jamk.fi

Oppimateriaalien pelillistämis-järjestelmän jatkokehitys ja analysointi

Ronnie Nygren

Opinnäytetyö
Huhtikuu 2018
Tekniikan ja liikenteen ala
Insinööri (AMK), mediatekniikan koulutusohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Nygren, Ronnie	Opinnäytetyö, AMK	23.04.2018
	Sivumäärä 85	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: X

Työn nimi
Oppimateriaalien pelillistämisjärjestelmän jatkokehitys ja analysointi

Tutkinto-ohjelma
Mediatekniikan koulutusohjelma

Työn ohjaaja(t)
Pasi Manninen

Toimeksiantaja(t)
Valteri-koulu Onerva

Tiivistelmä

Työn toimeksiantaja oli valtion erityisoppilaitos Valteri-koulu Onerva Jyväskylästä. Heidän toimeksiantona oli aiemmin toteutetun järjestelmän jatkokehitys ja tekninen dokumentointi.

Kyseinen järjestelmä on selaimelle tehty pelien tekoalustaja, jossa on myös pelien pelaamispuoli. Alkuperäinen järjestelmä oli saatu prototyypin tasolle, mutta siitä puuttui tärkeitä ominaisuuksia, kuten moduulityökalu, voittoruutueditori ja versiointi. Myös alkuperäinen mediakirjasto tuli tehdä pitkälti uudestaan. Jatkokehityksessä lähdettiin tekemään hallintapuolelle puuttuvia ominaisuuksia, parantamaan vanhoja ja tämän lisäksi tekemään dokumentaatiota jatkokehitykselle.

Työtä tehtiin sovitun kehityslistan pohjalta. Teknistä dokumentaatiota kirjoitettiin samalla kun tietty osa-alue tuli työn alle. Dokumentaation myötä tehdyn analysoinnin pohjalta järjestelmästä ymmärrettiin tiettyjä omituisuuksia paremmin ja tämän takia järjestelmää parannettiin, sekä teknistä velkaa korjattiin.

Tuloksena järjestelmään saatiin pitkälti kaikki halutut lisäominaisuudet ja parannukset. Järjestelmän hallintapuoli saatiin beta-vaiheeseen. Teknistä dokumentaatiota kirjoitettiin yli 40 sivua ja se pitää sisällään suurimman osan järjestelmästä, mukaan lukien ohjeet pelimoduulien kehittämiselle. Valteri voi jatkaa järjestelmän kanssa seuraavaan vaiheeseen eli pelipuolen kehittämiseen.

Johtopäätöksenä voidaan todeta, että on mahdollista rakentaa selaimen päälle toimiva ekosysteemi missä oppimateriaaleja voi pelillistää ja pelata. Projekti osoitti myös, että tämän kokoinen projekti vaatisi taustalle paljon enemmän senioritason osaamista.

Avainsanat (asiasanat)
Pelillistäminen, Reactin käyttö SPA-applikaation ulkopuolella, relaatiotietokanta, modulaarisuus, tekninen dokumentaatio, pelieditori

Muut tiedot

jamk.fi

Description

Author(s)	Type of publication	Date
Nygren, Ronnie	Bachelor's thesis	23.04.2018
		Language of publication: Finnish
	Number of pages: 85	Permission for web publication: X

Title of publication

Developing and analyzing system for learning material gamification.

Degree programme

Media Engineering

Supervisor(s)

Manninen, Pasi

Assigned by

Valteri school Onerva

Abstract

This project was assigned by Valteri school Onerva, a state funded institution for students needing special care. The commission was to further develop and improve the system created earlier. This also included writing a technical documentation.

The system is a platform on which a teacher can easily create games, and students can then play those games inside the same system. The original project managed to get the system to prototype level; however, it was still missing features such as victory screen editor, module editor and version control. The media library also needed rework. The goal of this project was to add missing key features, improve existing ones and write the technical documentation for later development.

The work was done based on the agreed task list. The documentation was written while working on a certain section. While writing documentation and analysing the system, certain problematic and badly written parts were discovered, which led to refactoring and other improvements.

As a result, all tasks on final backlog were completed. The admin side is now mostly in beta stage. Over 40 pages of technical documentation were written. This covers most of the existing system and includes the instructions on how to develop the game modules for the system. Valteri can now proceed to the next development cycle, finishing the playing side.

As a conclusion, it has been clearly demonstrated that it is very possible to build an ecosystem where the learning materials can be gamified and then played. However, the project has also shown that a task this big would require a great deal more senior level knowledge in a team.

Keywords/tags (subjects)

Gamification, Using React without SPA, Relational database, modularity, technical documentation, game editor

Miscellaneous

Sisältö

Kuviot

Taulukot

Käsitteet

AJAX

Asynchronous JavaScript And XML. Teknologia, jolla JavaScript-kielessä voidaan hakea dataa ulkoisesta resurssista taustalla ilman, että koodin suoritus pysähtyy odottamaan tuloksia. Alun perin suunniteltu toimimaan XML-muotoisen datan kanssa, sittemmin tukea laajennettu myös muihin formaatteihin.

Base64

Formaatti, jonka avulla minkä tahansa datan binääripresentaatio voidaan esittää ASCII-muotoisena merkkijonona.

Beta

Järjestelmän valmiusaste, jossa suurin osa tai kaikki halutut ominaisuudet on tehty ja kehitys keskittyy virheiden ja ongelmien korjaamiseen.

Boilerplate

Koodi, joka toistuu monessa paikassa lähes samanlaisena tai hyvin pienillä muokkauksilla.

Canvas

WebGL:n edeltäjä, joka tukee vain 2D-grafiikka.

Controller

MVC-ideologiassa järjestelmän osuus, joka on vastuussa bisneslogiikasta. Hyödyntää malli (model) ja antaa näkymällä (view) tarvittavat tiedot näkymän piirtämiseen.

Cron

Unix-pohjaisissa käyttöjärjestelmissä työkalu, jonka avulla pystyy tekemään ajastettuja koodin tai komennon suorituksia.

CSS

Cascading Style Sheet. Standardisoitu kieli tyylisäännösten kirjoittamiselle verkkosivulla. Selaimet lukevat CSS-säännöt ja tekevät HTML-elementeistä tietyn näköistä CSS-sääntöjen pohjalta.

CSV

Comma separeted values. Tiedon tallentamismuoto, jossa eri merkkijonon muotoiset tiedot eritellään pilkulla ja tietorivit tai -kokonaisuudet eri riveinä.

Debuggaus

Prosessi, jossa järjestelmässä olevan virheen syitä etsitään ja yritetään löytää ratkaisua.

DOM

Document Object Model. Tarkoittaa käytännössä HTML-kielen sisäistä rakennetta, joka muodostaa puurakenteen. DOM toimii ohjelmointirajapintana HTML-rakenteen muutoksille ja lukemiselle.

HTML

Hyper Text Markup Language. Standardisoitu merkkauskieli, jonka avulla voidaan määrittää verkkosivun semanttinen rakenne. Selaimet ymmärtävät HTML:n kielen tageja ja ymmärtävät niiden tarkoittavan tietynlaista sisältöä.

HTTP

Hyper Text Transfer Protocol. Protokolla linkkejä sisältävän tekstimuotoisen tiedon eli hypertekstin siirtämiseen selaimen ja internet-resurssien välillä. Internet-resurssi on käytännössä mikä tahansa pääte, johon pääsee kiinni muualta internetistä.

JavaScript

Selaimessa käyttäjän koneella pyörivä skriptikieli. Voi tarkoittaa myös palvelinpuolen kieltä, mutta ei tässä työssä.

JSON

JavaScript Object Notation. Standardisoitu tietorakenne, joka noudattelee JavaScript-kielessä oleva objektin muotoa. Viittaa myös tiedostotyyppiin, joka pitää sisällään JSON-muotoista sisältöä.

Konttiteknologia

Tekniikka, jossa sovellus jaetaan mahdollisimman pieniin itsenäisiin järjestelmiin, joita sitten ajetaan niin kutsuissa konteissa. Kontti on käytännössä vähän kuin yhden tiedoston mittainen kuvaus virtuaalisesta koneesta. Kontit yhdessä muodostavat kokonaisen järjestelmän.

LUEM

Projektissa tehtävän järjestelmän nimi. Tulee sanoista Ludus Ex Machina eli pelejä koneesta.

Model (-tiedosto), Malli (kun puhutaan tietokannasta)

ORM-mallissa luokka, joka määrittää, miltä tietokannan taulu näyttää luokkana tai objektina.

Moduuli & pelimoduuli

Järjestelmässä oleva yksittäinen luokka, joka määrittää yhden kokonaisen pelityypin toiminnot ja toimintalogiikan. Pelit on aina rakennettu jonkin moduulin pohjalta.

MVC

Model-view-controller. Ohjelman rakenteeseen liittyvä filosofia, jonka mukaan datan rakenne, bisneslogiikka ja visuaalinen logiikka eritellään toisistaan omiin kokonaisuuksiin ja tiedostoihin.

MySQL

Yksi tunnetuimmista SQL-standardiin pohjautuvista tietokantaratkaisuista.

NoSQL

Not only SQL. Ryhmä erinäisiä tietokantateknologioita, jotka eivät käytä relaatiotietokantaa tai SQL-kieltä tai käyttävät sitä vain osittain.

Ohjelmointikehys

Isohko ohjelmointikirjasto, jonka tarkoitus on antaa kehittäjälle valmis paketti työkaluja, logiikkaa, ideoita ja rakennetta yleisimpiin järjestelmissä oleviin tehtäviin.

ORM

Object relational mapping. Tekniikka, jossa tietokannan taululle luodaan sitä esittävä luokka. Muutokset käsiteltävään ORM-luokkaan siirtyvät tietokantaan ORM-järjestelmän avulla.

Phaser

JavaScript-pelimoottori ja -kirjasto, jonka avulla voi helpommin luoda selaimessa toimivia pelejä.

PHP

Palvelinpuolen ohjelmointikieli.

React

Facebookin kehittämän JavaScript-kirjasto näkyminen tekemiselle.

REST, GET & POST

REpresentational State Tranfer. HTTP-protokollaan perustuvaa malli ohjelmointirajapintojen tekemiseen. Get ja Post ovat RESTin eri tapoja lähettää pyyntöjä kohdejärjestelmälle. GET-pyynnöt lähettävät pyynnön tiedot suoraan URL-osoitteen perässä ja sillä ainoastaan saadaan tietoa ulospäin. POST-pyynnöt lähettävät tiedot pyynnön header-osassa (otsikko, alkuosa) ja sen avulla on myös sallittua muokata olemassa olevia tietoja.

SQL

Relaatiotietokannoille suunniteltu kieli operaatioiden suorittamiselle.

Symfony

PHP:lla toimiva ohjelmistokehys web-sovellusten tekemiseen.

Transaktio

Yhtenä toimenpiteenä suoritettu hakujen ja tallennusten sarja.

WebGL

JavaScriptissä oleva ohjelmointirajapinta 2D- ja 3D grafiikan piirtämiselle selaimeen canvas-tekniikkaa käyttäen.

1 Johdanto

1.1 Toimeksiantaja

Toimeksiantajana oli oppimis- ja ohjauskeskus Valteri ja tarkennettuna sen Jyväskylän toimipiste, Valteri-koulu Onerva. Valteri on opetushallituksen alaisuudessa oleva instituutti, joka keskittyy erityistukea tarvitsevien lasten ja nuorten koulunkäyntiin. Valteri ei tarjoa ainoastaan koulutusta, vaan kokonaisvaltaisesti tukea eritystarpeita vaativan lapsen tai nuoren oppipolkuun. Tämä tarkoittaa muun muassa erikseen räätälöityjä oppimistapoja, asuintiloja oppilaille, oppimateriaalisuunnittelua, -toteutusta ja -myyntiä ja virkistystiloja. Valteri tarjoaa palveluita näkemiseen, kuulemiseen, kieleen, vuorovaikutukseen, motoriikkaan ja liikkumiseen liittyviin tarpeisiin.

Valteri-koulu Onervan uusi koulu on suhteellisen uusi rakennus, joka korvasi aikaisemmat pahasti homeessa olleet tilat. Uudessa rakennuksessa on kattavat tilat, mukaan lukien uimahalli, näyttämö, äänieristetty soittotila, erilaisia tuotantotiloja oppimateriaalien tekoon, lukuisia neuvottelutiloja, laajat asuintilat ja kattavat ulkotilat. Onerva järjestää majoituksen lisäksi myös kuljetuspalveluita oppilailleen. Onerva on ollut vahvasti mukana Jyväskylän ammattikorkeakoulun projekteissa ja sinne on tehty viime vuosina muutamia opinnäytetöitä.

1.2 Taustaa

Nykyajan koulutusjärjestelmässä on hiljalleen herätty niin kutsutun pelillistämisen hyötyihin. Pelillistämisellä tarkoitetaan peleistä tuttujen mekaniikoiden ja pelisuunnittelussa käytettyjen metodien soveltamista oppimismetodeihin, oppimateriaaleihin ja oppimistilanteeseen.

Pelillistämisen avulla on mahdollista stimuloida monia tarvittavia taitoja yhtä aikaa, kuten kriittistä ajattelua ja ongelman ratkontaa. Tämän lisäksi opiskelijoiden on helpompi syventyä ja antautua oppimiselle, kun se on sidottu mielekkääseen tekemiseen. (RoNan 2015.)

Alfonso Gonzales muotoilee pelillistämisen suurimman hyödyn näin*:* ”Every player starts at zero and builds their score, rising up as they progress, as opposed to traditional grading, where every student starts at an A that gets chipped away at throughout the year”*.* (Ronan 2015.)

Tähän realiteettiin oli herätty myös Valterilla, jossa suurin osa jo olemassa olevista oppimateriaaleista koostui joko yksinkertaisista paperilla olevista yhdistämistehtävistä tai muista tämän kaltaisista tehtävistä. Koska Valteri on erityistarvetta vaativille suunnattu koulu, näitä tehtäviä suoritetaan yleensä opettajan tai ohjaajan avustuksella. Yksi esimerkki oppimistilanteesta voisi olla esimerkiksi tilanne, jossa on yhdenlaisia paperista leikattuja kortteja yhdessä joukossa ja sitten toinen joukko toisenlaisia kortteja vieressä. Oppilaan tulee sitten opettajan avustuksella löytää näistä joukoista vastaavat parit. Tämä tilanne muistuttaa jo itsessään paljon enemmän peliä kuin tavallinen oppimistilanne. Saman skenaarion voisikin sangen vaivattomasti pelilistää sähköisesti, jolloin mukaan voisi lisätä kehityksen seurannan, palkinnot ja muut peleistä tutut elementit. Tämä säästäisi myös opettajan aikaa, kun materiaali on valmiina eikä sitä tarvitse tulostella.

Valteri lähti syksyllä 2016 mukaan Jyväskylän ammattikorkeakoulun projektiopintojaksoon tarpeena oppimateriaalien sähköistäminen. Silloinen neljän hengen tiimi lähti rakentamaan järjestelmää tähän tarpeeseen. Järjestelmästä tehtiin kattavat suunnitelmat sen eri osa-alueista ja ominaisuuksista. Toteutustasolla lähdettiin tavoittelemaan ”proof-of-concept”-tason toteutusta, jolla voitaisiin näyttää, että järjestelmässä on potentiaalia.

Projektin lopputuloksena oli järjestelmä, jossa on kaksi osaa: pelaamispuoli ja hallintapuoli. Hallintapuolella opettajat ja muut järjestelmän käyttäjät voivat luoda erilaisia pelejä ja koostaa niistä paketteja. Tehtyjen suunnitelmien mukaan samalla puolella tapahtuu myös pelaajien seurannan eteneminen, pelaajien hallinta, mahdollisen myyntipuolen hallinta ja muut vastaavat. Pelaamispuolella pelaajat voivat pelata erilaisia pelipaketteja ja niissä olevia tehtäviä. Järjestelmä perustuu täysin modulaarisuuteen siten, että järjestelmällä tehtäviä pelityyppejä voi tulevaisuudessa lisätä helposti. Jokaisen pelityypin määrittää niin kutsuttu pelimoduuli, jossa määritellään pelin toimintalogiikka ja sen toiminta editorissa. Pelipuolen tekeminen jäi projektin aikana pahasti kesken. Toteuttamatta jääneistä osioista saatiin melko pitkälti tehtyä suunnitelmat ja niiden olemassaolo on huomioitu esimerkiksi tietokannassa.

1.3 Tarve jatkokehitykselle

Ammattikorkeakoulussa tehty projekti ei toiminnastaan huolimatta ollut vielä juurikaan käyttökuntoinen. Sitä oli hankala käyttää, ulkoasua ei ollut saatu hiottua juuri mitenkään, tunnistautumista ei ollut ollenkaan, käyttöliittymä oli testaamatta, tärkeitä ominaisuuksia puuttui, pelipuoli oli pahasti vaiheessa ja dokumentaatioita ei juurikaan ollut. Projekti kuitenkin osoitti, että järjestelmässä on potentiaalia ja mahdollisuuksia, joten Valteri halusi jatkaa kehittämistä vielä projektin jälkeen. Yksi ryhmän jäsenistä jäikin projektin pariin harjoitteluun. Hän on nyt tehnyt projektia vuoden 2017 toukokuusta asti harjoittelussa ja harjoittelu loppui joulukuussa. Hänen harjoittelussa tekemät työnsä eivät liity tämän opinnäytetyön tehtäviin.

Valterin vastaavien kanssa käytiin alustavia neuvotteluita siitä mitä järjestelmään tarvitaan kaikista eniten ja mitä voidaan siirtää tulevaisuuteen. Nämä tarpeet näkyvät taulukossa 1. Kun opinnäytetyön aloitus alkoi lähestyä, kokoonnuttiin keskustelemaan järjestelmän tilanteesta, kehitystarpeista ja kehityksen roolituksista. Tavoitteeksi muodostui enemmän tai vähemmän saada järjestelmä beta-käyttöön kevääseen 2018 mennessä. Näiden keskustelujen pohjalta luotu suunnitelma on kuvattu liitteessä 1.

Taulukko 1. Valterin kanssa katsotut alustavat kehitystehtävät prioriteeteineen.

Tehtävä	**Tarve**
Tekninen dokumentaatio	Suuri
Tehtävien järjestyksen säätäminen	Suuri
Pelaajan käyttäjätilit	Suuri
Tehtävän monimuokkauksen esto	Suuri
Voittoruutueditori	Keskinkertainen
Ohjeet-välilehti	Keskinkertainen
Moduulityökalu	Keskinkertainen
Ulkoasulliset korjaukset	Keskinkertainen
Luonnospainike	Keskinkertainen
Mediakirjaston parannus	Keskinkertainen
Editorin vasen sivupalkki (quick access)	Pieni

Jatkuu seuraavalla sivulla.

Jatkoa edellisestä sivulta.

Versiointi	Pieni
API rajapinnat tehtävien ja pelien käsittelylle (nykyinen PHP-pohjainen)	Pieni
Mediakirjasto näyttäisi suositellun resoluution kuville	Tehdään jos ylimääräistä aikaa

2 Oppimateriaalien sähköistäminen

Sähköisten oppimateriaalien ja pelillistämisen ympärillä on ollut viime vuosikymmenen aikana paljon positiivista kehitystä, mutta tämä ei ole tapahtunut ongelmitta, ja usein siihen tähtäävissä prosesseissa onkin havaittu parantamisen varaa. Osa näistä ongelmista on ratkaistavissa opinnäytetyössä jatkokehitetyllä järjestelmällä ja osa on sellaisia, että niiden kanssa joudutaan painimaan jatkossa myös tässä projektissa.

Eräs yleinen ongelma pelillistämisessä on, että on hyvin vaikeaa kehittää sellaisia pelejä, jotka sopisivat kattavasti kaikille. Jokin tietty peli saattaa toimia erittäin hyvin yhdellä kurssilla, mutta olla täysi floppi toisella (Jenkins 2016.). Kun tähän yhdistetään se tosiasia, että pelien kehittäminen vie aikaa ja maksaa rahaa, voi pelillistäminen näyttää sangen pelottavalta vaihtoehdolta. Opinnäytetyössä jatkokehitetty järjestelmä ratkaisee tämän ongelman siltä osin, että järjestelmällä kehitetään huomattavasti pienempiä ja tiukasti fokusoituja pelejä. Mikäli tehty peli ei sovi jollenkin tietylle ryhmälle, pelistä voidaan tehdä yksinkertaisesti uusi versio toisen ryhmän tarpeet huomioon ottaen.

Oppimispelien kalleus on myös yksi huomattava ongelma pelillistämisessä. Tyypillisimpiä kustannuksia ovat käyttöottokustannukset, ohjelmistonkehityskustannukset ja koulutuskustannukset. Tämän lisäksi saattaa olla vielä ylläpitokustannuksia ja lisenssimaksuja. (Ford n.d.) Mikäli oppimispelit hankitaan ulkoiselta tekijältä, tulee tähän vielä mukaan tekijän voittomarginaali.

Ylläpitokustannuksia tulee toki myös opinnäytetyössä luodussa järjestelmässä, mutta koska fokus on pienissä tehtävissä, joita opettajat itse toteuttavat, ohjelmistokustannuksia tai siihen päälle tulevia ulkoisen kehittäjän voittomarginaalia ei ole. Lisenssimaksuja ei myöskään joudu järjestelmästä maksamaan. Myöskin koulutuskustannukset pysyvät hyvin rajattuina, sillä järjestelmän pääfokuksena on ollut hyvin intuitiivinen käyttökokemus, jotta tavalliset opettajat ilman sen kummoisempaa teknologista osaamista pystyisivät käyttämään järjestelmää. Käyttöönottokustannusten kohdalla tilanne riippuu pitkälti siitä, mitä niihin lasketaan. Jos kysymys on laitehankinnoista, tämän suhteen ei ole juurikaan eroa, sillä asiakkaalla on jo entuudestaan kattavasti laitteita oppilaille. Mikäli lasketaan asennukseen ja konfigurointiin meneviä kustannuksia, tässä suhteessa kannattaa vertailla näitä kustannuksia suhteessa tahoon, joka ostaisi tämän järjestelmän palveluna, sillä tämäkin malli on ollut mukana järjestelmän suunnittelussa. Tällöin ei varsinaisia pystytyskustannuksia olisi, vaan asiakas maksaisi ainoastaan vuosi- tai lisenssimaksun. Tähän valittiin vertailuksi toinen asiakas, koska Valteri itsessään on saanut tämän sovelluksen asennuksen ja kehityksen käytännössä nollakustannuksilla, sillä sovellusta on tehty opinnäytetyönä ja osana projektiopintojaksoa.

