skills_guide, 2013.

[13] *EVE Online Blueprints*. http://wiki.eveonline.com/en/wiki/Blueprint, 2013.

[14] *EVE Online Pod Death*. http://wiki.eveonline.com/en/wiki/Pod_Death, 2013.

[15] *Eve Online passes 500,000 subscribers*. http://www.edge-online.com/news/eve-online-passes-500000-subscribers/, 2013.

[16] Mark Lutz. *Programming Python*. O'Reilly Media, 2011.

[17] Paul DuBois. *MySQL*. Addison-Wesley Professional, 2008.

[18] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High Performance MySQL: Optimization, Backups, and Replication*. O'Reilly Media, 2012.

[19] Jay A. Kreibich. *Using SQLite*. O'Reilly Media, 2010.

[20] David Flanagan. *JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)*. O'Reilly Media, 2011.

[21] Jonathan Chaffer and Karl Swedberg. *Learning jQuery*. Packt Publishing, 2011.

[22] *jQuery UI*. http://jqueryui.com/, 2013.

[23] Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*. Apress, 2009.

[24] *HTML5*. http://www.w3schools.com/html/html5_intro.asp, 2013.

[25] *HTML 5 data- Attributes*. http://ejohn.org/blog/html-5-data-attributes/, 2013.

[26] *Cascading Style Sheets*. http://www.w3schools.com/css/, 2013.

[27] *CSS3 Animations*. http://www.w3schools.com/css3/css3_animations.asp, 2013.

[28] *CSS3 box-sizing Property*. http://www.w3schools.com/cssref/css3_pr_box-sizing.asp, 2013.

[29] *Thoughts and research on the capacitor*. http://oldforums.eveonline.com/?a=topic&threadID=116993, 2013.

[30] *Stacking penalties*. http://wiki.eve-id.net/Stacking, 2013.

[31] *The amazing em unit and other best practices*. http://www.w3.org/WAI/GL/css2em.htm, 2013.

[32] *Memcached*. http://memcached.org/, 2013.

[33] *CSS3 :nth-of-type() Selector*. http://www.w3schools.com/cssref/sel_nth-of-type.asp, 2013.

[34] *Sortable Widget API*. http://api.jqueryui.com/sortable/, 2013.

[35] *Retrieving specific objects with filters*. https://docs.djangoproject.com/en/dev/topics/db/queries/#retrieving-specific-objects-with-filters, 2013.

[36] *Complex lookups with Q objects*. https://docs.djangoproject.com/en/dev/topics/db/queries/#complex-lookups-with-q-objects, 2013.

[37] *Django Compressor*. http://django_compressor.readthedocs.org/en/master/, 2013.