Järjestelmässä helpottuu myös seurannan ja statistiikan seuraamisen ongelma. Tavallisesti pelillistämisessä joudutaan miettimään, miten pelin tulokset ja eteneminen saadaan osaksi kurssia ja miten etenemistä seurataan. (Ford n.d.) Järjestelmässä opettajien vastuulle jää edelleen järkevien kokonaisuuksien kehittäminen suhteessa sen hetkiseen oppimistarpeeseen, mutta tästä huolimatta yhden yhtenäisen järjestelmän ansiosta etenemisen, tulosten ja statistiikan seuraaminen on hyvin helppoa ja data on helposti saatavilla.

Yksi merkittävä ongelma pelillistämisessä nousee esille silloin, kun pelejä tulisi oikeasti ruveta saamaan erilaisille kursseille ja erilaisille opettajille. Pelien markkina liikkuu ja muuttuu nopeasti. Opettajien on vaikea seurata ja käydä läpi olemassa olevaa markkinaa muun opetustyön ohella (Marquis 2012.). Mikäli sen sijaan pelejä kehitetään talon sisällä, nousevat kustannukset kehitystiimin ylläpitämisestä melkoisesti. Ulkopuolelta tilattuna tilanne on vielä pahempi, sillä ulkopuoliset tahot haluavat yleensä sitoutua liiketaloudellisista syistä isompiin projekteihin. Esimerkiksi Psyon

Games pelistudion tekemän Antidote-pelin kehitysaika oli ainakin kaksi vuotta. Peliyrityksillä kehitys on kallista ja odotukset korkealla, kun taas oppilaitoksissa tarpeet voivat muuttua nopeastikin. Projektiopintojaksolla aloitettu ja tässä opinnäytetyössä jatkokehitetty LUEM-järjestelmä on pitkälti suunnattu juuri tähän ongelmaan. Valterin puolella tehtävät ovat yleensä olleet sangen pieniä, joten on loogista, että myös niiden pelillistämiseen tehty järjestelmä keskittyy pienempiin, fokusoituihin tehtäviin. Ohjelmointimaailmassa tällaisen fokuksen hyödyt on tiedetty jo pitkään, mikä näkyy erinäisinä ketterän kehityksen malleina.

3 Atomisuus ja eheys

3.1 Mitä ja miksi

Atomisuus konseptina tarkoittaa jotain, mitä ei pysty enää hajottamaan pienemmäksi. Tietojärjestelmissä atomisuus ilmenee primitiivisenä arvona. Esimerkiksi kokonaisluku on arvo, jolla ei ole lapsenaan mitään muita arvoja, eikä se koostu kuin itsestään. Tietokantatieteessä atomisuudella on kaksoismerkitys. Puhutaan atomisista transaktioista sekä atomisista sarakkeista.

Atominen transaktio tietokannassa on toiminto, jossa kaikki transaktion tehtävät tapahtuvat tai mikään niistä ei tapahdu. Esimerkkinä lentolipun ostaminen, jossa tietokantaan halutaan tallentaa sekä ostettu paikka että itse ostotapahtuma. Mikäli paikan päivitys epäonnistuu tietokannassa, niin atomisen transaktion sääntöjen mukaan myöskin ostotapahtuma jää merkkaamatta tietokantaan. (Atomicity n.d.)

Atominen sarake taas liittyy tietokannan ensimmäiseen normaalimuotoon (1NF, first normal form) eli sääntöön, jonka mukaan yhdessä kentässä voi olla vain yksi tieto ja yhdessä taulussa ei tule olla useita kenttiä kuvaamassa samanlaista tietoa (Tietokannan normalisoinnin perusteiden kuvaus n.d.; What is atomicity id dbms 2014). Esimerkkinä voisi olla tunnisteet-kenttä, jossa voisi olla seuraavan kaltainen merkkijonoarvo: ”tunniste1,tunniste2,tunniste3”. Tämä kuitenkin rikkoo ensimmäistä normaalimuotoa, sillä yhdessä kentässä on nyt monta arvoa. Säännön mukaan tunnisteiden tulisi olla omassa taulussaan, josta ne relaatioilla yhdistetään muihin tauluihin.

Tietokannan eheys on atomisuuden kanssa osa tietokantajärjestelmien ACID-periaatetta (Atomicity, Consistency, Isolation, Durability, suom. Atomisuus, eheys, eristyneisyys, pysyvyys). Eheydellä viitataan tietokannassa siihen, että kannassa oleva tieto kuvaa jatkuvasti sitä, mitä sen on tarkoituskin, kuten että osoitekentässä on osoitteita eikä nimiä, että jokainen rivi on uniikki ja ettei tietokannassa ole relaatioita, jotka eivät viittaa mihinkään. Näitä kolmea konseptia kutsutaan samassa järjestyksessä määrittelyjoukon eheydeksi (domain integrity), entiteetin eheydeksi (entity integrity) ja viittausten eheydeksi (Referential Integrity). (What is Data Integrity 2016.)

Atomisuuden ja yhtenäisyyden vaalimisesta seuraa monia hyötyjä järjestelmälle. Tiivistetysti voidaan todeta, että ilman näitä ominaisuuksia tietokannan data voi olla epätarkkaa, hidasta, tehotonta ja antaa vääriä tuloksia (Poolet 1999).

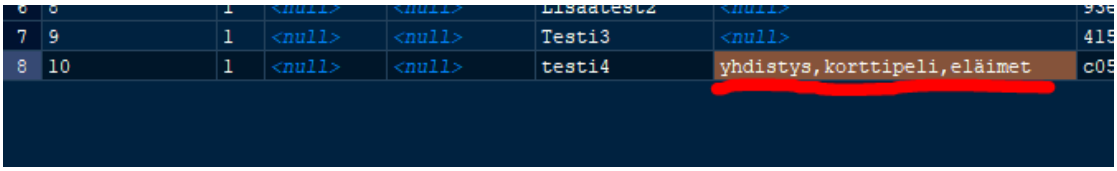
Mikäli tietokannan data on epätarkkaa, se johtaa helposti koko informaation korruptoitumiseen. Esimerkkinä voidaan käyttää tietokantaa, jossa säilytetään rahaliikennettä. Mikäli paljastuu, että viisi päivää sitten tehty rahasiirto onkin asettanut tietokantaan väärän arvon, niin kaikki rahasiirrot sen jälkeen asettuvat hyvin kyseenalaiseen valoon. Atomisen transaktion hyöty ja tarpeellisuus taas syntyy, kun jokin transaktio on hyödyllinen vain, jos se tapahtuu kokonaan. Aiemmin mainittuun esimerkkiin lentolipusta ja paikasta viitaten, jos tietokantaan tallentuu ostettu paikka mutta ei lentolippua, koko tapahtuma on hyödytön. Yksittäisten arvojen normalisointi näkyy pienentyneenä virheherkkyytenä, kun tietoa ei tarvitse kasata ennen tallentamista. Samasta syystä syntyy myös suorituskyvyllisiä hyötyjä, sillä esimerkiksi merkkijonojen käsittely voi olla raskasta.

3.2 Järjestelmän tarkistelu

Yleisten hyötyjen lisäksi järjestelmässä on tiettyjä erikoistapauksia, joissa näitä konsepteja ja niiden toteutumista on täytynyt miettiä tarkemmin. Yksi tällainen ongelma on muun muassa tehtävän tietojen tallennus tietokantaan. Järjestelmän modulaarisuudesta johtuen tehtävän informaation rakenteen määrittää tehtävälle valittu tehtävätyyppi eli moduuli. Tästä seuraa kaksi asiaa, tietokantaan ei pysty luomaan tehtävän tietorakenteelle taulua, koska rakenne vaihtelee ja taulujen luonti lennosta on

sen sijaan tehotonta ja vaarallista ja toisekseen, kun tieto tallennetaan edellä mainitusta syystä JSON-enkoodattuna merkkijonona, niin yhden kentän arvoa muuttamalla on mahdollista muuttaa koko tehtävä toisenlaiseksi. Tämä korostaa eheyden tärkeyttä ja seurantaa siitä, miten tehtävän tieto elää. Tehtävän tallennuksen lisäksi tiettyjen tietueiden kohdalla on tehty valintoja, jotka rikkovat luvun 3.1 periaatteita. Näiden päätösten kohdalla on mietitty päätöksen hyötyjä ja haittoja.

Yksi tällainen ensimmäistä normaalimuotoa rikkova kenttä on kuviossa 1 näkyvä tagit-kenttä, johon tallennetaan tageja eli tunnisteita CSV-muodossa (comma separated values, pilkulla erotetut arvot). Tästä seuraa se, että tunnisteita käsitellessä tulee tämä merkkijono hajottaa taulukoksi ja tietokantaan tallentaessa taulukosta merkkijonoksi. Tähän on päädytty ratkaisun helppouden ja tietokannan yksinkertaistamisen takia. Tageja käytetään järjestelmässä lähinnä entiteetin meta-tietona ja vaikka ne olisivat kuinka väärin, se ei johda itsessään mihinkään kriittiseen. Tämä on yksi hyvä esimerkki siitä, miten käytännöllisyys on ajanut periaatteiden edelle.



7	9	1	<null>	<null>	Testi3	<null>	415
8	10	1	<null>	<null>	testi4	yhdistys,korttipeli,eläimet	c05

Kuvio 1. Tehtävien tags-kenttä tietokannassa

Miksi järjestelmässä ei sitten käytetty moduulien kanssa samaa systeemiä kuin esimerkiksi WordPress-julkaisujärjestelmässä, jossa lisäosan asentaminen luo tarvittaessa uuden taulun lisäosan tarpeisiin? Tämä johtuu siitä, että toisin kuin WordPress, joka asennetaan per käyttötarve, tässä järjestelmässä kehys on oma itsenäinen järjestelmänsä ja moduulien kehittäjät voivat tulla järjestelmän ulkopuolelta. Koska käytössä on Symfony-ohjelmistokehys, tauluille tulisi luoda myös model-tiedostot ja kartoitustiedot, tai sitten käyttää normaalia SQL-kieltä jolloin menetetään ORM:n hyödyt ja virheherkkyys kasvaa. Myös tietokannan rakenne ja relaatioiden ymmärtäminen muuttuisi vaikeaksi ymmärtää ja ylläpitää.

Koska tehtävän monieditointijärjestelmää ei ole suunniteltu eikä ole näillä näkymin tulossa, on ensi sijaisen tärkeää, että vain yksi ihminen voi muokata tehtävää editorissa kerrallaan. Käytännössä editoinnin aloitus ja lopetus ovat siis yhtä pitkää transaktiota. Tämä ratkaistiin luomalla tietokantaan tehtävälle kaksi arvoa, lukitus ja viimeisin aktiviteetti. Kun tehtävä avataan editorissa, asetetaan lukitusarvo todeksi. Tämä kertoo sitten käyttöliittymässä muille käyttäjille, ettei tätä peliä voi nyt editoida. Käyttäjän ollessa editorissa, lähetetään minuutin välein palvelimelle AJAX-kutsu, jolla kerrotaan, että henkilö on vielä editorissa. Kutsu muuttaa tehtävän viimeisimmän aktiviteetin aikaleimaa. Samaan aikaan taustalla pyörii kahden minuutin välein ajettava cron-skripti, joka hakee kaikki tehtävät, joissa viimeisin aktiviteetti on yli kaksi minuuttia sitten ja vapauttaa nämä tehtävät. Näin saadaan käyttäjän toimista riippumaton järjestelmä, joka varmistaa, ettei kantaan jää tehtäviä lukittuun tilaan turhaan.

Muita transaktioita, joissa atomisuus on ensi sijaiseen tärkeää, ei järjestelmässä tällä hetkellä oikeastaan ole. Ainostaan rekisteröitymisessä atomisuuden varmistaminen transaktiossa olisi ensi sijaisen tärkeää, mutta nykyinen toteutus vaatii vain yhden tietokanta kyselyn, joten se joko onnistuu tai sitten ei. Tulevaisuudessa tällainen tarvetta saattaa kuitenkin olla, esimerkiksi mahdollisessa tilausjärjestelmässä ja käyttäjien tietoihin liittyvissä seikoissa. Symfonyn käyttämä Doctrine-kirjasto, jonka läpi tietokantaoperaatioita käytetään mahdollistaa sangen vaivattomasti atomisten transaktioiden käyttämisen.

Lisäksi tietokannassa on muutamia kenttiä, joissa normaalimuotoja voisi viedä vielä pidemmällä. Näitä kenttiä on muun muassa nimi-kenttä, joka pitäisi muuttaa etunimi ja sukunimi kentiksi. Tietokannan rakenne on kuvattu liitteessä 2.

4 Käytetyt teknologiat hyötyineen ja haittoineen

4.1 Symfony

4.1.1 Yleistä

Symfony on PHP:lle tehty ohjelmistokehys web-pohjaisille sovelluksille. Se on yksi suosituimpia PHP-pohjaisia ohjelmakehyksiä. Symfony rakentuu irrallisten, itsenäisten ja uudelleen käytettävien komponenttien varaan. Symfony on myös tietyllä tapaa ideologia. Symfony pyrkii edistämään parhaita käytäntöjä, standardisointia ja yhteensopivuutta. Tämän lisäksi Symfony on itsekin rakennettu täysin komponenttien päälle. (What is Symfony. n.d.)

4.1.2 Symfonyn käyttö projektissa

Järjestelmän MVC-arkkitehtuuri ja HTTP-reititys rakentuvat kokonaan Symfonyn sisäänrakennettujen toiminnallisuuksien päälle. Kansiorakenne on Symfonyn määrittämä ja toiminnallisuudet on tehty Symfonyn komponenteiksi (Service). Tietokantaa operoidaan Symfonyn mukana tulevalla Doctrine-kirjaston avulla ja HTML-sivut on kirjoitettu Twig-pohjilla.

HTTP-kutsujen kiinni saaminen ja sitominen tiettyyn logiikkaan on Symfonyllä helppoa. Tietyille kokonaisuuksille kirjoitettaan niin kutsuttu controller, joka on käytännössä PHP-luokka. Tällä luokalla on metodeja, jotka vastaavat tietynlaisia HTTP-kyselyitä. Symfonyssä pystyy sitten kirjoittamaan niin kutsuttujen annotation-kommenttien avulla, että mitkä HTTP-kyselyt tulee suorittaa kyseisellä logiikalla. Kuviossa 2 näkyy esimerkki siitä, miten tätä annotations-tekniikkaa yksinkertaisimmillaan käytetään. Controllerissa tulee palauttaa aina niin kutsuttu Response-objekti, eli käytännössä lopulta standardimuotoinen HTTP-vastaus. Kuviossa on näkyvillä esimerkkinä controller, joka tässä tapauksessa vastaa pelin asetusten näkymästä. Funktiossa on käytetty Symfonyn service-mallia datan saamiseksi ja annotations-tekniikkaa funktioon liittyvän URL-osoitteen liittämiseen.

```php
class SettingsController extends Controller
{
    /**
     * @Route("/peli/settings")
     */
    public function mainAction() {
        $doc = $this->get("app.databank");
        $userdata = $doc->userData(1);
        return $this->render( view: 'peli/settings.html.twig', [
            'userData' => $userdata,
        ]);
    }
}
```

Kuvio 2. Esimerkki controller-luokasta, joka on vastuussa pelaajan asetuksista.

HTTP-vastaukset ovat yleensä HTML-muotoista. HTML-sivun struktuuri määritetään Twig-nimisellä muottikielellä (template language). Muottikielen avulla on paljon helpompaa ja mukavampaa tehdä näyttölogiikkaa, kuten taulukon läpikäyntiä, kuin se olisi tavallisen PHP:n avulla. Tosin, Symfony tukee myös sen käyttöä. Näyttölogiikka ja HTML kirjoitetaan omiin tiedostoihin, joihin sitten viitataan controllereista. Alla olevassa kuviossa 3 on esimerkki Twig-kielestä.

```twig
<!DOCTYPE html>
<html>
<head>
    <title>My Webpage</title>
</head>
<body>
<ul id="navigation">
    {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
    {% endfor %}
</ul>

<h1>My Webpage</h1>
{{ a_variable }}
</body>
</html>
```

Kuvio 3. Esimerkki Twig-kielestä ja taulukon läpikäymisestä

Tietokantaa käytetään ORM-kirjasto (Object Relational Mapper) Doctrinen läpi. Jokainen tietokannan taulu määritellään Doctrinen haluamalla tavalla PHP-luokaksi ja kun tietokantaa halutaan käyttää, sitä käytetään käyttämällä PHP-luokkaa. Doctrine hoitaa taustalla SQL-kyselyiden, arvojen tarkistamisen ja muun SQL-logiikan. Kyselyitä Doctrinessä tehdään joko käyttämällä QueryBuilder-luokkaa, eli käytännössä OOP-tyylillä (Object oriented programming), tai käyttämällä DQL-kieltä, joka on muunnos SQL-kielestä. Kuviossa 4 ja 5 on näkyvillä esimerkit molemmista tavoista. Esimerkeissä etsitään kuvitteellisesta tuotetaulusta tietyn hinnan ylittäviä tuotteita.

```
$qb = $entityManager->createQueryBuilder('p')
    ->andWhere('p.price > :price')
    ->setParameter('price', $price)
    ->orderBy('p.price', 'ASC')
    ->getQuery();
return $qb->execute();
```

Kuvio 4. Doctrine QueryBuilder-tyylillä

```
$query = $entityManager->createQuery(
    'SELECT p
        FROM App\Entity\Product p
        WHERE p.price > :price
        ORDER BY p.price ASC'
)->setParameter('price', 10);

// returns an array of Product objects
return $query->execute();
```

Kuvio 5. Doctrine DQL-tyylillä

Myös käyttäjien autentikointi on tehty Symfonyn avulla. Siinä käytetään valmista Guard-komponenttia. Tämän lisäksi suurin osa sivun lomakkeista on tehty käyttäen Symfonyn Form Builder komponenttia. Sen avulla lomakkeella kysyttävät asiat voi määrittää controller-funktiossa ja ne saa suoraan sidottua tietokannan tauluihin. Form Builder tekee lomakkeen lähetyksen ohessa automaattisesti Doctrine-objektin lomakkeen arvojen pohjalta. Tämän objektin avulla tiedot siirretään tietokantaan.

4.1.3 Symfonyn hyödyt ja vertailua Laraveliin

Symfony, kuten moni muukin PHP-ohjelmistokehys tekee MVC-pohjaisen web-palvelun kehityksestä hyvin helppoa. Ilman ohjelmistokehystä tulisi projektissa kehittää kaikki pyyntöjen ja vastausten käsittelyyn liittyvä logiikka itse mikä ei ole ainoastaan työlästä, mutta tämän lisäksi siinä on helppo tehdä virheitä. Symfonyn avulla logiikan pystyy helposti jaottelemaan eri controllereihin ja Symfony tuo jokaiseen tällaiseen funktioon tiedot HTTP-pyynnöstä. Näin pääsee vaivatta käsiksi muun muassa POST- ja GET-parametreihin.

Ohjelmistokehystä käyttämällä voikin keskittyä enemmän business-logiikan tekemiseen tai esimerkiksi testeihin, ja näin saadaan tehtyä todellista arvoa järjestelmään. (Why should I use a framework. n.d.)

Symfonyssä oleva ORM-kirjasto Doctrine tuo kehitykseen huomattavia hyötyjä. SQL-logiikan kirjoittaminen on hyvin työlästä käsin ja kehittäjä saattaa hyvin helposti luoda tietoturvaukon, mikäli hän ei ole tarkka tietoturvan kanssa. Tämän lisäksi tietokannasta ulos saatavan tiedon käsittely on yleisesti ottaen raskasta. Se pitää yleensä kääntää objekti- tai taulukkomuotoon käytettävyyden vuoksi. Tällöin tarvitsee mahdollisesti tehdä datan yhdistelyjä ja parsimista. Tämän jälkeen tulee kaikkialla järjestelmässä olla tarkkana, että tietoa käsitellään samalla tavalla. Mikä haluaa tehdä muutoksen tietokantaan, niin muutos pitää päivittää koodissakin mahdollisesti todella moneen paikkaan.

Doctrine ratkaisee tämän ongelman luomalla tietokannan ja ohjelmoijan väliin luokan. Doctrine abstraktoi tietokantaan menevät kyselyt ja käskyt ohjelmoijalta ja ohjelmoijan ei tarvitse kuin asettaa luokan arvoja kuten PHP:ssa normaalistikin. Doctrine luo taustalla tarvittavat SQL-lausekkeet ja käskystä ajaa ne tietokantaan. Mikäli tietokantaan tarvitsee esimerkiksi lisätä kenttä, ei tarvitse kuin muuttaa sitä esittävä luokka, eikä tarvitse miettiä, että jokin muu hajoaisi. Olemassa olevien kenttien tietoja voi myös muuttaa helposti, koska ORM toimii välissä tiettyyn pisteeseen asti abstraktoivana kerroksena.

Suurimpia hyötyjä Doctrinessa on sen käytön luonnollisuus ja suoraviivaisuus. Kun tietokannasta halutaan löytää jotain tietoa, niin käytetään Doctrinen tarjoamia funktioita. Kun tieto on löydetty, sitä voi käsitellä kuin objektia. Relaatioiden asettaminen

tapahtuu myös samalla tavalla. Ero tavallisen SQL:n kirjoittamiseen on valtava. Virheiden määrä tässä vaiheessa vähenee todella paljon ja tietojen käsittely on huomattavasti nopeampaa.

Symfonyn isoin kilpailija Laravel käyttää Eloquent nimistä ratkaisua. Eloquent poistaa Doctrinessa olevaa ”boilerplatea” automatisoimalla toimintoja. Eloquent kuitenkin tuo mukanaan vaikeammin hahmotettavaa ”taikakoodia” eli koodia, joka toimii tietyllä tavalla, mutta jonka syytä ja lähdettä voi olla vaikea löytää. Symfonyn Doctrine tukeutuu määrityksissä PHP Annotations-tekniikkaan, joka antaa kehittäjälle paljon vapauksia. Eloquent sen sijaan liittää kolumnit objektin arvoiksi tietyillä nimeämissäännöillä, jotka kehittäjän tulee tietää.

Autentikaation tekeminen on tehtävä, jonka kanssa tulee yleensä olla hyvin tarkkana ja joka voi osoittautua hyvin työlääksi. Symfonyssä tulee mukana valmis työkalu autentikoinnin tekemiseen. Työkalun nimi on Guard. Guardin avulla on helpohkoa tehdä autentikointi, jonka toimintavarmuudesta ei tarvitse murehtia ja tämän lisäksi Guardia on arvioinut suuri määrä ihmisiä, joten sitä voi pitää melko luotettavana. Tämä säästää myös kehitysaikaa ja resursseja. Verrattuna Laraveliin, tässä ei kuitenkaan ole juuri eroja, sillä Guard on käytössä molemmissa.

Symfonyn mukana tuleva Twig helpottaa HTML-osuuksien kirjoittamista. Normaalin PHP:n kirjoittaminen on melko työlästä, sillä syntaksi on raskas. Tämän lisäksi PHP:n avulla saattaa helposti aiheuttaa tietoturvaongelman koodiin. Näin voi käydä esimerkiksi silloin, jos HTML:n kanssa oleva PHP-koodi aiheuttaa virheen. Virheviesti päätyy HTML-rakenteeseen ja voi näin ollen paljastaa tietoa, joka voi auttaa hyökkääjää. Twigissä HTML:n kirjoittaminen tapahtuu muuten normaalisti, mutta kaikki logiikka tapahtuu Twigin omalla syntaksilla, jonka kirjoittaminen on huomattavasti tavallista näppärämpää. Tämän lisäksi Twig ajaa kaiken koodin hiekkalaatikossa, eli eristettynä muusta koodista. Tämä tekee siitä turvallisen. (Why yet another template engine? n.d.)

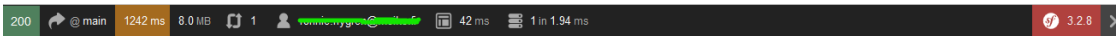
Laravelissa on käytössä Blade-pohjamoottori. Suurimpana erona Twigiin, Blade käyttää @-prefiksattua syntaksia ja muu PHP:n kirjoitus on käytännössä tavanomaista PHP:ta, siinä missä Twigissä se on Twigin syntaksia. Blade suosii siis enemmän tuttua PHP:ta ja on tällä osaa ehkä maanläheisempi kuin Twig, mutta samalla mukana tulee

PHP:n syntaksin omat ärsyttävyydet. Samalla myös menetetään Twigin hiekkalaatikkoon rajoitettu turvallisuus.

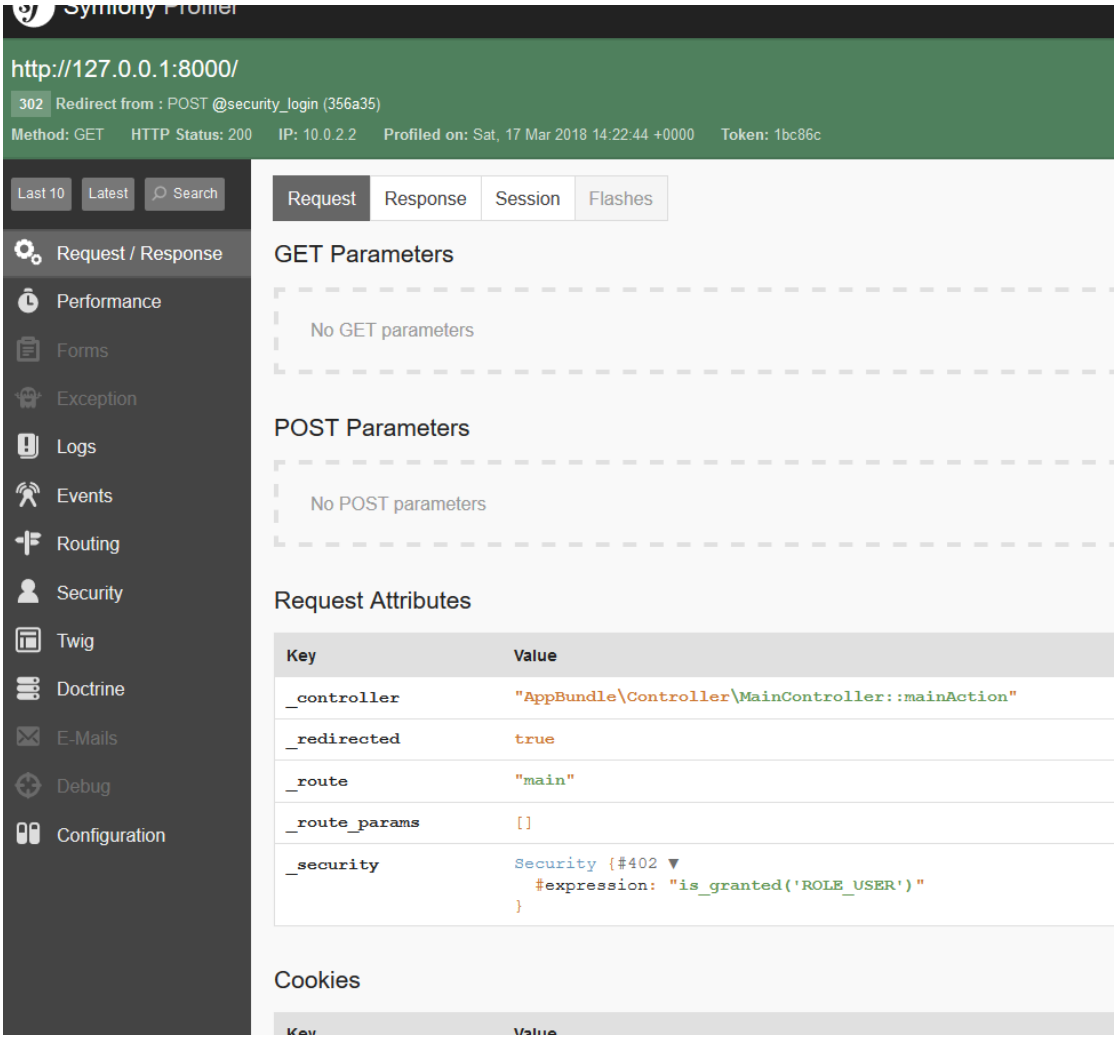
Symfony on rakennettu komponenttien päälle ja komponenttius on myös Symfonyn kantava idea. Symfonyyn on helppoa ottaa käyttöön muita komponentteja lisäosina, jotka sitten toimivat sulavasti yhteen Symfonyn muiden osien kanssa. PHP-kirjastojen käytössä itsessään ei ole juuri mitään erikoista, mutta Symfonyn service-rajapinta mahdollistaa juuri Symfonyä varten optimoitujen kirjastojen teon. Muun muassa aiemmin mainittu Guard oli aikaisemmin erillinen pakettinsa. Samoin Doctrine on oma pakettinsa ja niin on myös Twig. Ne vain sattuvat tulemaan Symfonyssä valmiina. Samojen komponenttien hienous on myös siinä, että niitä voi käyttää myös Symfonyn ulkopuolella. Esimerkiksi Symfonyn Routing-komponenttia käytetään myös Drupalissa ja Laravelissa. Komponentit mahdollistavat JavaScript puolelta tutun kirjastojen helpon käyttöönoton suoraan Symfonyyn toimivaksi.

Sama service-patterni on käytössä myös silloin, kun projektiin itsessään tehdään itsenäisiä toiminnallisuuksia. Nämä kokonaisuudet saavat service-rajapinnan kautta pääsyn suoraan Symfonyn sisälmyksiin. Symfonyssä on toiminto nimeltä Auto-Wiring jonka avulla voi määrittää mitä Symfonyn toimintoja palvelu tarvitsee. Esimerkiksi moduulijärjestelmä käyttää Doctrinea sisällään.

Symfonyssä on myös valmiit kehitystyökalut, joiden kautta pystyy muun muassa näkemään mikä sivun lataamisessa kestää, mitä tietokanta kyselyitä tehtiin ja onnistuivat ne, mitä reittejä pitkin HTTP-pyyntö käsiteltiin, onko järjestelmässä kirjautuneena, erinäisiä Symfonyn ja PHP:n tietoja ja muistin käytön. PHP:n debuggaus on tunnetusti melko hankalaa ja kehitystyökalujen avulla Symfony pelastaa paljon kehittäjältä. Kehitystyökalujen avulla on helppoa tutkia niin selkeitä ohjelmointivirheitä kuin myös sivun tai sovelluksen suorituskykyä. Kehitystyökalut tulevat sivun alareunaan kuvion 6 osoittamalla tavalla. Palkista pääsee painamalla laajennettuun näkymään, mikä näkyy kuviossa 7.



Kuvio 6. Kehitystyökalut tulevat sivun alaosaan palkkina.



Kuvio 7. Kehitystyökalut avaamalla saa tarkempia tietoja muun muassa kekseistä, sessiosta ja tietokannasta.

Symfonyn ja Doctrinen kanssa on myös helppo käyttää Migrations nimistä toimintoa. Tämän avulla pystytään jokainen muutos tietokantaan dokumentoimaan migraatio-tiedostoihin. Kehittäjän tarvitsee sitten vain ajaa migraatiot ja näin hän saa itselleen uusimman version tietokannasta. Halutessaan migraatiot voi myös perua, mikäli jokin migraatio tekikin rikkovan muutoksen. Näin vältytään tilanteilta, joissa tekijöillä on tietokannat eri vaiheissa ja asiat eivät toimi oikein.

Myös Laravelissa on migraatiot. Laravelin migraatiot kirjoitetaan Blueprint-luokan metodeja hyödyntäen, kun taas Doctrinessa käytetään tavallista SQL-kieltä. Laravelin tavan käyttö voi olla helpompaa, mikäli ei ole erikoistunut tavalliseen SQL-kieleen. Doctrinessa on kuitenkin yksi loistava ominaisuus jota Laravelin puolelta ei löydy.

Doctrinessa pystyy tekemään muutokset suoraan ja ainoastaan malleihin ja konsolikomennolla generoimaan migraatiot automaattisesti. Laravelin puolella pitää tehdä muutokset sekä malliin, että migraatioon.

Ohjelmakehysten hyötynä on yleisesti ottaen se, että mikäli tiimiin tulee uusi kehittäjä, on paljon helpompaa päästä sisälle projektiin, joka on tehty tunnetun kehyksen päälle ja josta on kattavat dokumentaatiot, kuin järjestelmään joka on täynnä omia kustomisoituja ratkaisuja. Mitä tunnetumpi ohjelmakehys, sitä todennäköisemmin uudella kehittäjällä on jo entuudestaan kokemusta sen käytöstä. Näin uusista kehittäjistä tulee tiimille hyödyllisempiä nopeammin ja on helpompaa hahmottaa mikä on sovelluksen bisneslogiikkaa ja mikä ympäröivää infraa. Symfonyssä sisäinen koodi myös pyritty pitämään mahdollisimman yksinkertaisena, jotta tarvittaessa kuka tahansa pystyy tutkimaan, että miten se toimii.

4.1.4 Symfonyn ja muiden ohjelmistokehysten huonot puolet

Ohjelmistokehyksien ongelmana on, että niillä on tapana olla melko raskaita. Ne on kehitetty palvelemaan tarpeita pienemmistä palveluista valtaviin kokonaisuuksiin. Ohjelmistokehykset määrittävät säännöt kehitykselle ja pyytävät kehittäjää toimimalla tietyllä tavalla. Voi hyvinkin olla, että käsillä oleva tehtävä on yksinkertaisesti niin pieni, ettei ison ohjelmistokehyksen tuominen projektiin anna juurikaan hyötyjä. Pahimmassa tapauksessa ohjelmistokehyksestä voi olla jopa haittaa tai se voi hidastaa kehitystä. Toisaalta, osaa Symfonyn komponenteista voi käyttää myös sellaisenaan, mikä voi olla joskus parempi ratkaisu pienemissä projekteissa.

Koska ohjelmistokehykset pyytävät kehittäjää toimimaan tietyllä tavalla, ohjelmistokehyksen tavalla, seuraa tästä aina kehittäjälle oppimiskäyrä. Tietyt kehykset ovat vaikeampia oppia kuin toiset. Yhtä kaikki, tämä opiskeluun menevä aika on pois itse kehityksestä ja korkea oppimiskäyrä johtaa väistämättä virheisiin ajan myötä. Ohjelmakehystä valitessa tuleekin pohtia, että kuinka paljon oppimista kyseinen kehys vaatii ja mitä se tarjoaa vastineeksi siitä. Symfonyn tapauksessa oppimiskäyrä on suhteellisen pieni, verrattuna joihinkin muihin järjestelmiin kuten Yii tai Laravel. Tämä selittyy osittain sillä, ettei Symfony yritä olla yhtä kattava kuin edelliset, vaan tähtää enemmän aloituspisteenä olemiseen. Toisaalta taas, esimerkiksi Doctrinella tuppaa

olemaan aivan päin vastainen maine, sitä pidetään vaikeana sisäistää. Symfonyllä on kuitenkin saatavilla valtava määrä tutoriaaleja ja dokumentaatiota.

Ohjelmistokehykset abstraktoivat yleiset ja toistuvat tehtävät kehittäjältä siten, että tämä voi keskittyä olennaiseen. Tämä on mukavaa, ainakin siihen asti, että tulee ongelmia. Omasta koodistaan kehittäjä tietää, että miten se toimii ja miten sitä voi muokata, mutta ohjelmistokehyksen sisusten kanssa voi mennä nopeasti sormi suuhun. Tähän ongelmaan voi päätyä, mikäli ohjelmistokehyksessä itsessään on ohjelmointivirhe, mikä tosin on onneksi sangen harvinaista suurissa ja suosituissa kehyksissä. Isompana ongelmana abstraktointi kuitenkin näkyy ongelman selvittämisessä. Abstraktointi tarkoittaa, ettei kehittäjä näe kokonaiskuvaa ja tällöin ongelman lähteen paikantaminen voi muuttua hyvin vaikeaksi. Abstraktointi näkyy myös mahdollisissa virhelokeissa. Ongelman lähteen seuranta pysähtyy ikävästi, kun väliin tulee kehyksen sisäiset palikat. Symfonyn puolesta voinee tässä sanoa, että Symfonyn lähdekoodi on sinällään melko helposti luettavissa, mikä voi auttaa jonkin verran. Kuitenkin, Symfony kärsii tästä ongelmasta siinä missä muutkin kehykset.

4.2 React

4.2.1 Yleistä

React on Facebookin kehittämä ja ylläpitämä JavaScript-kirjasto, joka on suunnattu itsenäisten ja päivittyvien näkymien luomiseen. React ei ole JavaScript-kehys, kuten esimerkiksi Googlen Angular on, vaan se keskittyy yhteen tehtävään, käyttöliittymien tekemiseen, ja on todella hyvä siinä. Jos mietitään perinteistä MVC-mallia, niin React keskittyy ainoastaan V, eli View-kohtaan. (Samer Buna, 2017)

Reactia voisi kuvailla deklaratiiviseksi. Ohjelmoija kuvaa mitä haluaa näkyviin ja React rakentaa tämän näkymän annettujen ohjeiden pohjalta. Erona tavalliseen tapaan on, että normaalisti ohjelmoijan tulisi itse huolehtia näkymän rakentamisesta DOM-puuhun. (Samer Buna, 2017).

React loistaa parhaiten niin kutsutuissa yhden sivun applikaatioissa (single page application, SPA). Tällaiset sovellukset eivät koskaan siirry alkuperäisestä sivulta, vaan

sisältö muuttuu dynaamisesti sen mukaan mitä käyttäjä tekee. Normaalisti tämä vaatii, että ohjelmoija kirjoittaa ensiksi logiikan kaikille sellaisille toiminnoille, jotka muokkaavat sivun sisältöä jollain tapaa. Tämän jälkeen ohjelmoijan tulee kehittää jokin tapa tallentaa sovelluksen globaalia tilaa eri näkyvien välillä. Lopuksi täytyy vielä luoda järjestelmä, joka varmistaa, että sovelluksen datan muututtua sivun sisältö reakoi oikealla tavalla. HTML-standardia ja JavaScript-kieltä ei ole koskaan optimoitu tällaiseen käyttötarkoitukseen. Se on hyvinkin mahdollista, mutta työlästä, virheherkkää ja sekavaa.

Reactin deklaratiivisessa mallissa viimeisen vaiheen käsittely muuttuu naurettavan helpoksi. Yksinkertaisimmillaan yksittäisen React-komponentin voi ajatella funktiona, jollaisen sen pystyy myös kirjoittamaan. Funktiota kutsutaan tietyillä parametreilla ja parametreista riippuen ulos saadaan jotain, tässä tapauksessa DOM-komponentti. Oikeastaan ulos tulee toinen funktio, johtuen Reactin tavasta toimia, mutta yksinkertaistuksena voidaan ajatella, että ulos tulee komponentti. Tämä ratkaisee aikaisemmasta ongelmasta yhden kolmanneksen. Aina kun tila muuttuu, kutsutaan funktiota ja ulos tulee uusi DOM-puun, tai yksittäisen DOM-elementin uusi tila. Jotenkin pitäisi vielä kuitenkin vielä tietää milloin funktiota kutsutaan.

Reactissa jokaisella komponentilla on tila. Komponentin tila on kapsuloitu, eikä siihen pääse ulkopuolelta käsiksi. Tämän lisäksi komponentilla on ominaisuudet (prop, property). Ominaisuudet vastaavat käytännössä aiemmin mainittuja funktion parametrejä, eli ne tulevat komponenttiin ulkopuolelta. Sisäisesti React kiinnittää jokaisen komponentin sen sisäiseen tilaan siten, että kun tila muuttuu, niin komponentti ikään kuin rakennetaan uusiksi. Kun komponentti itse päivittyy, se päivittää samalla kaikki sen lapsikomponentit, jotka saavat uuden tiedon ylhäältä alaspäin parametreissa. Komponentti voi antaa lapsilleen parametrina funktion, joka aktivoituessaan päivittää funktion määrittäneen komponentin tilan. Kun käyttäjä aktivoi funktion, päivittyy komponentin tila ja siinä sivussa myös lapset. Näin Reactin maailmassa tilan muutoksiin kiinnittyminen muuttuu lähes automaattiseksi.

Alkuperäisestä ongelmasta on vielä viimeinen osa jäljellä, eli miten siirtää komponentin uusi näkymä osaksi DOM-puuta? React abstraktoi tämän vaiheen kokonaan pois kehittäjältä. Ohjelmoijan tarvitsee ainoastaan huolehtia siitä, että saa komponentin

tilan muuttumaan ja React hoitaa loput. Sisäisesti React tutkii muistissa olevaa virtuaalista DOM-puun presentaatiota ja mikäli on tapahtunut muutos, React siirtää muuttuneen osan DOM-puuhun. React ei siis muuta yksittäisiä elementin arvoja, vaan ylikirjoittaa koko muuttuneen alueen uudella tilalla ja tämän lisäksi React rajoittaa muutoksen vain siihen osaan, joka oikeasti muuttui. DOM-puun käsittely on tunnetusti sangen hidasta sekä raskasta ja React on ratkaissut tämän ongelman siten, ettei itse DOM-puuta vasten koskaan verrata mitään ja vain rajattu alue muuttuu. Tuloksena ohjelmoijalle ei ole ainoastaan automaattinen muutosten siirtyminen DOM-puuhun, vaan myös hyvin nopea ja tehokas tapa tehdä se.

4.2.2 Reactin huonot puolet

React on työkalu siinä missä kaikki muutkin kirjastot ja sen käyttöönotossa tuleekin aina miettiä, että minkä ongelman se ratkaisee ja onko siitä oikeasti hyötyä. Hyötyjen lisäksi tulee myös ymmärtää mitä haittapuolia Reactissa on.

Ensimmäinen ongelma on hyvin yksinkertainen. Reactin käyttö vaati melkoisen paljon pystytystä projektissa ja käyttäminen lisää tarvittavan koodin määrää, jotta saataisiin toteutettua yksinkertainen tehtävä. Toimiakseen React vaatii käytännössä BabelJS-tulkin kääntämään syntaksia, Webpack-pakkaustyökalun, mikä ei ole pakollinen, mutta oletus kun Reactia käytetään ja kaksi eri skriptiä toimiakseen. Tämän jälkeen yksinkertaisen sivun tekeminen, jolla lukee, että ”Terve Maailma” sisältää vielä monta vaihetta. Ensiksi täytyy tehdä indeksitiedosto HTML-rakenteelle, sitten kirjoittaa Reactin käynnistävä initialisointikoodi ja sen jälkeen vielä määrittää JavaScript-luokka, joka määrittää komponentin, jossa lukee jossain elementissä: ”Terve Maailma”. Ja ennen kuin tätä voi käyttää, täytyy sivun koodi ajaa Babel-tulkin läpi. Olkoonkin, että kyseessä on hyvin yksinkertaistettu esimerkki, sama pätee myös hieman monimutkaisemmissa tapauksissa. Sen minkä pystyi aikaisemmin helposti tekemään tapahtuman kuuntelijalla ja asettamalla elementtiin arvoja, täytyykin Reactissa nyt tallentaa komponentin tilaan dataa ja sitten siirtää tilaa lapsikomponenteille sekä samalla määrittää näille funktiot tilan päivittämiselle.

Reactissa on myös sangen korkea oppimiskäyrä. Suurin syy tähän on, että kehittäjien täytyy muuttaa heidän ajattelutapansa ja ajatella niin sanotusti Reactin tavalla. Tämä on henkisesti vaativa toimenpide ja vaatii aikaa ja virheiden tekemistä. Tämän ajan

jakson aikana voi tuntua helpolta vain todeta, ettei React ole itselleen sopiva. Kun Reactin ajatusmaailman sisäistää, niin sen jälkeen sen käyttäminen helpottuu huomattavasti. Reactin oppimiskäyrää nostaa myös siinä tarvittavat lisätyökalut, jotka voivat olla sangen monimutkaisia. Viimeiseksi, tiettyjen ihmisten mielestä Reactia vaikeuttaa siinä pitkälti käytetty ES6 JavaScript-syntaksi, joka tuo JavaScriptiin muun muassa luokat ja nuolifunktiot.

Reactilla kehittämisessä ongelmaksi saattaa myös muodostua äärimäisen epäselvä virheviesti. Tämä on ajan myötä parantunut, mutta edelleenkin on tapauksia, joissa kannattaa virheviestin kanssa lähteä suoraan Googleen. Tämä voi tehdä kehittämisestä hidasta ja ärsyttävää, varsinkin Reactiin tutustuville, joille tällaisia virheviestejä tulee useammin.

Viimeiseksi, kuten aiemmin mainittiin, React on vain käyttöliittyminen määrittämiseen tehty kirjasto. Se ei ota mitään kantaa bisneslogiikkaan tai datan rakenteeseen, vaan tämä jää ohjelmoijan vastuulle. Vertailun vuoksi, markkinoilla on myös paljon kirjastoja, jotka tarjoavat kokonaisvaltaisia ratkaisuja. Se, että kumpi on parempi vaihtoehto, riippuu täysin tekijästä ja projektista. Reactia käyttäessä onkin yleistä, että sen kylkeen kytketään erikseen muita kirjastoja, kuten tilanhallintaan käytetty Redux ja sivuvaikutusten tekemiseen käytetty Redux-Saga. Joidenkin mielestä tämä vapaus on hyvä asia, jotkut taas haluavat valmiita ratkaisuja. Reactille on myös kehitetty niin kutsuttuja boilerplate-kirjastoja, jotka Reactin lisäksi asentavat muita kirjastoja ja työkaluja yhdellä kertaa.

4.2.3 React tässä projektissa

Tässä projektissa käytettiin Reactia, mutta kyseessä ei ole kuitenkaan yhden sivun applikaatio. Yksi käyttösyistä on ollut puhtaasti Reactin testaaminen, mikä ei kuitenkaan ole hyötyjen valossa kovin hyvä syy. Tätä kulmaa vasten onkin hyvä tarkastella, että voiko Reactista saada hyötyjä myös SPA-applikaation ulkopuolella.

Reactin käyttökohteeksi muotoutuivat lopulta dynaamiset komponentit. Järjestelmän sivut tarjotaan PHP:lla ja rakenne määritetään Twig-pohjakielellä. Sivun yksittäiset osaset saattavat sitten käyttää Reactia. Yksi hyvä esimerkki tällaisesta käyttötavasta löytyy itse editorinäkymästä. Näkymässä on kaksi React-komponenttia: yksi oikean

puolen toiminnot-palkille ja toinen vasemman puolen pikalistaukselle. Pikalistauksessa listataan käyttäjän viimeisimmät muokatut tehtävät ja pelit. Listassa on myös hakutoiminto. Reactin avulla on yksinkertaista tehdä komponentti, joka päivittää hakusanan sisäiseen tilaansa ja aina render-funktiossa filtteröi tehtävien listaa jonkin hakusanan avulla. Konditionaalisen renderöinnin helppous on Reactin vahvuuksia, sillä pohjimmiltaan render-funktiokin on ihan vain JavaScript-koodia. Ilman Reactia pitäisi tehdä hakupalkille oma kuuntelija, sitten pitää hakusana jossain muistissa ja lopuksi tehdä funktio, joka ensiksi ottaa nykyisen listan DOM-puusta, parsii arvot ja rakentaa uuden DOM-elementin uudella järjestyksellä. Vaihtoehtoisesti listaa pitää jossain globaalissa muistitilassa. Alla olevat kuviot 8-9 ovat tehtävien listauksesta ja ne kuvaavat vaaditun logiikan haun toteuttamiselle. Kuten kuviosta näkyy, on tällaisen ehdollistetun renderöinnin tekeminen Reactissa helppoa.

```
render() {
    let self = this;
    let txt = this.state.filterText;
    let reg = new RegExp(txt,"i");
    if (assignmentsLoaded) {
        var assignmentNodes = this.props.assignments.map(function (assignment, i) {
            if (reg.test(assignment.gameName) || txt == '') {
            return (
                <AssignmentBox
                    title={assignment.gameName}
                    gameIcon={assignment.imgUrl}
                    status={assignment.published}
                    underEdit={assignment.isUnderEdit}
                    desc={assignment.description}
                    tags={assignment.tags}
                    values={assignment.values}
                    id={assignment.gameId}
                    key={i}
                    index={i}
                    isActive={self.state.activeIndex === i}
                    onClick={self.handleClick.bind(self)}
                >
                    {assignment.assignment}
                </AssignmentBox>
            );
        }
        });
    } else {
```

Kuvio 8. Listauskoodi, jossa RegEx-testillä katsotaan, että halutaanko tätä tehtävää näyttää.

```
class SearchBar extends  React.Component {
    handleChange() {
        this.props.onUserInput(this.refs.filterTextInput.value);
    }
    render() {
        return (
            <input
                type="text"
                placeholder="Etsi..."
                value={this.props.filterText}
                ref="filterTextInput"
                onChange={this.handleChange}
            />
        );
    }
}
```

Kuvio 9. Hakupalkki itsessään on täysin itsenäinen komponentti. Esimerkissä onUserInput asettaa hakusanan aiemman komponentin sisäiseen tilaan.

Toinen hyvä esimerkki Reactin käytöstä on editorin toimintopalkki. Tämä toimintopalkki muuttaa näkyvissä olevia toimintoja sen mukaan, minkä tyyppinen elementti itse editorissa on valittuna. Koska editori itsessään taas on Phaser-pohjainen, täytyy Phaser-pelissä tapahtuva muutos saada siirrettyä sivupalkin tietoon. Tämä on tehty siten, että sivupalkki haluaa vain listan toiminnoista ja renderöi sitten toimintoja tämän listan pohjalta. Tiedon siirtäminen editorista React-komponentille tapahtuu siten, että luotu komponentti otetaan globaalisti talteen ja koska Reactin komponentit ovat pohjimmiltaan JavaScript-luokkia, niin editori voi kutsua tämän luokan funktiota setState. Tämä saa komponentin reagoimaan tilan muutoksen ja päivittämään listan. Tämä logiikka on kuvattu seuraavalla sivulla olevassa kuviossa 10.

```
class EditorActionsList extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            actions: EditorData.sidebarActions,
            element_actions: []
        };
    }
    render(){
        return (
            <ul >
                {this.state.element_actions.map(function (item) {
                    return <button
                        type="button"
                        key={item.name}
                        onClick={function(){callSystemEditorAction(item.action.func_name,item.action.params)}}
                        className="button-success">
                        {item.name}
                    </button>
                })}
                <hr/>
                {this.state.actions.map(function (item) {
                    return <button
                        type="button"
                        key={item.name}
                        onClick={function(){callSystemEditorAction(item.action.func_name,item.action.params)}}
                        className="button-primary"
                        dangerouslySetInnerHTML={formatHTMLtoText(item.name)}
                    />
                })}
            </ul>
        )
    }
}
function callSystemEditorAction(func_name, params){...}
function formatHTMLtoText(string){...}
let EditorActionListNode = ReactDOM.render(
    <EditorActionsList />, document.getElementById("editor-actions")
);
```

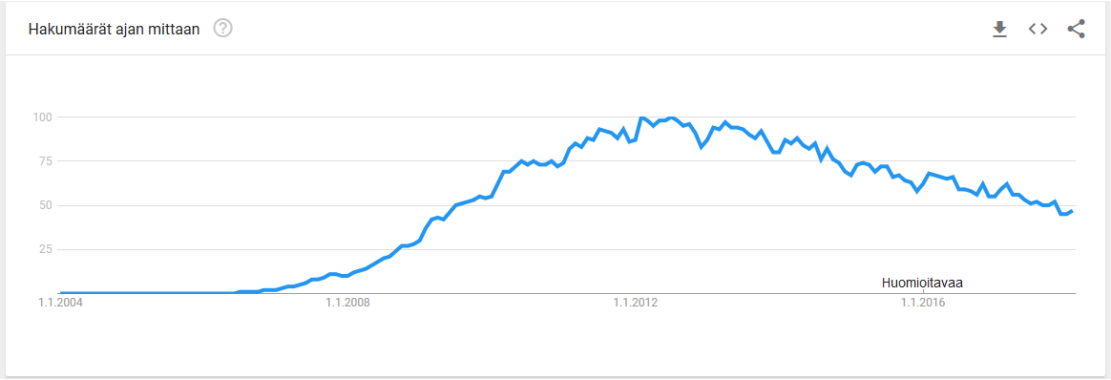
Kuvio 10. Noin 50 riviä koodia riittää dynaamisen sivupalkin luomiseen. Huomaa viimeiset kolme riviä, jossa sivupalkki tallennetaan globaalisti EditorActionListNode-muuttujaan.

Edellä kuvatut esimerkit valoittavat miten helppoa dynaamisten ja reaktiivisten komponenttien teko Reactilla on. Yhden sivun applikaatioiden lisäksi React voikin olla oiva työkalu myös yksittäisten, nimensä mukaisesti reaktiivisten, komponenttien luontiin. Voidaankin siis sanoa, että Reactin käyttö projektissa on hyvin perusteltua, vaikkakin projektista löytyy myös muutama kohta, jossa toteutuksen olisi voinut tehdä myös ilman. Esimerkkinä tällaisesta toimii listausnäkymät, joissa ei ole hakupalkkia. Reactin hyöty listausnäkymissä on, että lista päivittää itsensä tietyin aikavälein, eli se on dynaaminen lista, mutta tämän päivittämisen hyöty voidaan kyseenalaistaa suhteessa sen aiheuttamaan ylimääräiseen koodiin. Listauksen voisi staattisena kirjoittaa myös suoraan Twig-pohjaan, joka saa tehtävälistan suoraan palvelimelta.

4.3 jQuery

jQuery on JavaScript-kirjasto, joka on alun perin kehitetty tekemään JavaScript-kehityksestä lähestyttävämpää. Ennen jQueryä oli hyvin työlästä tehdä koodia, joka olisi toiminut kaikilla eri selaimilla. JavaScript-tuessa eri selainten välillä oli huomattavia eroa ja selaimilla oli omia funktioitaan eri asioille. Tämän lisäksi JavaScript on aina ollut sangen verbaalinen kieli. jQuery ratkaisi nämä ongelmat tekemällä DOM-manipulaatiosta todella yksinkertaista ja helppoa, sen lisäksi, että se toimi kaikilla selaimilla. Kirsikkana kakun päällä, jQuery teki myös animaatioista ja AJAX-kutsuista hyvin yksinkertaisia. (Simon Hamp, 2015.)

Viime vuosien aikana jQueryn suosio on ollut laskemaan päin, mikä on nähtävillä myös kuviossa 11. Uudet ja suositut kirjastot kuten React, Vue ja Angular ovat syöneet jQueryn piirasta. Esimerkiksi React abstraktoi DOM-puun kokonaan, joten jQueryn isoin käyttötarve jää React-pohjaisissa sovelluksissa lähes kokonaan pois. Tämän lisäksi JavaScript-kielen kehittyminen on parantanut sen heikkouksia ja selaintuki alkaa olla lähes identtinen uusimmissa selaimissa.



Kuvio 11. jQueryn suosio Googlen hakutuloksissa

JQuery on kuitenkin loistava siinä asiassa mitä se tekee parhaiten eli DOM-manipulaatiossa. Mikäli DOM-manipulaatiota joutuu suorittamaan, on sen tekeminen jQueryn avulla huomattavan paljon mukavampaa. Yksi tällainen perinteinen käyttötapa voisi olla esimerkiksi animaatiot. Toisekseen, esimerkiksi Reactissa API-kutsujen suorittaminen jää täysin ohjelmoijan vastuulle. Miettiessä sopivaa ratkaisua

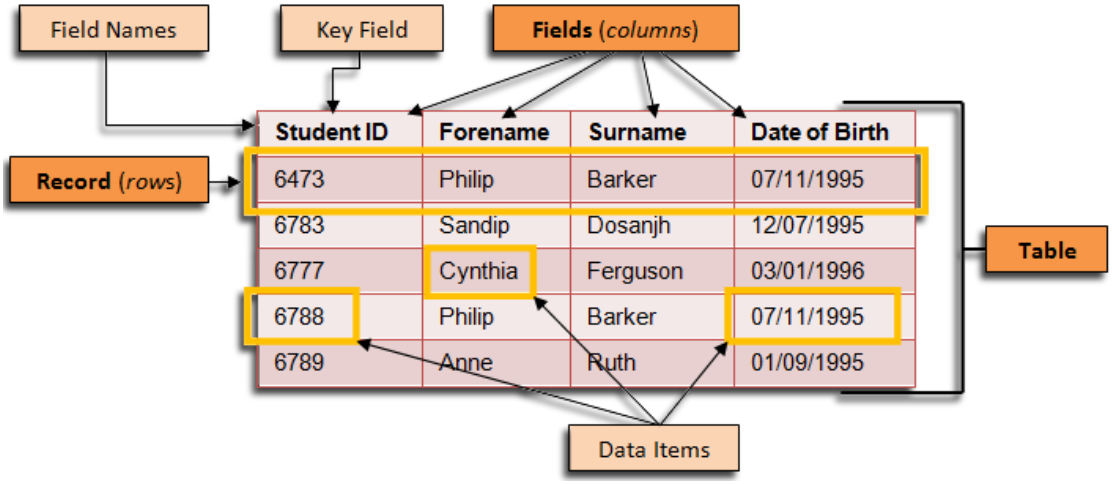
näiden tekemiseen, on jQueryn $.ajax varteen otettava vaihtoehto. Tämä onkin yleisin käyttötapa jQueryllä tässä projektissa. Loppujen lopuksi ratkaiseekin se, että saako jQueryn käytöstä sellaisia hyötyjä suhteessa paketin kokoon, että se kannatta ottaa osaksi projektia.

4.4 SQL-tietokanta

4.4.1 Teknologia

Järjestelmässä käytetty tietokanta on MariaDB-niminen MySQL-tietokantaan pohjautuva relaatiotietokanta. MariaDB on alun perin MySQL:stä tehty suora kopio. Tämä tehtiin sitä varten, että jos MySQL:n ostanut Oracle muuttaisi MySQL:n pois avoimesta lähdekoodista, MariaDB olisi edelleen saatavilla avoimena ja ilmaisena. MariaDB on yhteisön kehittämä ja täysin yhteensopiva MySQL:n kanssa. Ajan myötä MariaDB:seen on kuitenkin tehty huomattavia parannuksia suhteessa MySQL:ään. Tällaisia parannuksia ovat esimerkiksi kattavampi SQL-standardin toteutus, parannettu suorituskyky, JSON-tuki, enemmän tuettuja tietokantamoottoreita ja vähemmän virheitä. (MariaDB versus MySQL – Features, n.d.)

Relaatiotietokanta on tietokanta, jossa tietoa säilytetään tauluissa ja taulun sisällä tieto esitetään riveissä ja sarakkeissa. Yksi taulu on kokoelma, yksi rivi on entiteetti ja yksi sarake on entiteetin ominaisuus. Jokainen rivi yksilöidään uniikilla tunnuksella, joka erittelee eri entiteetit toisistaan. Eri taulut liittyvät toisiinsa loogisilla suhteilla, eli käytännössä entiteetin tiedoissa on jonkin toisessa taulussa olevan rivin tunniste. Taulun rakennetta visualisoidaan kuviossa 12. Järjestelmää, joka hallinnoi relaatiotietokannan tiedon säilytystä, hakemista, kirjoittamista ja muita toimenpiteitä, kutsutaan nimellä Relational Database Management System tai lyhennettynä RDBMS. (A Relational Database Overview, n.d.)



Kuvio 12. Relaatiotietokannassa olevan taulun läpileikkaus (What is a Relational Database? n.d.)

4.4.2 Hyödyt

Järjestelmässä päädyttiin käyttämään SQL-pohjaista tietokantaa. Tähän päädyttiin, koska sen hyödyt voittivat todennetut haitat. Yksi suurimmista hyöty tekijöistä oli, että käytetty Symfony-ohjelmointikehys toimii suoraan ja helposti SQL-tietokannan kanssa. Tämän lisäksi tiimin jäsenillä oli eniten kokemusta SQL-pohjaisista tietokannoista.

SQL-pohjaisten tietokantojen hyötynä on, että niissä on yleensä toteutettu hyvin pitkälti kaikki relaatiotietokannan ja SQL-standardin vaatimukset. Näitä ovat esimerkiksi ACID-pohjaiset operaatiot, koontifunktiot ja tietysti relaatiot. Verrattaessa NoSQL-pohjaisiin ratkaisuihin, niin osassa on osa näistä funktioista ja osassa taas vähän huonommin. Missään näistä ei kuitenkaan ole yhtä kattavaa ja pitkälle vietyä ekoympäristöä kuin SQL-pohjaisissa tietokannoissa ja onkin myös yleistä, että tiettyjen SQL-tietokannoissa automatisoitujen toimintojen toteutus on tarkoituksella jätetty käyttäjän vastuulle NoSQL-ratkaisuissa. ACID:n toteutumisen hyödyt on selitetty paremmin luvussa 3.

Koska SQL-pohjaisia ratkaisuja on ollut olemassa jo niin pitkään, löytyy näille huomattavan hyvin tukimateriaaleja ja valmiita ratkaisuita, siinä missä NoSQL ratkaisut ovat yleensä paljon uudempia ja nopeasti kehittyvimpiä. SQL ja MySQL ovat myöskin vuosikymmenten saatossa hyväksi todettuja ratkaisuja ja se onkin ollut alan standardina

jo pitkään. Se todennetusti toimii varsin hyvin myös todella isojen datamäärien käsittelyssä. SQL-kieli itsessään on paljon kehittyneempi ja kattavampi kuin suurin osa NoSQL-pohjaisten ratkaisujen kielistä. SQL-kielellä pystyy tekemään hyvinkin monimutkaisia kyselyitä.

SQL-tietokannan käyttäminen mahdollistaa niin kutsutun Object Relational Mapping-teknologian (ORM) käyttämisen PHP:n ja Symfonyn kanssa. ORM tekee tiedon käsittelystä hyvin helppoa ja suorastaan mukavaa. Nimensä mukaisesti ORM kääntää relaatiotietokannan dataa objektimaiseen muotoon ja toisinpäin. Samalla ORM piilottaa suurimman osan tarvittavasta työstä mitä tietokannan suhteen tarvitsee tehdä. Objektien kanssa työskentely on paljon helpompaa ja yksinkertaisempaa ymmärtää. Näin kehitystyössä säästyy aikaa ja vaivaa, minkä voi sitten käyttää johonkin järkevämpään. Symfonyssä käytetään Doctrine-nimistä ORM-kirjastoa.

Eri SQL-tietokanta vaihtoehdoista päädyttiin MariaDB:hen, koska se on avoimen lähdekoodin ratkaisu ja sen käyttäminen on PostgreSQL-kantaa helpompaa. PostgreSQL kärsii myös lukuoperaatioiden suhteellisesta heikkoudesta ja siitä, että sille on vähemmän tukea saatavilla kuin MySQL-tietokannoille (O.S. Tezer, 2014). MariaDB taas ratkaisee monia ongelmia, jotka tekevät itse MySQL-tietokannasta huonon ratkaisun. MySQL:n kehitys on valitettavan pysähtynyttä ja osan mielestä melkein mikä tahansa muu tietokanta vaihtoehto on MySQL-tietokantaa parempi. Koska MariaDB on täysin MySQL yhteensopiva, jolloin sitä on yhtä helppo käyttää ja asentaa kuin MySQL-kanta mutta ollessaan huomattavasti tavallista MySQL-kantaa parempi, niin ratkaisu oli helppo.

4.4.3 Haitat

Yksi isoimmista ongelmista relaatiotietokannan suhteen nousi tarpeesta tallentaa pelin dataa. Relaatiotietokannoissa datan odotetaan olevan tiukasti jäsenneltyä ja määriteltyä, sillä tämä mahdollistaa datan tehokkaan hakemisen, kun rakenne on tiedossa. Pelien kohdalla rakenne riippuu täysin rakentamisessa käytetystä moduulista, joten sellaisenaan tiedon tallentaminen ei ole mahdollista. NoSQL-ratkaisut toimivat yleensä paremmin tällaisen epämääräisemmän datan kanssa, sillä osassa NoSQL-ratkaisuja tallennetaan lähinnä JSON-muotoista dataa. Myös PostgreSQL tukee JSON-muotoista dataa, joten sekin voisi olla parempi ratkaisu pelien datan tallentamiseen.

Kun käytössä on MySQL-pohjainen ratkaisu, käytettäväksi jää muutama ratkaisu. Näistä yleisin on tallentaa JSON-data tekstinä yhteen sarakkeeseen. Ratkaisu on sinällään sangen toimiva, mikäli dataa ainoastaan luetaan. Tällöin kuitenkin menetetään moni SQL:n ominaisuus kuten kyselyiden suorittaminen JSON-dataan. Dataa kuitenkin myös tallennetaan tietokantaan. Datan päivittäminen päivittää koko JSON-tekstin yhdellä kertaa, eikä yksittäisten osien päivittäminen ei ole mahdollista.

SQL-pohjaisten ratkaisujen ongelma on myös tunnetusti ollut horisontaalinen skaalautuvuus. Horisontaalisessa skaalautumisessa järjestelmän suorituskykyä kasvatetaan lisäämällä uusia laitteita tai palvelimia, kun taas vertikaalissa skaalaamisessa parannetaan yhden laitteen tai palvelimen suorituskykyä. (Nati Shalom, 2017) Horisontaalinen skaalautuminen on SQL-tietokannoille vaikeaa, sillä se vaatii datan hajauttamista eri palvelimille, joka taas tekee ACID-operaatioiden varmistamisesta hyvin hankalaa. (Joeri Sebrechts, 2013)

Verrattuna esimerkiksi PostgreSQL-tietokantaan, MariaDB ei toteuta SQL-standardia yhtä kattavasti eikä siinä ole yhtä hyvää tukea JSON-kentille. Toisaalta, käytetty Doctrine ORM-kirjasto ei muutenkaan tue JSON-kenttiä, joten tämä on melko pieni menetys. PostgreSQL pärjää myös paremmin kirjoitusraskaissa operaatioissa, mutta tämän kokoisessa järjestelmässä MariaDB:n suorituskyky on enemmän kuin tarpeeksi riittävä. Mikäli suurin osa järjestelmästä olisi kirjoittamista, niin tilannetta pitäisi miettiä uudestaan.

5 Modulaarisen järjestelmän suunnitteleminen ja kehittäminen

5.1 Tarve modulaarisuudelle

Jo alusta alkaen oli selvää, että järjestelmän tulee olla hyvin universaali eli taipua moniin erilaisiin pelin luonti tarkoituksiin. Mikäli itse järjestelmä rupeaisi rajoittamaan sitä, minkälaisia pelejä sillä voi tehdä, johtaisi se hyvin nopeasti ongelmiin, koska ei ole mitenkään mahdollista vetää universaaleja sääntöjä sille, minkälaisia oppimateriaalipelien pitäisi olla. Tietysti järjestelmän voisi suunnitella siten, että sitä jatkokehitetään aina tarpeen mukaan, mutta tämä aiheuttaa ison liudan ongelmia. Se vaatisi

työhön aina sellaisen tiimin, joka tuntee järjestelmän kattavasti ja se kuormittaisi ylläpitävän tiimin työtaakkaa. Se olisi myös itsessään hyvin työlästä, sillä pelilogiikkaa olisi sidottu ympäri järjestelmää. Viimeiseksi, se kasvattaisi järjestelmän kokoa ajan myötä huomattavasti ja tekisi järjestelmästä aivan valtavan monoliitin.

Tästä syystä jäljelle jäi kaksi vaihtoehtoa. Joko itse editorista tehdään täysin universaali, jolla voi tehdä vähän kaikkea, tai sitten kehitetään järjestelmä, jossa on tiettyjä pelityyppejä ja editori mukautuu aina pelityypin mukaan. Universaalin editorin tekemisessä ei olisi ollut mitään järkeä jo siitäkin syystä, että sellaisen kehittäminen on aivan valtava työ eikä siihen ole resursseja. Tämän lisäksi oman universaalin editorin kehittäminen ei ole millään tavalla järkevää, sillä saatavilla on jo tusinoittain tähän tarkoitukseen kehitettyjä työkaluja, joita on kehitetty vuosia ja joissa on isot kehitystiimit takana. Tämän lisäksi yksi järjestelmän isoimmista vaatimuksista sotii universaalia editoria vastaan. Järjestelmän on tarkoitus olla käytössä tavallisilla opettajilla ilman, että heidän täytyy erikseen opetella pelien kehitystä.

Editori, jossa koko editorin toiminnallisuus pystytään rakentamaan yhden pelimekaniikan tai -tyypin ympärille, mahdollistaa hyvin pitkälle viedyn kustomoinnin. Tällä tavalla käyttöliittymä pystytään pitämään hyvin suoraviivaisena ja selkeänä. Niin editorin kuin itse pelinkin toiminnallisuus määritetään erikseen pelimoduulitiedostossa ja ympäröivä järjestelmän vain mukautuu annetun datan pohjalta. Näin myös järjestelmän laajentaminen on helppoa, sillä moduulien kehitys voidaan tehdä täysin erillään muusta järjestelmästä ja sellaisten ihmisten toimesta, jotka eivät tunne koko järjestelmää läpikotaisin. Heidän täytyy tuntea vain moduulin toiminnan säännöt.

5.2 Modulaarisuuden asettamat haasteet

Modulaarisuuden ja jatkettavan (extensible software) arkkitehtuurin on jo pitkään tiedetty tuovan huomattavia hyötyä sovelluskehitykseen tietyllä saralla. Samalla on myös tunnistettu mallin haittapuolet ja haasteet. Samojen ja myös uusien haasteiden eteen jouduttiin myös tässä projektissa.

Yksi selkeistä haasteista on varmistaa, että ympäröivän järjestelmän logiikka pidetään puhtaana kaikesta logiikasta, joka kuuluu moduulille. Varsinkin tässä projektissa, jossa niin moduulin kuin järjestelmän kehitystä tehdään rinta rinnan, on riski näiden

sotkeutumisesta todellinen. Moduulien rajapinnan tulisi olla mahdollisimman geneerinen ja yksinkertainen. Ympäröivässä järjestelmässä tulee myös pyrkiä geneerisyyteen ja keveyteen. (Noufal Ibrahim, 2015). Useasti kehityksen aikana on joutunut pysähtymään ja miettimään, että kannattaako kyseistä ominaisuutta tehdä ydinjärjestelmään vai toteuttaa se moduulin kautta. Kaikista haastavinta on ollut tunnistaa ne toiminnot, jotka esimerkiksi pelin muokkauksessa ovat geneerisiä ja yleisesti käytettävissä, ja ne jotka koskevat vain tiettyä tai tietyn tyyppisiä pelejä.

Moduulien mukaan tuonti on myös aiheuttanut järjestelmässä niin sanottua jojo-ongelmaa. Jojo-ongelmasta puhutaan, kun järjestelmän periytymiskaava (inheritance graph) on niin pitkä ja monimutkainen, että kehittäjä joutuu hyppimään usean eri luokan välillä ymmärtääkseen logiikan kulkua (Sharique Khan, 2013). Moduulien kanssa kehittäjä joutuu ensin seuraamaan koodia ydinjärjestelmästä, kunnes polku yhtäkkiä katkeaa ulkoiseen funktioon ja sen jälkeen etsimään tämän funktion moduulista ja sitten seuraamaan sitä moduulissa. Tämän takia ydinjärjestelmässä on pyritty minimoimaan edellä mainittu hyppiminen ja tekemään siirtymät ulkoisiin funktioihin selkeäksi. Siirtymä ydinjärjestelmästä moduuliin aiheuttaa myös niin kutsuttua taikuutta, eli toiminnallisuuden, jonka logiikka on piilotettu ja jota voi olla vaikea ymmärtää. (Pete Kirkham, 2009). Tämä on valitettavaa, mutta luontainen seuraus modulaarisesta järjestelmästä.

Haasteena moduulien kanssa on myös varmistaa järjestelmän turvallisuus. Moduulit sisältävät koodia, joka ajetaan sellaisenaan joko käyttäjän selaimessa tai jopa palvelimella. Moduulien kehitystä ei ole kuitenkaan koskaan tarkoitettu kaikille avoimeksi leikkikentäksi. Vaikka dokumentaation avulla kuka tahansa voi luoda uuden moduulin, vain admin-tason käyttäjä pystyy hallinnoimaan järjestelmän moduuleja. Näin vastuu ladattavista moduuleista on loppujen lopuksi ylläpidolla.

Viimeiseksi, moduuleista ladattavan datan määrä voi aiheuttaa haasteita. Järjestelmä joutuu lataamaan tämän datan muistiin ja välillä jopa kuljettamaan sitä HTTP-kyselyssä. Tästä aiheutuu teknisiä rajoituksia moduuleille ja kaatumisriski itse järjestelmälle. Riskiä voidaan hallita toteuttamalla tietynlaisia ”batch” (suom. sarja, joukko) -latauksia, joissa tieto ladataan ja siirretään pienemmissä erissä. Tällä hetkellä on kuitenkin katsottu, että moduuleissa määritettävät kokonaisuudet säilyvät

sen verran pieninä, ettei toimenpiteisiin ole ollut tarvetta ryhtyä. Moduulin on tarkoitus esittää yksittäistä pelityyppiä ja mitä isompi ja monimutkaisempi kyseinen pelityyppi on, sitä enemmän se sotii järjestelmän perusideaa vastaan.

5.3 Modulaarisuuden myötä nousseet rajoitteet

Järjestelmässä, jossa kaikki yksittäiseen pelin logiikkaan liittyvä toiminnallisuus on ulkoistettu moduuliin ja jossa ydinjärjestelmä pyrkii olemaan mahdollisimman riisuttu peliä koskevista toiminnoista, voi olla vaikeaa toteuttaa uusia ominaisuuksia pelin luontiin. Jokin tietty toiminto voisi esimerkiksi toimia erittäin hyvin muutamassa eri pelityypissä, mutta moduulin kautta sen toteuttaminen voi olla hyvin vaikeaa ja monimutkaista. Ydinjärjestelmään tällaista toimintoa ei kuitenkaan voi tehdä, jos sitä ei pysty universaalisti käyttämään kaikissa tyypeissä.

Edellisessä luvussa puhuttiin muistin käytöstä. Koska moduulin sisältö joudutaan lataamaan palvelimella muistiin ja muistin määrä on rajallinen, seuraa tästä tietty absoluuttinen rajoitus moduulin koolle. Moduulin koon rajoituksen myötä seuraa rajoitus sille, kuinka monimutkaisia pelityyppejä moduulilla voidaan toteuttaa.

Moduuleille annettavien mahdollisuuksien eli niin kutsutun moduulirajapinnan kehittäminen on eräänlaista jatkuvaa kompromissien tekemistä. Toisaalta, moduuleja ei halua rajoittaa, mutta toisaalta moduulien tekemisen haluaa pitää tarpeeksi yksinkertaisena, jotta moduulien tekeminen on mielekästä. Moduulien kehittämiseen tuo rajoitteita myös se, että ainoastaan ylläpito voi loppujen lopuksi hallita moduuleja. Näin ollen moduuleja tekee vain joko talon sisäinen tiimi tai sitten ulkoinen palkattu tiimi. Mikäli moduulien kehittämistä halutaan avata laajemmassa mittakaavassa, tulee perustaa jonkinlainen myötävaikuttamisjärjestelmä (contribute system), sillä turvallisuussyistä jonkin ylläpitävän tahon on aina hyväksyttävä moduuli ennen sen asettamista järjestelmään. Täysin eristetyn moduulin tekeminen tässä järjestelmässä ei vain ole mahdollista, sillä moduulin on tarkoitus määrittää pelinlogiikka eli koodi sekä operoida pelin datalla.

5.4 Modulaarisuuden toteutus

Modulaarisuus toteutettiin lopulta siten, että moduuli on yksittäinen PHP-tiedosto, joka määrittää PHP-luokan. Tähän ratkaisuun päädyttiin, koska moduulia käyttävä järjestelmä itsessään on tehty PHP:lla, joten moduulin lataaminen ja lukeminen on hyvin vaivatonta. Tässä ratkaisussa on tietysti myös omat haittapuolensa.

Moduuliluokan tulee toteuttaa tietyt funktiot ja määrittää tietyt muuttujat. Moduuli ei kuitenkaan toteuta mitään ennalta määritettyä rajapintaa (interface), sillä itse moduulin kehitys on tarkoitettu tapahtuvaksi järjestelmän ulkopuolella. Taulukossa 2 on lyhykäisyydessään kuvattu moduulin muuttujat ja funktiot.

Taulukko 2. Moduulin osat tarkoituksineen, tiivistetty

Nimi	**Tyyppi**	**Tarkoitus**	**Pakollisuus**
json_structure	Muuttuja	Määrittää pelityypin datarakenteen kokonaisuudessan	Kyllä
build_script	Muuttuja	String-muodossa olevaa JavaScript-koodia joka määrittää koko pelityypin logiikan	Kyllä
game_states	Muuttuja	Taulukko joka kertoo järjestelmälle minkälaisia tilakoneita logiikasta löytyy. Tämä liittyy pelikirjasto Phaserin toimintaan. Tiivistettynä, Phaser vaati tiloille string-muotoiset avaimet	Kyllä
editor_states	Muuttuja	Kerran määritettyä logiikkaa uudelleen käytetään editorissa. Editori ei kuitenkaan tarvitse niitä tiloja joissa peliä pelataan, joten tämä muuttuja määrittää luonnin kannalta keskeiset funktiot.	Kyllä
json_explained	Muuttuja	Tällä taulukolla avataan aiemmin määritetty json_structure siihen muotoon, että siitä voidaan rakentaa lomake	Kyllä
custom_actions	Muuttuja	Normaalisti editoriin rekisteröityvät funktiot kutsuvat ydinjärjestelmän funktiota. Tähän taulukkon voi määrittää omia kustomoituja toiminnallisuuksia	Ei
build_editor	Funktio	Nimensä mukaisesti rakentaa editorin. Tässä funktiossa rekisteröidään toimintoja sivupalkkiin ja lomakkeisiin	Kyllä

Pelin logiikan joutuu kirjoittamaan moduuliin string-muodossa, sillä mitään muuta järkevää tapaa toteuttaa JavaScript-koodia PHP-tiedostossa ei oikeastaan ollut. Tämä muuttuja on yksi niistä, joka joudutaan lataamaan sellaisenaan muistiin ja kuljettamaan HTTP-kutsussa selaimelle. Tämä JavaScript-koodi määrittää käytännössä koko pelin toiminnan eli esimerkiksi pelin luomisen, logiikkaan, pisteiden keräämisen ja funktioiden ja tapahtumien määrittelyn. Ainoa asia joka tehdään pelin suhteen moduulin puolesta, on tiedostojen lataaminen. Tämä niin kutsuttu esilatausvaihe (preload) on käytännössä aina samanlainen ja se toteutetaan aina user_assets-nimisen taulukon arvoista. Kaikessa yksinkertaisuudessaan esilatausvaihe lataa tiedostoja annetuista URI-osoitteista ja tallentaa ne välimuistiin Phaser-kirjaston sisuksiin tietyn ennalta määritetyn avaimen taakse. Tässä järjestelmässä avaimena käytetään pelin dataan tallennettuja tiedostojen nimiä. Automaattisen esilatauksen käyttäminen ei tietenkään ole pakollista, vaan halutessaan user_assets-kohdan voi jättää pois rakenteesta ja määrittää logiikassa oman esilatausvaiheen.

Pelin ja editorin tilojen määrittäminen moduulissa on pakollista, sillä vaikka tilan vaihto tapahtuu manuaalisesti moduulissa määritetyssä koodissa, tiloihin viitataan erikseen määritetyillä avaimilla. Automaattinen esilataus tallentaa myös nämä tilat. Itse pelin koodi voisi vielä pärjätä ilman tällaista toteutustapaa, mutta editorin puolella tällainen taulukko on pakollinen. Editorin tulee tietää missä vaiheessa peli muuttuu tilaan, joka ei kuulu editoriin. Tällaisen tilan tunnistettuaan, editori siirtyy omaan sisäiseen tilaansa, jossa pyörii kaikki editorin toiminnot, kuten raahaus ja koon muuttaminen. Tila on ikään kuin kohtaus (scene), tai oikeastaan kohtaukset ovat tietynlaisia tiloja. Tila ja tilakone (finite state machine) on malli, jossa yhdessä kokonaisuudessa, esimerkiksi luokassa, on aloitustila, sisääntulofunktio ja siirtymäfunktio (Paul E. Black, 2016). Aloitustila määrittää miltä tila näyttää alussa, sisääntulofunktion kautta tilalle voi syöttää arvoja ja siirtymäfunktio siirtää koneen seuraavaan tilaan toteuttaen jotain logiikkaa samalla ja näin määrittäen seuraavan tilan arvot (Paul E. Black, 2004). Esimerkkinä, päävalikko voi olla yksi tila, josta siirrytään sitten latausruudun kautta itse peliin. Tässä olikin jo kolme tilaa.

Editorissa käytetään paljon lomakkeita datan muokkaamiseen. Lomakkeita varten tarvitaan kuitenkin lisätietoa rakenteen kentistä ja ihmisluettavat nimet eri arvoille. Tätä dataa ei kannata laittaa itse JSON-rakenteeseen, koska se on tarkoitettu vain pelin kannalta tärkeälle datalle. Tästä syystä moduulissa on json_explained-muuttuja, johon voi määrittää miten lomake rakentuu millekkin kentälle. Jättämällä tiettyjä arvoja pois selityksestä, voi rajoittaa lomakkeella olevia arvoa. Tätä käytetään paljon esimerkiksi sijainnin kanssa, sillä sijaintia voi vaihtaa raahamalla eikä sitä tarvita lomakkeella. Niiden kenttien kohdalla, jossa määrityksiä haluaa antaa, niin ainoa pakollinen arvo on label eli nimike. Tämä kenttä määrittää lomake-elementin otsikon. Mikäli kyseessä on jokin muu kuin normaali tekstikenttä, niin tämän voi määrittää type-kohtaan. Riippuen kentän tyypistä, on sitten muita asetuksia, kuten monivalinnan vaihtoehdot.

Ydinjärjestelmä tarjoaa tiettyjä geneerisiä funktiota, joista voi rakentaa editoriin haluamansa toiminnot. Toiminnot tulevat painikkeina editorin sivupalkkiin. On kuitenkin usein todennäköistä, että halutaan luoda myös kustomisoituja toimintoja, sillä ydinjärjestelmään ei pysty määrittämään mitään pelityyppikohtaista logiikkaa. Tätä varten moduulissa voi määrittää kustomisoituja funktioita, joihin voi sitten linkittää toimintoja build_editor-funktiossa. Kustomisoitavat funktiot ovat yleensä JavaScript-pohjaisia, joten toiminnolle määritetään aiemmin määritetty JavaScript-funktio, joka ajetaan kun linkattua toimintoa painetaan.

Editori pitää tiettyjä muuttujia globaaleina, että moduulin koodi pystyy käsittelemään tätä dataa. Näin moduulin koodi pääsee esimerkiksi käsiksi tehtävän määrittävään dataan. Editorissa on myös tiettyjä avustefunktioita, joiden avulla moduulista voi esimerkiksi asettaa aputekstejä näkyviin haluamillaan sisällöillä.

Moduulin ainoa funktio on se paikka, jossa määritetään mitä toimintoja editoriin halutaan käyttäjälle. Tämä funktio on pakko löytyä moduulista, sillä sitä kutsutaan editoria ladatessa. Funktio saa parametrinaan konteksti-muuttujan, joka on käytännössä viittaus moduulijärjestelmän sisäiseen pääluokkaan. Rekisteröidäkseen toimintoja, tulee build_editor-funktiosta kutsua pääluokan muita funktioita ja antaa niille parametreinä toimintojen tiedot. Nämä funktiot näkyvät taulukossa 3.

Taulukko 3. Moduulisysteemin pääluokan funktiot, joita voi kutsua moduulissa.

Pääluokan funktio	**Toiminto**
register_sidebar_actions	Sivupalkin toiminnot, jotka ovat jatkuvasti näkyvissä.
register_element_sidebar_actions	Sivupalkin toiminnot, jotka tulevat näkyviin kun määritetty elementti on valittuna.
register_settings_form_actions	Tällä voi määrittää ylimääräisiä toimintoja jokaisella pelillä olevaan ”yleiset asetukset” lomakkeeseen.
register_victory_form_actions	Sama kuin edellä, mutta voittoruudun määrityslomakkeelle.

Editoria ladatessa tarkistetaan ensiksi tietokannasta pelin tiedoista mitä moduulia peli käyttää. Moduuleilla on tietokannassa oma taulu. Tästä taulusta saadaan käytetyn moduulin tiedostonimi ja luokan nimi. Tämän jälkeen moduulin latausjärjestelmä lataa ennalta määritetystä nimitilasta (namespace) moduulin nimisen luokan. Tämän jälkeen pääluokka ajaa tiettyjä koostajafunktiota, kuten lomakkeidenrakentajan ja lopulta kutsuu moduulin build_editor-funktiota. Pääluokassa on myös funktioita datan ulosottamiselle valmiissa formaatissa.

Pelin käynnistyessä käytetään pitkälti samaa tapaa. Itse pelin JavaScript-koodi laitetaan sellaisenaan sivulle ja pelin data ja pelin tilat tallennetaan globaaleihin muuttujiin. Init-funktio käynnistää esilatausfunktion ja esilatausfunktio kutsuu valmistuttuaan tilataulukon ensimmäistä tilaa. Tämän jälkeen peli siirtyy moduulin koodiin.

5.5 Toteutustavan haittapuolet

Moduulien toteuttaminen PHP-luokkana aiheuttaa oikeastaan kolme selkeää ongelmaa. Ensimmäinen ongelma on niin sanottu vendor lock (suom. toimittajalukitus). Vaikka on epätodennäköistä, että järjestelmä kirjoitettaisiin myöhemmin uusiksi jollain toisella kielellä, jos se tehdään, niin jokainen moduuli pitäisi uusia samalla. Tämän lisäksi PHP-kielen valinta esimerkiksi XML:n sijaan rajaa omalla tavallaan moduulien universaaliutta. Moduulin tekijän täytyy osata yhtä tiettyä kieltä jonkin universaalin kielen sijaan.

Toinen ongelma syntyy moduulin muokkaamisessa. Itse moduulin sisältöä ei ole tarkoitus muokata, vaan kehitys on tarkoitus tehdä joko ulkopuolella tai käyttää

moduulin muokkaustyökalua. Tiedostoa itsessään ei kuitenkaan voi muokata kuin tekstitiedostona eikä PHP:lla pysty lataamaan PHP:ta tekstitiedostona siten, että sen voisi muuttaa joksikin järkeväksi tietorakenteeksi. Esimerkiksi XML-muotoisen tiedoston pystyisi ottamaan muistiin ja parsimaan osa-alueisiin. Yksittäisen XML-noodin muokkaminen olisi helppoa ja parsitun version pystyisi siitä suoraan kirjoittamaan tiedostoon. Nyt kun kyseessä on PHP-tiedosto, josta muokataan työkalujen avulla yksittäisiä osia, täytyy nämä osat eritellä tekstistä ja ajaa raskaita tekstin muuntofunktioita. Esimerkiksi json_structure kohta päivitetään siten, että tekstistä etsitään $this->json_structure ja sen jälkeen sitä seuraava ;-merkki. Näiden kahden sijainnin väliltä sitten korvataan kaikki uudella koodilla. Luonnollisesti monen tällaisen operaation ajaminen on skaalautuvuuden kannalta katastrofi, mutta onneksi palvelussa moduulien muokkaustyökalua käyttävien ihmisten määrä voidaan parhaassakin tapauksessa todennäköisesti laskea sadoissa. Mikäli skaalautuvuus alkaa olla ongelma, niin yksi vaihtoehto on kehittää muokkaustoimenpiteille niin kutsuttu jonosysteemi (delayed queue), jossa raskaat tehtävät irrotetaan itse HTTP-kutsusta ja ajetaan myöhemmin taustalla.

Kolmas ongelma on JavaScript-koodin vaatimus moduulissa. Koska JavaScript-koodin kirjoittaminen string-muotoiseen muuttujaan on yksi huonoimmista tavoista tehdä koodia, on moduulin tekijä pakotettu käyttämään joko moduulin muokkaustyökalua tai kehittämään JavaScript-koodi moduulin ulkopuolella. Tästä aiheutuu myös mahdollisten ongelmien löytämisen ja koodin seuraamisen vaikeutuminen, epäformatoitu koodi on vaikeaselkoista.

6 Tulokset ja järjestelmäkuvaus

6.1 Yleiskuvaus

6.1.1 Kaksi eri osaa, yksi järjestelmä

Järjestelmässä on kaksi hyvin selkeää itsenäistä kokonaisuutta. Toinen on pelipuoli, jossa pelaajat pelaavat heille saatavilla olevia pelikokonaisuuksia ja toinen on hallintapuoli, jossa kasataan ja muokataan pelikokonaisuuksia, luodaan uusia pelejä, hallinnoidaan voittoruutuja, muokataan pelimoduuleja, hallitaan pelaajia ja niin edelleen.

Samaan aikaan nämä kokonaisuudet ovat kuitenkin kiinteästi sidoksissa toisiinsa. Pelipuolella ei voisi tehdä mitään ilman sille pelejä tuottavaa hallintapuolta, eikä hallintapuolella olisi mitään funktiota ilman sen tuloksia käyttävää pelipuolta.

Kun eri kokonaisuudet suhtautetaan käyttökohteeseen, niin pelipuoli on suunnattu Valterin oppilaille, kun taas hallintapuoli on opettajille. Pelipuolella tuleekin siis keskittyä erityisesti esteettömyyteen ja hallintapuolella taas pyrkiä intuitiiviseen ja helppokäyttöiseen hallintapaneeliin.

Teknisesti nämä kaksi selkeästi erilaista kokonaisuutta tarjotaan kuitenkin samasta projektista ja ne on eritelty projektissa sisäisesti omiin reitteihinsä ja kansiorakenteisiin. Tämä eroaa toisesta sangen yleisestä tavasta toteuttaa eri kokonaisuuksista koostuva järjestelmä. On yleistä, että selkeät kokonaisuudet koostavat yhden itsenäisen projektin ja kokonaisuus muodostuu, kun nämä eri osa-alueet liitetään toisiinsa joko erinäisillä rajapinnoilla tai sitten eri osa-alueita hyödynnetään komponentteina. Muun muassa viime vuosina suosiota saanut konttiteknologia on vienyt tätä kehityssuuntaa entistä pidemmälle.

Tässä järjestelmässä on kuitenkin päädytty enemmän monoliittimaiseen ratkaisuun sangen yksinkertaisista syistä. Ensimmäinen syy on helppous ja nopeus. Projektia aloittaessa aikaa oli rajallinen määrä ja tekijöillä rajallinen osaaminen. Toinen syy on yksinkertaistettu pystytys. Osa-alueiden ollessa samassa projektissa, kun ympäristön saa kerran pystyyn, niin molemmat kokonaisuudet toimivat. Näiden kokonaisuuksien eriyttäminen voi olla tulevaisuudessa ihan varteen otettava vaihtoehto, mutta nykyisellään ne toimivat hyvin myös monoliitissa.

6.1.2 Pelit ja tehtävät

Termillisesti ehkä yksi projektin hämäävimmistä asioista on ollut pelien ja tehtävien ero. Tehtävä tarkoittaa yhtä yksittäistä pelattavaa asiaa. Hallintapuolella olevalla editorilla luodaan näitä tehtäviä. Peli taas on näistä tehtävistä koostuva kokonaisuus, tehtäväpaketti. Peli itsessään ei siis ole mitään pelattavaa, vaan sen sisällä olevat tehtävät ovat niitän mitä varsinaisesti pelataan.

Luonnollisesti olisi ehkä intuitiivisempaa kutsua tehtäviä peleiksi ja pelejä pelipaketeiksi. Projektia aloittaessa termistöstä käytiin paljonkin keskustelua ja lopputuloksena asiakas halusi nimetä nämä asiat tällä tavalla. Toisaalta, nimissä on se järkevyys, että normaalistikin pelit yleensä koostuvat kentistä. Yhden kentän taas voi ymmärtää yhtenä tehtävänä.

Ideana on ollut, että pelaaja voi yksittäisten tehtävien lisäksi valita pelin pelattavakseen. Tällöin yhden tehtävän suoritettuaan pelaaja siirtyy automaattisesti seuraavaan tehtävään. Tämä mahdollistaa oppimisen kannalta luontevien kokonaisuuksien muodostamisen ja antaa pelaajalle mukavan progression. Yhtä tehtävää voi myös käyttää monessa pelissä ja pelin sisällä tehtävien järjestystä voi säätää.

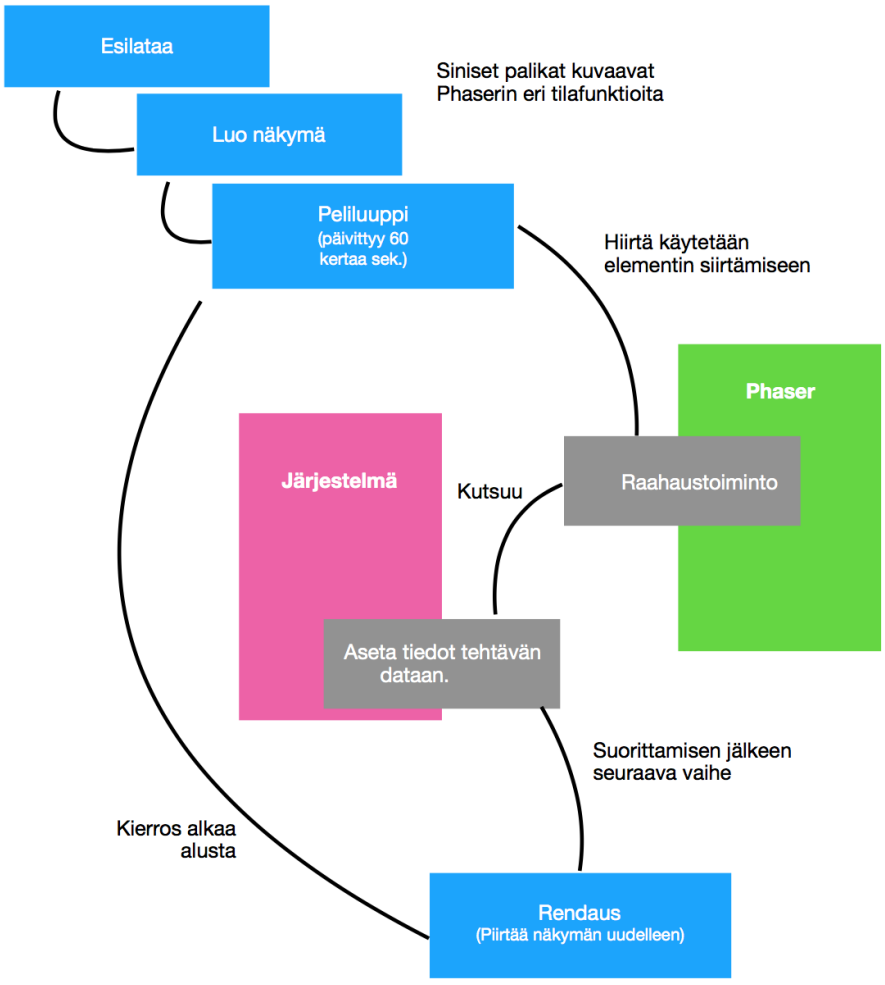
6.2 Editori

Editori on hallintapuolen suurin kokonaisuus. Se on paikka, jossa tehtäviä rakennetaan ja muokataan. Editorisivun perustoiminnot ovat aina samat, mutta editointi itsessään on erilaista eri moduulien välillä.

Kaikki tehtävän tiedot ovat tallennettuna JSON-muotoisena merkkijonona tietokannassa. Editoriin mennessä palvelin lataa tietokannasta tehtävän tiedot annetun pelitunnisteen avulla. Näistä tiedoista löytyvät sekä tehtävän tiedot, että käytetty moduuli. Jokaisesta moduulista on tietokannassa tieto. Näin voidaan ladata tehtävälle tarvittava moduuli. Moduulien toimintaa varten on järjestelmään tehty moduulien käsittelijä. Tämä palvelu lataa ennalta määritellystä kansiosta muistiin moduulin, joka on toteutettu PHP-luokkana. Tämän jälkeen käsittelijä lukee moduulin tiedot ja kasaa niistä käytettävän datapaketin. Käsittelijä esimerkiksi rakentaa erinäiset lomakkeet pelin tietojen muokkauksia varten. Kun palvelin siirtyy pohjatiedostoon luomaan näytettävää HTML-rakennetta, sillä on tiedossa tehtävän tiedot sekä tarvittavat tiedot moduulista, jotta tehtävän dataa voidaan käyttää. Käytännössä tehtävän data ja moduulissa oleva luontiskripti asetetaan JavaScript-muuttujaan ja skriptitagiin.

Kun palvelinpuoli on valmis, käynnistyy JavaScript-koodi viimeistelemään editorin. Ensimmäiseksi se käynnistää sivulle tuodun luontiskriptin ja odottaa, että luontiskripti on siirtynyt sellaiseen vaiheeseen, joka ei enää kuuluu editoriin. Tämän tiedon JavaScript-koodi on saanut moduulista.

Editori itsessään on yksi Phaser-peli, kuten ovat myös pelattavat pelit. Näin samaa koodia, joka rakentaa tehtävän pelipuolella, voidaan käyttää tehtävän alkutilan rakentamiseen editorissa. Phaserissa, kuten muissakin pelimoottoreissa, on tuki tiloille tai näkymille (states / scenes). Näitä tiloja seuraamalla ympäröivä editorijärjestelmä katkaisee luontiskriptin ajamisen ja siirtyy omaan tilaansa, jossa se muun muassa luo jokaiselle elementille muokkaimet ja tarvittavat tapahtumakuuntelijat. Käytännössä tehtävän muokkaaminen onkin eräänlaista pelin pelaamista. Tässä pelissä elementtien sijaintia, kokoa ja tietoja muokataan halutunlaiseksi. Kuviossa 13 avataan tarkemmin esimerkkinä raahaustoiminnon toimintaa editorissa. Raahaus on Phaserin sisäinen toiminto, johon on kiinnitetty järjestelmän omia funktioita. Phaser huolehtii elementin siirtämisestä ja järjestelmä uuden tiedon tallentamisesta sekä muiden elementtien synkronoinnista raahattavan elementin kanssa.



Kuvio 13. Läpileikkaus raahauksen toiminnasta

Itse editointi tapahtuu joko lomakkeiden avulla, käyttämällä editorin oikealla olevan sivupalkin toimintoja tai hyödyntämällä muokkaimia. Kuviossa 14 on näkyvissä sinisellä ympyröitynä itse editori, joka on siis Phaser-peli. Keltaisella ympyröitynä näkyy tehtävän elementtejä, jotka ovat tässä tapauksessa kortteja. Näistä vasemman puoleisella elementillä ovat muokkaimet aktivoituna. Muokkaimet kertovat valitun elementin ja niistä voi muuttaa kortin kokoa ja kulmaa. Siirtäminen tapahtuu korttia raahaamalla. Muokkauslomaketta kuvassa ei näy, mutta sen saa näkyviin oikea klikkaamalla elementtiä ja lomake avautuu modaalina editorin päälle.

Kuvio 14. Editorinäkymä alueineen

Kuviossa on punaisella ympäröity editorin sivupalkki, johon tulevat kaikki pelityypille ominaiset toiminnot. Kaikki nämä toiminnot on määritetty tehtävän määrittävässä moduulissa. Vihreät toiminnot ovat elementtikohtaisia eli ne näkyvät vain, kun tietyn tyyppinen elementti on valittuna. Siniset taas ovat jatkuvasti näkyvissä olevia toimintoja.

Kuviossa vihreällä värillä on merkittynä muokkausnäkymän eri välilehdet. Nämä pysyvät samanlaisina kaikissa pelityypeissä.

Eri välilehtiä ovat:

- Editori, itse editointinäkymä
- Ominaisuudet, tehtävän tietyt ominaisuudet, jotka eivät riipu pelityypistä
- Ohjeet, pelaajalle näkyvät ohjeet.
- Muistiinpanot, pelintekijä voi kirjoittaa tälle tehtävälle muistiinpanoja itselleen tai muille muokkaajille.

Vaaleanpunainen alue pitää sisällään peruspäätoiminnot: tallenna, poista ja esikatsele. Tallentaessa muistissa pidetty ja editoinnissa muokattu tehtävän data lähetetään palvelimelle ja tallennetaan sellaisenaan tietokantaan. Esikatselu luo väliaikaisen pelinäkymän, jossa käytetään sen hetkistä tehtävän dataa. Esikatselu käyttäytyy täsmälleen samalla tavalla kuin pelin käynnistäminen pelipuolellakin.

6.3 Modulaarisuus

Jokaisen järjestelmällä tehtävän ja pelattavan tehtävän toiminnallisuus on määritelty erillisessä moduulitiedostossa. Nämä moduulitiedostot ovat periaatteessa ja hieman yksinkertaistettuna järjestelmän ydin ja ympäröivä järjestelmä niiden ympärille rakennettu kehys.

Jokaisen moduulin tulee määrittää seuraavat asiat:

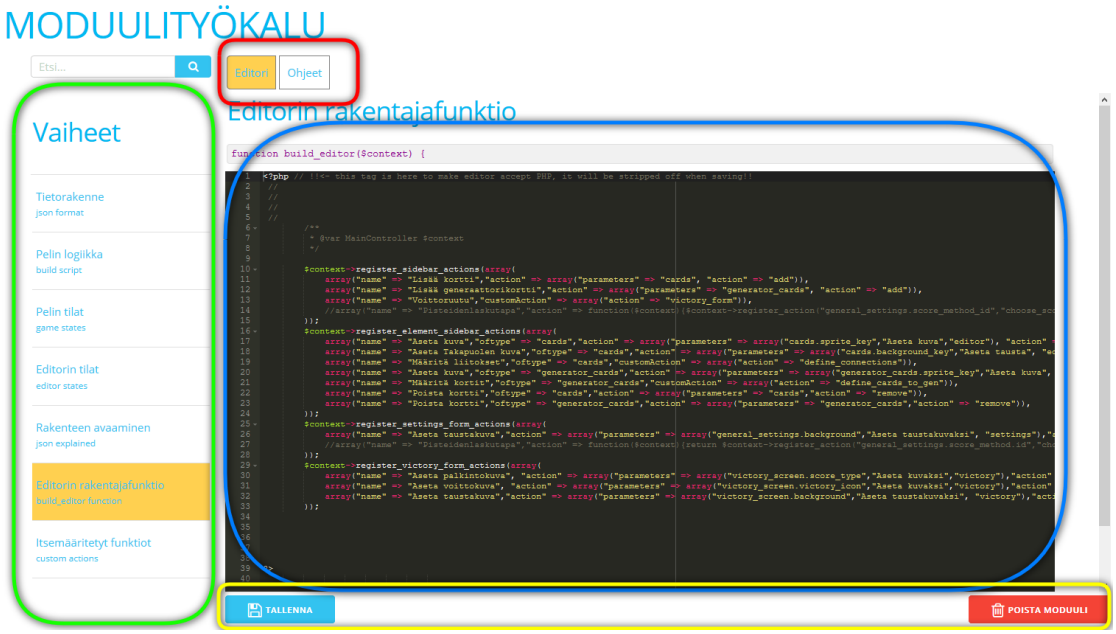
- Tietorakenne ja oletusarvot
- Tehtävän logiikka
- Tehtävän tilat
- Mitä näistä tiloista tarvitaan editorissa
- Mitä funktioita editoriin tuodaan (oletusten lisäksi)
- ”Selitys” tietorakenteelle, eli tavata tietorakenne siten, että sille voi luoda lomakkeen.
- Määrittää käytettävät pisteenlaskutavat
- määrittää voittoruutu

Tämän lisäksi moduuleissa voidaan määrittää täysin kustomisoituja toimintoja editoriin käyttäen JavaScript-kieltä.

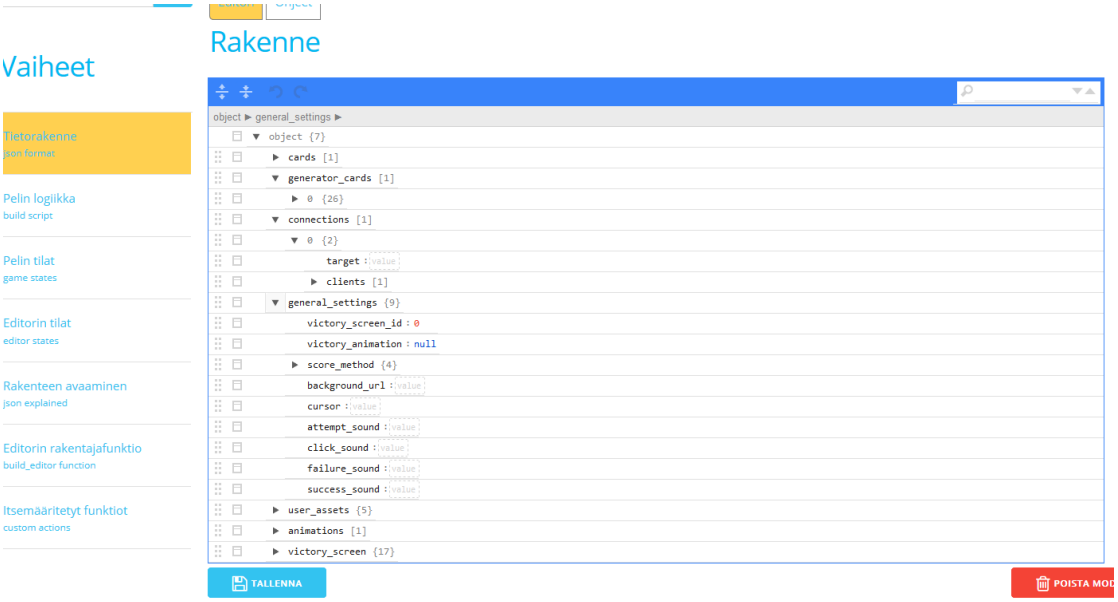
Editori itsessään luo työkalun tehtävän luomiselle siinä formaatissa, kuin moduuli on sen määrittänyt. Tehtävää editoidessa käytetään moduulin määrittämiä työkaluja ja editorista tallennettava data tallennetaan tietokantaan JSON-muotoisena siinä muo-

dossa, kuin se on moduulissa määritetty. Tehtävää pelatessa tallennettu tieto ladataan tietokannasta ja siirretään moduulissa määritetylle pelin logiikkakoodille. Tehtävää luodessa tulee valita, että minkä moduulin päälle tehtävä rakennetaan.

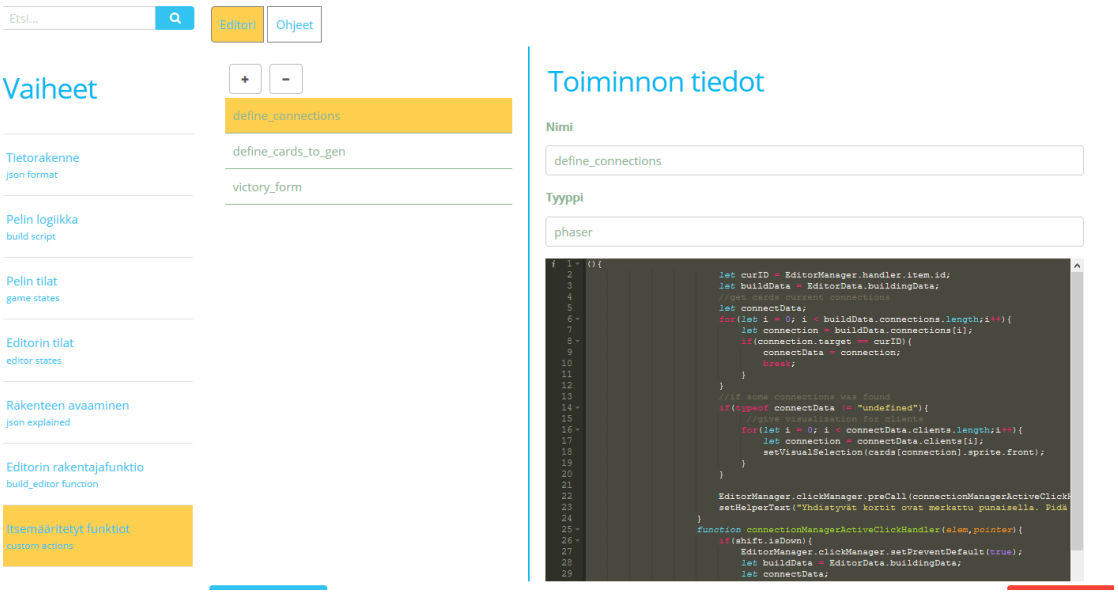
Järjestelmässä olevia moduuleja pääsee katsomaan ja muokkaamaan moduulityökalussa. Moduulityökalun avulla järjestelmän tekniset ylläpitäjät voivat helposti hallita moduuleja ilman SSH- ja FTP-yhteyksiä. Työkalulla voi avata yksittäisen moduulin ja muokata sitä määritettyjen toimintojen avulla. Näin saadaan monimutkaiselle moduulitiedostolle selkeämpi rakenne ja parempi muokattavuus. Tämä ratkaisee tiettyjä ongelmia, kuten luvussa 5.5 todettu epäformatoidun koodin hankaluus. Kuviossa 15 näkyy yksittäisen moduulin muokkausnäkymä toimintoineen. Vasemmalla on moduulin eri osiot, punaisella on ympyröity yhden osan välilehdet, keltaisella on ympyröity päätoiminnot ja sinillä alueella näkyy päätyökalu. Päätyökalu voi olla esimerkiksi koodieditori, JSON-editori kuten kuviossa 16, tai jokin muu, kuten kuviossa 17 nähtävä kustomoitujen toimintojen työkalu. Kuvat ovat näkyvissä suurempikokoisina opinnäytetyön liitteinä 3, 4 ja 5.



Kuvio 15. Moduulin muokkaustyökalu



Kuvio 16. JSON-editori moduulityökalussa



Kuvio 17. Kustomoitujen funktioiden työkalu moduulityökalussa

6.4 Pelien hallinta

Peli on tehtävistä koostuva kokonaisuus. Pelin avulla luodaan oppimiselle merkittäviä kokonaisuuksia ja helpotetaan pelaamista, kun tehtäviä ei tarvitse erikseen vaihdella.

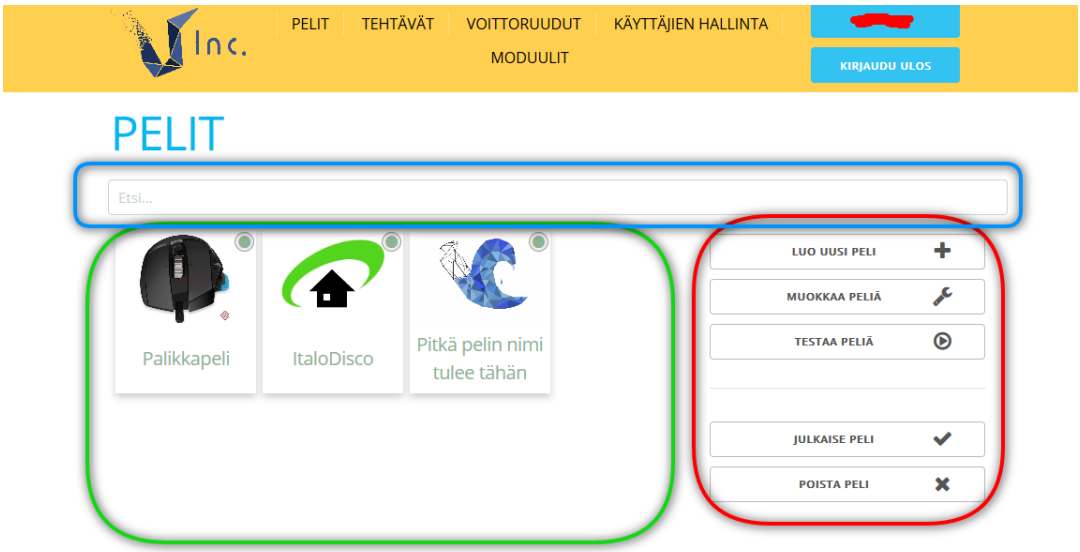
Pelin itsensä hallinta

Kaikissa listaukseen pohjautuvissa näkymissä on hyödynnetty samaa mallia. Vasemmalta suurimman osan ottaa listaus, jossa listattavat asiat on esitetty korttityylillä. Kuviossa 18 näkyy tämä alue merkattuna vihreällä. Kortissa näkyvä pieni vihreä pallo kertoo, ettei peliä ole julkaistu. Julkaisemattomia pelejä ei näytetä pelaajille. Listausta päivitetään palvelimelta aina tietyin aikavälein.

Kuviossa punaisella rajattuna näkyy toimintojen sivupalkki. Myös tämä on samanlainen kaikissa muissa listausnäkymissä. Pelien tapauksessa sivupalkista löytyy seuraavat viisi toimintoa:

- Uuden pelin luonti. Siirtyy lomakkeelle, jossa kysytään pelin perustiedot.
- Valitun pelin muokkaus. Siirtyy pelin sisäiseen muokkausnäkymään.
- Testaa peliä. Siirtyy pelipuolelle.
- Julkaise peli (tai epäjulkaise). Julkaisee tai epäjulkaisee valitun pelin.
- Poista peli. Poistaa valitun pelin, mutta ei tehtäviä.

Sininen alue kuvassa on hakupalkki. Hakupalkki on osa listauksen React-komponenttia ja se suorittaa käyttäjän kirjoittaessa välittömästi haun JavaScriptillä listauksessa olevaa pelilistaa vasten.

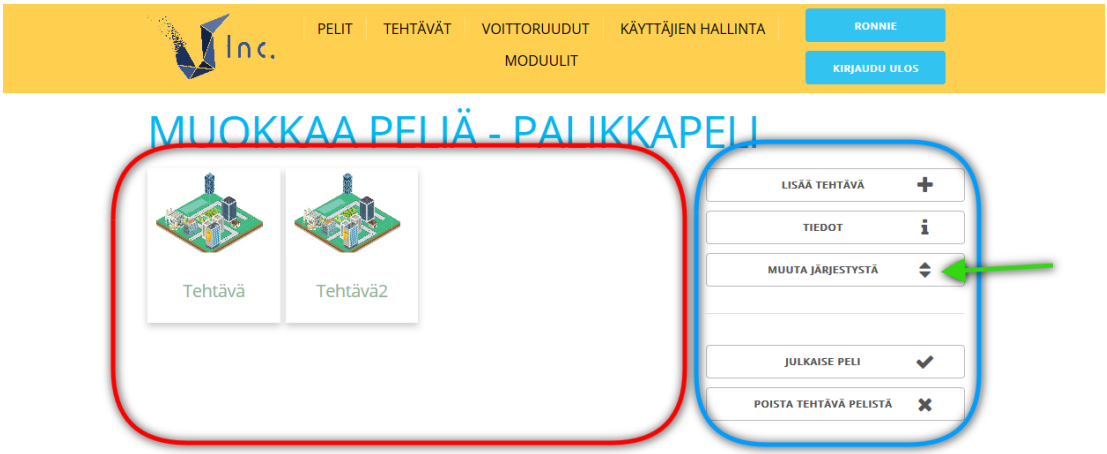


Kuvio 18. Pelien hallinnan päänäkymä

Pelin tehtävien hallinnointi

Kun käyttäjä painaa edellisessä pelien listausnäkymästä ”muokkaa peliä”-painiketta, siirtyy hän pelin sisäiseen hallintaan. Tässä näkymässä listataan pelin sisäiset tehtävät. Kuviossa 19 on näkyvissä punaisella samanlainen listauskomponentti kuin pelien listauksessakin. Sinisellä näkyy tämän näkymän toimintopalkki. Toimintoina ovat seuraavat:

- Lisää tehtävä. Avaa modaalin järjestelmässä olevista tehtävistä.
- Tiedot. Näyttää pelin tiedot.
- Muokkaa järjestystä. Vihreä nuoli, toiminto avattu kuvan alapuolella
- Julkaise peli. Sama kuin edellisessä näkymässä, julkaisee tämän pelin
- Poista peli. Sama kuin edellisessä näkymässä.



Kuvio 19. Pelin sisäinen hallintasivu

Painamalla ”muokkaa järjestystä” käyttäjä siirtyy järjestysnäkymään. Tässä välissä ei tapahdu HTTP-kyselyä, vaan siirtyminen tapahtuu React-komponentin sisällä. Järjestysnäkymässä käyttäjä voi siirtämällä järjestää tehtävät uuteen järjestykseen.

Kuviossa 20 on näkyvissä punaisella tehtävien listaus järjestysnäkymässä. Sinisellä on näkyvissä tehtäväpalkki. Tehtäväpalkissa tapahtuu järjestysnäkymään siirryttäessä sellainen muutos, että ”muuta järjestys”-painike katoaa ja tilalle tulee kolme uutta toimintoa. Ensimmäinen näistä tallentaa uuden järjestyksen ja palaa alkuperäiseen näkymään. Tallennus tapahtuu suorittamalla AJAX-kutsu järjestelmän rajapintaan.

Toinen toiminto palauttaa järjestyksen, joka oli aloittaessa, se on siis käytännössä resetointipainike. Viimeinen painike palaa alkuperäiseen näkymään tallentamatta muutoksia eli vanha järjestys jää voimaan.

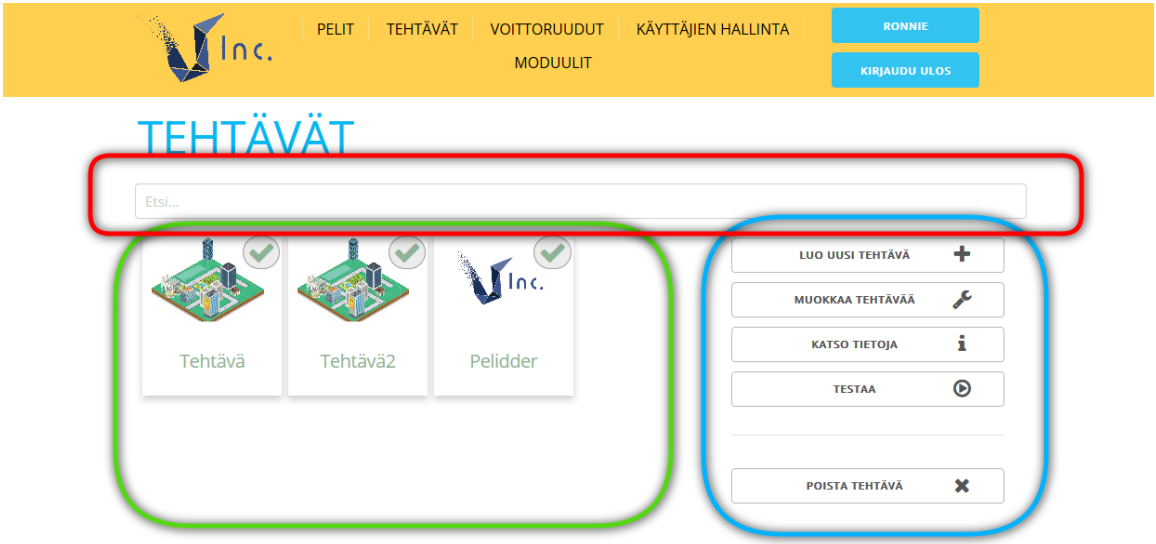
Kuvio 20. Järjestysnäkymä

Teknisesti järjestystoiminto on toteutettu hyödyntämällä Sortable.js-kirjastoa yhdistämällä se Reactin kanssa. Tästä aiheutui tiettyjä vaikeuksia, sillä käytetty kirjasto pohjautuu DOM-puun manipulointiin ja React taas kirjoittaa DOM-puun tietyiltä osin uusiksi komponentin tilan päivittyessä. Lopuksi ongelmat saatiin kuitenkin ratkaistua siten, että Sortable saa vapaasti muokata DOM-puuta raahauksen ajan ja raahauksen loputtua Sortablelle asetettu tapahtumafunktio päivittää React-komponentin sisäisen tilan vastaamaan sitä mikä raahaustoiminnosta seurasi. Tilan päivitys saa komponentin päivittymään, mikä ylikirjoittaa sen hetkisen DOM-rakenteen listan osalta ja näin ollen poistaa Sortablen sisäiset tapahtuman kuuntelijat. Tästä johtuen Reactin päivittymiseen on kiinnitetty tapahtuman kuuntelija, joka käynnistää Sortablen uudestaan päivityksen jälkeen.

6.5 Tehtävien hallinta

Tehtävien hallintaa käytetään yksittäisten pelien luomiseen ja hallinnoimiseen. Tehtävien hallinta eroaa pelin sisäisten tehtävien hallinnasta siten, että jälkimmäisessä näkymässä voi tehtäviä muokata vain suhteessa peliin eli esimerkiksi poistaa tehtävä kyseisestä pelistä. Tehtävien hallinnassa sen sijaan hallitaan tehtäviä itsessään eli tässä näkymässä voi muun muassa poistaa tehtävän kokonaan ja tällöin kyseinen tehtävä poistuu myös kaikista peleistä. Tässä näkymässä voi myös luoda uuden tehtävän, kun taas pelin hallinnassa voi vain lisätä olemassa olevia tehtäviä.

Rakenteeltaan tämäkin näkymä noudattaa samaa asetelmaa kuin pelin hallinnan näkymät. Tehtävät listataan kortteina vasemmalla ja toiminnot tulevat oikealle sivupalkkiin. Hakupalkilla voi hakea tiettyjä pelejä. Kuviossa 21 näkyy korteissa oikeassa yläkulmassa oleva julkaisuindikaattori, joka kertoo sen, että onko tehtävä asetettuna yhteenkään peliin.

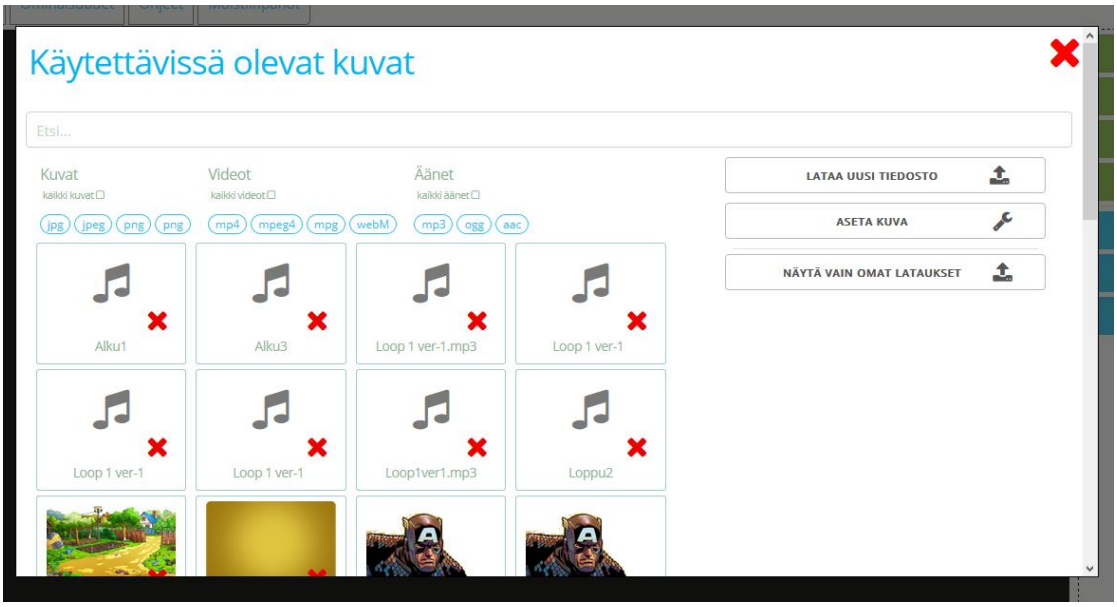


Kuvio 21. Tehtävien hallinnan näkymä

6.6 Mediakirjasto

Jo aikaisessa vaiheessa oli tiedossa, että järjestelmä tulisi tarvitsemaan mediakirjaston. Alkuperäinen tarve mediakirjastolle nousi, kun editorissa piti pystyä valitsemaan tiedostoja elementteihin. Koska samanlaisesta kirjastosta olisi varmasti hyötyä myös muualla, niin kirjastosta tehtiin itsenäinen komponentti. Koska kirjastossa on paljon reaktiivisuutta ja dynaamisuutta, niin se päätettiin toteuttaa React-komponenttina.

Kuviossa 22 näkyy mediakirjaston päänäkymä. Kirjastossa on hakupalkki ja tiedostopäätepohjainen suodatin. Tiedostot listataan korttityylillä ja muista näkymistä kopioituna, toiminnot ovat oikeassa reunassa. Ääni- ja videotiedostojen kohdalla kyseinen media toistetaan, kun hiiren vie kortin päälle.



Kuvio 22. Mediakirjasto

Mediakirjastoa käytetään yksinkertaisesti sen CSS-näkyvyystyyliä muokkaamalla. Kirjastolle voi asettaa ulkopuolelta asetuspainikkeen tekstin, valinnasta kutsuttavan funktion ja halutessaan pakottaa näytettävien tiedostojen tyypin. Tämä mahdollisti mediakirjaston tekemisen siten, että se toimii käytännössä mustana laatikkona, joka ei tarvitse kuin käynnistää ja ulos saa valitun tiedoston. Tästä on huomattavasti hyötyä, jos esimerkiksi pelin tekijä haluaa käyttää mediakirjastoa suoraan moduulista.

Kirjastolle voi asettaa myös muita arvoja, mutta nämä arvot kirjoitettaan kirjaston sisällä yleensä myöhemmin uusiksi.

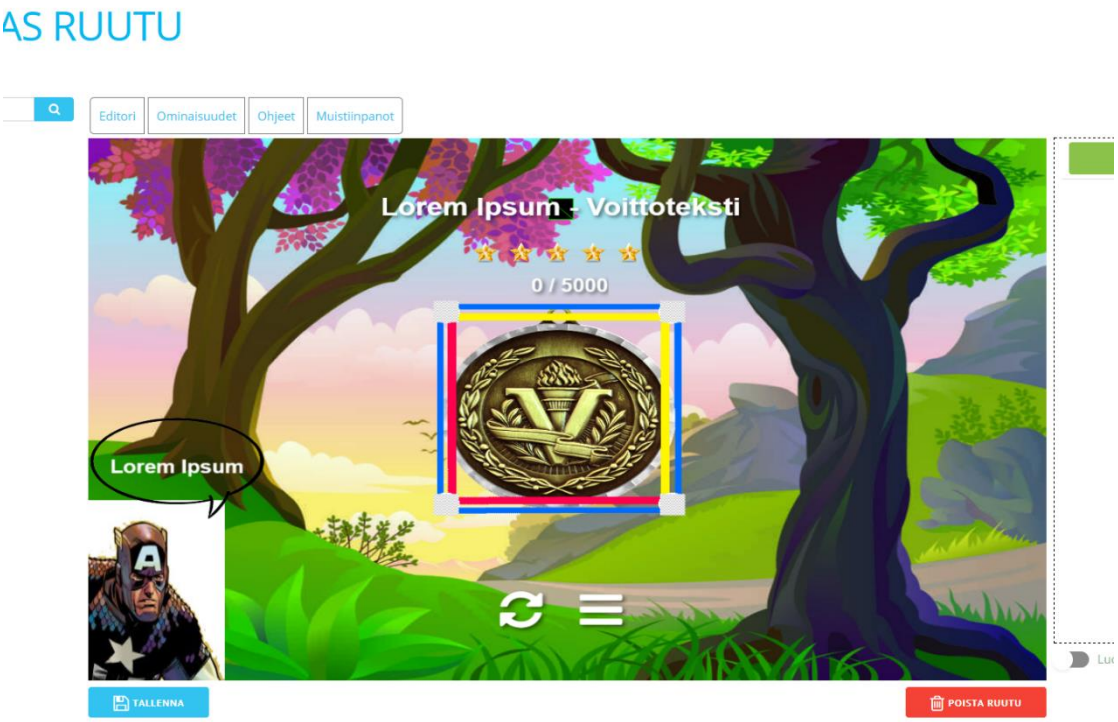
Mediakirjastossa käytettiin ns. callback-mallia, joissa tiettyihin tapahtumiin voi kiinnittää omaa koodia parametrin tai ennalta nimetyn arvon määrittämisellä. Sama malli on vahvasti käytössä esimerkiksi React-kirjastossa. Mediakirjastossa ulkopuolinen käyttäjä määrittää sellaisen funktion mitä haluaa kutsuttavan, kun valinta tapahtuu. Määritetty funktio ei kuitenkaan saa parametrina tietoja, vaan mediakirjasto tallentaa valitut elementit selaimen SessionStorageen eli selaimen sisään rakennettuun avain-arvo-tietokantaan. SessionStoragea käytetään järjestelmässä muutenkin tiedon tallentamiseen, joten näiltä osin mediakirjasto on yhtenäinen muun järjestelmän toiminnan kanssa. Tietovarastosta tätä dataa pystyy sitten hakemaan mistä päin järjestelmää tahansa.

Tämän lisäksi mediakirjastossa pyrittiin minimalisoimaan palvelinkutsut latausaikojen vähentämiseksi. Kun mediakirjasto ensimmäisen kerran laitetaan sivulle, se tekee heti kutsun palvelimelle tiedostojen lataamiseksi. Ainostaan omien latausten näyttäminen laukaisee uuden kutsun mediakirjastossa. Tämä johtuu siitä, että omat lataukset haetaan tietokannasta, kun taas normaalisti tiedostot haetaan skannaamalla määritettyä tiedostokansiota. Tämä rikkoo hieman tavanomaisia toteutustapoja, joissa käyttäjän lataukset yleensä tallennetaan tietokantaan jo siitäkin syystä, että tietokantakyselyt ovat tiedostojen käsittelyä nopeampia. Järjestelmässä päädyttiin kuitenkin tähän malliin, jotta voitaisiin varmistaa, että kaikki palvelimen tiedostot ovat aina nähtävillä, vaikka tietokannassa tapahtuisi tietojen katoamista tai korruptiota. Järjestelmässä ei ole erikseen tiedostojen käsittelytoimintoa, vaan se tapahtuu mediakirjaston kautta.

6.7 Voittoruutueditori

Voittoruutujen määrittelyä varten tehtiin oma editori. Koska voittoruudut pyörivät myös Phaser-pelimoottorin päällä ja myös voittoruutujen data tulee jotenkin säilyttää tietokannassa, oli tehtävien samankaltaisuuden takia luontevaa käyttää voittoruudussa uudelleen tehtäväeditorin koodia. Tällä tavalla myös käyttökokemus pysyy yhtenäisenä ja helppokäyttöisenä. Voittoruutueditori näkyy kuviossa 23. Editorissa

näkyvät elementit ovat kaikki kiinteitä eli editorissa ei voi lisätä tai poistaa asioita. Tietyt asiat saa kyllä poistettua näkyvistä, mutta muuten itse rakenne on kiinteä ja editorilla muokataan elementtien sisältöä kuten tekstiä tai kuvaa.



Kuvio 23. Voittoruutueditori

Voittoruutujen toimintaa ei tarvitse määrittää moduulilla, koska tietueita on rajattu määrä. Tästä syystä määritettiin se tieto mikä yleensä tulee moduulista suoraan JavaScript-tiedostossa. Voittoruutujen rakennuskoodi taas on aina ollut kiinteä osa järjestelmää, joten editoria varten tarvitsi vain määritellä, että se on ainoa käytettävä tila, jonka editori ajaa ennen omia toimintojaan.

Resurssina voittoruudut ovat oma resurssinsa ja niitä voi listata ja muokata samalla tavalla kuin pelejä ja tehtäviä. Editorista on linkki suoraan tehtäväruudun editoriin helppokäyttöisyyden vuoksi. Palvelinpuolella tuo jonkin verran monimutkaisuutta datan muuntimet. Tieto on tietokannassa sarakkeissa, kun taas JavaScript-koodi käyttää JSON-muotoista sisäkkäisistä tietueista koostuvaa datarakennetta. Näin kuitenkin voidaan pitäytyä standardin mukaisessa tietokantarakenteessa voittoruutujen suhteen.

Ratkaisu mahdollistaa luontevan tavan luoda ja muuttaa voittoruutuja. Koska WebGL- ja canvas-elementeistä, joita myös Phaser käyttää, voi helposti tallentaa base64-muotoisen kuvankaappauksen, pystyy pelissä nyt valitsemaan kuvien perusteella, minkä voittoruudun haluaa pelilleen. Verrattuna edelliseen ratkaisuun, jossa voittoruutujen tiedot piti määrillä umpimähkäisesti lomakkeiden kautta, ratkaisu on paljon parempi.

6.8 Tietokantarakenne

Tietokannan rakenne on kuvattu liitteessä 2. Rakenteessa on eri väreillä eroteltu toisiinsa liittyvät kokonaisuudet.

6.8.1 Pelaajaan liittyvät taulut

Keltainen alue on pelaajiin liittyvää dataa. Päätauluna tälle toimii ”player”-niminen taulu. Tässä taulussa pidetään kaikki suoraan pelaajaan kuuluva data. Pelaajille voi myös asettaa tunnisteita. Samat tunnisteet ovat yleiskäyttöisiä eli ne voivat olla monella pelaajalla. Tunnisteiden avulla opettajat pystyvät jakamaan pelaajia sisäisesti tiettyihin ryhmiin.

Toinen moni moneen taulu on ”player_has_games”, jonka toinen pää on tehtävien taulussa. Tässä taustalla on toiminallisuus, jonka avulla opettajat voivat asettaa hallintapuolelta pelaajalle tiettyjä pelejä. Tässä vaiheessa kaikilla pelaajilla on kaikki pelit näkyvissä, mutta lopullisessa käytössä on hyödyllisempää, että pelaajille voidaan luoda erillisiä paketteja heille suunnatuista peleistä.

Viimeinen pelaajin liittyvä taulu on ”loginInfo_player”. Tämän taulun ideana on eriyttää kirjautumiseen liittyvä data itse käyttäjään liittyvästä datasta. Tämä on tehty sen vuoksi, että mikäli pelaajataulun sisällön sotkee esimerkiksi tekemällä migraation väärin, niin kirjautumiseen liittyvät tiedot eivät katoa samalla. Salasanojen lisäksi taulussa voisi olla esimerkiksi lokitietoja kirjautumisista.

6.8.2 Admin-käyttäjien tiedot

Admin-käyttäjien päätaulu on ”controller”. Kuten käyttäjien kohdalla, tässä taulussa pidetään kaikki admin-käyttäjään suoraan liittyvät tiedot. Tämän lisäksi myös admin-käyttäjien kirjautumistiedot on eriytetty, samasta syystä kuin pelaajilla.

Admin-käyttäjillä on lisäksi ”assets”-taulu. Tällä taululla pidetään seurantaa siitä, mikä ladattu tiedosto kuuluu millekin käyttäjälle. Tieto on hyödyllinen muun muassa mediakirjastossa, sillä ajan myötä media kirjastossa olevien tiedostojen määrä kasvaa sangen suureksi.

6.8.3 Pelipakettien tiedot ja kansiot.

Pelipaketteihin liittyvät taulut on ympyröity liitteessä 2 punaisella. Päätaulu on ”game_packages”. Pelipaketteihin kuuluu tehtäviä ja näiden suhde on kuvattu ”gamepackage_has_games”-taulussa. Yksi tehtävä voi myös kuulua moneen tehtävään, joten relaatio on moni moneen.

Tämän lisäksi alkuperäisessä suunnitelmassa pelipaketeille haluttiin kansiointimahdollisuus. Kansiot toimisivat hallintapuolella avustavana lajittelumekanismina siten, että pelipaketin tehtävät voisi asettaa sisällä kansioihin. Ominaisuudesta olisi hyötyä suurissa pelipaketeissa. ”folders”-taululla on viittaus myös tehtäviin. Tehtävä voi olla yhdessä kansiossa, mutta yksi kansio voi olla monessa paketissa. Kansio voi olla myös toisen kansion sisällä, joten ”folders”-taululla on viittaus myös itseensä.

Kansiointiominaisuudelle ei ole tehty minkäänlaista tukea tietokannan ulkopuolella. Kunhan sen suunniteltu toimintatarkoitus on saatu dokumentoitua, nämä taulut todennäköisesti poistetaan.

6.8.4 Tehtävät

Tehtävien perustaulu on ”games” ja niihin liittyvät taulut on ympyröity mustalla. Perustaulussa on kaikki suoraan tehtäviin liittyvät tiedot.

Tehtäville voi asettaa ohjeita. Yhdellä tehtävällä voi olla monta ohjetta ja nämä muodostavat yhdessä vaiheistetun ohjeistuksen. Ohjeita varten on taulu ”game_instructions”. Yksi ohje voi kuitenkin olla vain yhdessä pelissä, joten tämä on perinteinen yksi moneen relaatio.

Versiointijärjestelmää varten on olemassa taulu ”games_bleedind_edge”. Jokaista tehtävää kohden on olemassa vähintään yksi rivi versiointitaulussa. Ennen julkaisua jokainen tallennus asettaa tehtävän datan versiointitauluun, eikä varsinaiseen tehtävään. Tehtävän editointi ja esikatselu hakevat datan versiointitaulusta, kun taas pelaaminen itse tehtävätaulusta. Julkaisussa data siirretään versiotaulusta itse tehtävätauluun. Tällä hetkellä jokaisella tehtävällä voi olla vain yksi versio, mutta relaatio on silti yksi moneen eli periaatteessa tehtävällä voisi olla monta versiota.

Viimeisenä tehtäviin liittyvänä tauluna on ”Difficulties”. Kuten ”folders”, tämä liittyy alkuperäiseen vaatimusmäärittelyyn, eikä ole tällä hetkellä käytössä. Ideana tässä taulussa on, että järjestelmään voi luoda yleispäteviä vaikeusasteita, joita voi sitten liittää tehtäviin. Dokumentoinnin jälkeen taulu tullaan poistamaan.

6.8.5 Muut taulut

Muut taulut ovat itsenäisiä kokonaisuuksia, jotka saattavat tosin liittyä jollain relaatiolla muihin järjestelmän osioihin.

Violetilla värillä näkyy voittoruuduille tarkoitettu taulu. Voittoruudut ovat itsenäisiä kokonaisuuksia, joita muokataan voittoruutueditorilla. Jokaisella tehtävällä on olemassa yksi voittoruutu ja yksi voittoruutu voi olla monessa pelissä. Tämä olisi siis yksi moneen relaatio, jos käytössä olisi normaalit relaatiot. Tieto käytetystä voittoruudusta säilytetään kuitenkin tehtävän JSON-muotoisessa datassa, sillä näin varmistetaan yhtenäinen rajapinta datan käyttöön moduuleissa. Tästä syystä voittoruututaulu on irrallaan muista.

Sinisellä on moduuleille tarkoitettu taulu. Moduulit ovat PHP-tiedostoja, mutta ladatuista moduuleista pidetään tietoa tietokannassa, koska sieltä lukeminen on helpompaa kuin kansiosta. Näin voidaan myös mahdollistaa helposti erillisen nimen asetta-

minen moduulille sekä helppo relaation luonti tehtävään. Tämä yksi moneen relaatio, tehtävä on yhden moduulin mallinen, mutta yhden moduulin päälle on voitu tehdä monta tehtävää.

Harmaalla värillä on ympyröity ”migration_versions”-taulu. Symfonyn Doctrine Migrations toiminto käyttää tätä taulua tallentaakseen tiedon tietokannan nykyisestä versiosta. Näin migraation ajaminen tietää mitkä migraatiota ovat uusia ja mitkä vanhoja. Taulu ei liity muuhun järjestelmään mitenkään.

Tilastoille on taulu ”stats”, joka on merkitty kuvaan vaaleammalla sinisellä. Taulussa on viittaukset tehtävään ja pelaajaan. Näin voidaan tallentaa tilastoja sekä pelaajista, että tehtävistä itsestään. Taulun ”data” tietue on vapaamuotoinen kenttä, johon voi tallentaa joko yhden arvon tai esimerkiksi JSON-muotoista dataa. Statistiikan tyyppi määritetään ”type”-kentällä ja näin voidaan helposti hakea tietyn tyyppiset tilastot taulusta. Koska on hyvin vaikeaa määrittää ennalta, että minkälaista statistiikkaa milloinkin tarvitaan, niin tässä ei ole yritettykään lähteä rajoittamaan sitä tiukasti määritetyllä taulun rakenteella.

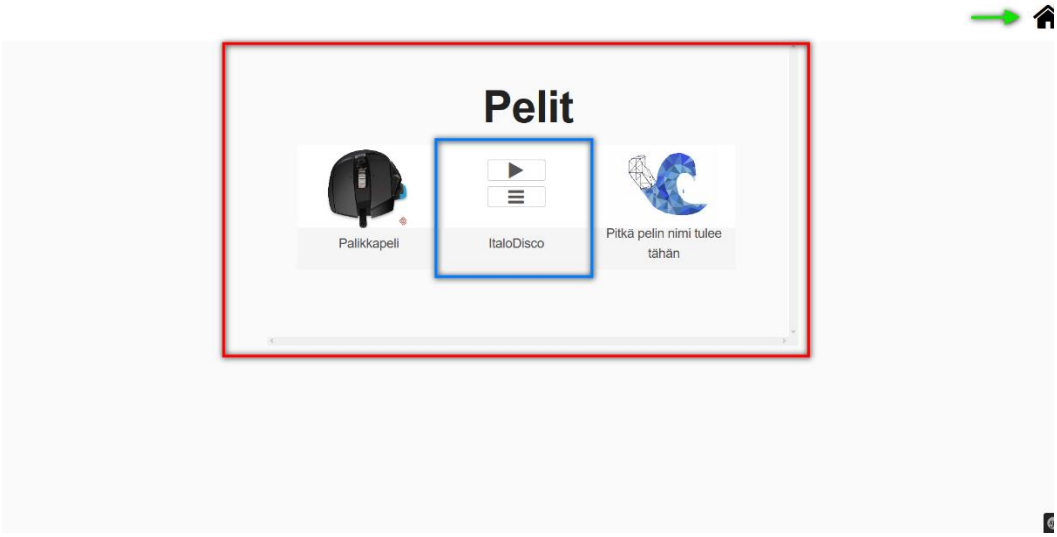
6.9 Pelipuoli

Pelipuoli on järjestelmän toinen puolikas, jossa pelaajat eli käytännössä oppilaat, pelaavat heille näkyvissä olevia pelejä ja tehtäviä. Ideana on, että tämä pelipuoli on järjestelmän oletus sisääntuloreitti. Pelaajat kirjautuvat tunnuksillaan sisään ja näkevät heille asetetut pelit, pääsevät tutkimaan profiiliaan ja näkemään tilastoja etenemisestään. Pelaajat voivat joko valita suoraan jonkin pelin pelattavaksi, tai valita pelistä jonkin tietyn tehtävän.

Pelipuoli ei ollut osa tätä opinnäytetyötä, mutta se on sisällytetty tähän lukuun kokonaiskuvan saamiseksi.

Pelipuolen tyylillinen fokus on selkeästi kliinisyydessä ja helppokäyttöisyydessä. Toimintoja ei tuoda paljoa ja ne toiminnot mitä on, pyritään tuomaan selkeästi ja isona. Pelipuolen tekeminen on ollut alemman tason prioriteetti alusta alkaen eikä se myöskään ole osa tätä opinnäytetyötä, joten sen nykyinen tila on hyvin prototyyppinen.

Kuviossa 24 näkyy punaisella pelien listaus. Sinisellä on ympyröity mitä tapahtuu, kun yhdestä pelistä painaa. Kuva siirtyy pois ja tilalle tulee selkeät toiminnot. Vihreällä nuolella hieman kuvan ulkopuolella näkyy navigaation idea, tuodaan toiminnot sielläkin selkeillä isoilla ikoneilla. Toimintoja navigaatiossa olisi suunnilleen kolme, siirtyminen etusivulle, siirtyminen palkintoihin/tilastoihin ja siirtyminen omaan profiiliin. Toimintojen tulee olla intuitiiviset ja itsensä selittävät.



Kuvio 24. Alustava pelipuolen näkymä, jossa näkyy ajateltu rakenne.

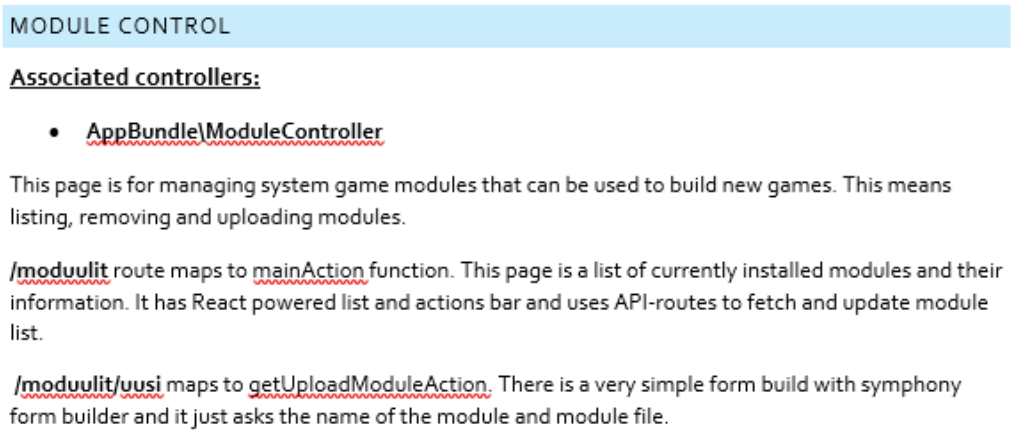
6.10 Tekninen dokumentaatio

Yksi tärkeistä tehtävistä oli kirjoittaa järjestelmälle kattava tekninen dokumentaatio. Tämän avulla voidaan varmistaa järjestelmän kehitettävyys ja laajennettavuus myös ilman alkuperäisiä tekijöitä. Tekniselle dokumentaatiolle päätettiin kolme päätavoitetta:

- Luoda sellainen yleisen tason katselmus järjestelmään, että sen avulla saisi hyvän teoreettisen käsityksen järjestelmän rakenteesta ja osista.
- Määrittää sellaiset ohjeet moduulien luonnille, että ne lukemalla ulkopuoliset tahot osaisivat luoda uusia moduuleja järjestelmään.
- Luoda tarkempi katselmus eri järjestelmän osiin siten, että kehittäjä jolle järjestelmä ei ole tuttu pystyy samaan kuvan siitä, miten yksittäiset ratkaisut toimivat.

Lopullisen dokumentaation rakenne noudattaa melko pitkälti samaa järjestystä kuin yllä olevassa listassa. Teknistä dokumentaatiota saatiin kirjoitettua noin 90% siitä mitä järjestelmästä oli järkevästi dokumentoitavissa. Tähän lukuun ei sisälly pelipuoli, sillä se on vielä hyvin keskeneräisessä vaiheessa eikä sen tekeminen kuulu tähän opinnäytetyöhön. Dokumentoimatta jäi muun muassa laajalti käytössä oleva ”data-bank”-osuus, joka toimii järjestelmässä abstraktoituna rajapintana tietokantaan ja se käyttää sisäisesti Doctrinea. Tätä ei haluttu dokumentoida, koska se on ajalta ennen kuin tekijät ymmärsivät täysin ORM:n hyödyt ja käyttötavan. Kyseinen databank on-kin enemmän teknistä velkaa joka pitäisi poistaa, kuin dokumentoitavaa rakennetta. Myöskin itse teknisen velan kattava dokumentointi puuttuu vielä toistaiseksi.

Teknisessä dokumentaatiossa ei otettu pohjalle mitään varsinaista standardia, vaan sitä kirjoitettiin, kuten tilanteeseen parhaaksi nähtiin. Dokumentaatiota kirjoittaessa muodostui kuitenkin tiettyjä toistuvia tapoja ilmaista asioita. Esimerkiksi kyseiseen aihealueeseen liittyvät luokat pyrittiin ilmoittamaan heti luvun alussa, kuten näkyy kuviossa 25.



MODULE CONTROL

Associated controllers:

- **AppBundle\ModuleController**

This page is for managing system game modules that can be used to build new games. This means listing, removing and uploading modules.

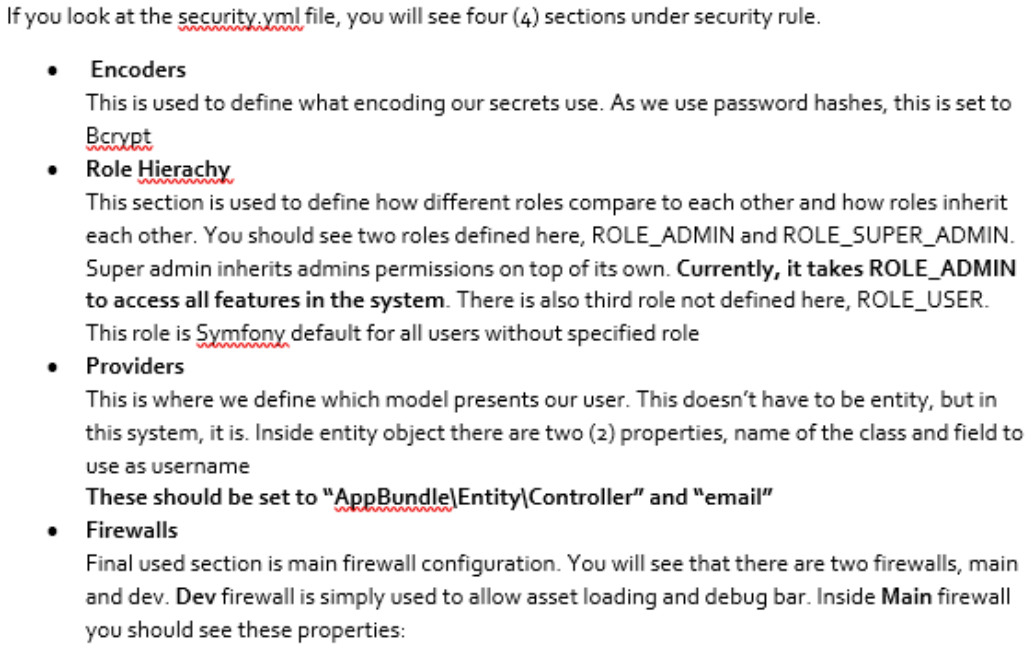
/moduulit route maps to mainAction function. This page is a list of currently installed modules and their information. It has React powered list and actions bar and uses API-routes to fetch and update module list.

/moduulit/uusi maps to getUploadModuleAction. There is a very simple form build with symphony form builder and it just asks the name of the module and module file.

Kuvio 25 Moduulien muokkausta kuvaava kappale teknisessä dokumentaatiossa

Samaa tapaa käytettiin myöhemmin moduulien rakennetta kuvatessa merkitsemällä luvun alkuun sen hetkisen muuttujan nimi. Dokumentaatiossa käytettiin paljon vah-vennuksia merkitsemään tärkeitä kohtia. Tässä oli ideana, että dokumenttia selaava henkilö voi vahvennettuja kohtia seuraamalla löytää haluamansa asian helpommin.

Listoja käytettiin paljon kuvaamaan erinäisiä rakenteita. Kuviossa 26 näkyy security.yml-tiedoston rakenne avattuna listan avulla. Kuviossa näkyy myös, miten tekstissä olevia vahvennuksia on käytetty korostamaan alueita, joiden uskotaan kiinnostavan lukijaa eniten.



If you look at the security.yml file, you will see four (4) sections under security rule.

- **Encoders**
  This is used to define what encoding our secrets use. As we use password hashes, this is set to Bcrypt
- **Role Hierachy**
  This section is used to define how different roles compare to each other and how roles inherit each other. You should see two roles defined here, ROLE_ADMIN and ROLE_SUPER_ADMIN. Super admin inherits admins permissions on top of its own. **Currently, it takes ROLE_ADMIN to access all features in the system**. There is also third role not defined here, ROLE_USER. This role is Symfony default for all users without specified role
- **Providers**
  This is where we define which model presents our user. This doesn't have to be entity, but in this system, it is. Inside entity object there are two (2) properties, name of the class and field to use as username
  These should be set to "AppBundle\Entity\Controller" and "email"
- **Firewalls**
  Final used section is main firewall configuration. You will see that there are two firewalls, main and dev. **Dev** firewall is simply used to allow asset loading and debug bar. Inside **Main** firewall you should see these properties:

Kuvio 26 Listan ja vahvennuksien käyttöä dokumentaatiossa

7 Toteutetun ratkaisun soveltuvuus käytännössä

Ratkaisun soveltavuutta käytäntöön mitattiin sarjalla kysymyksiä, jotka esitettiin Valterille. Myös tekijä itse vastasi kysymyksiin sikäli kuin se on mahdollista ja näin voidaan vertailla asiakkaan näkemystä projektista tekijän näkemykseen. Kysymykset olivat:

- Mikä on käyttökokemus aloittamisesta tehtävän tekemiseen ja testaamiseen?
- Mikä on editorista saatu käyttökokemus?
- Mikä on pelaamisen käyttökokemus?
- Kuinka helppoa on tehdä uusi tehtävä?
- Kuinka helppoa on luoda pelipaketti?
- Onko versioinnista ollut hyötyä
- Onko voittoruutueditorista kerennyt olla hyötyä?
- Mikä on ollut pelaajien käyttökokemus?

- Onko järjestelmää testattu oppilailla vai vain opettajilla?
- Onko järjestelmä koettu hyödylliseksi?
- Pidetäänkö järjestelmää lupaavana?
- Miten nähdään projektin tulevaisuus?
- Paljonko uudelle käyttäjälle tarvitaan koulutusta tai kuinka kauan hänellä menee oppia käyttö?

Tekijän mielestä käyttökokemus alusta loppuun pitäen sisällään tehtävien teon oli kelvollinen. Muutamia ongelmia tuli matkan varrella. Nämä olivat lähinnä järjestelmään jääneitä tiedossa olleita ohjelmointivirheitä. Muuten pelin luominen, tehtävän luominen, esikatselu, pelin tehtävien ja järjestyksen asettaminen toimivat kaikki hyvin ja ilman ongelmia. Käyttökokemusta heikensi jossain kohdissa järjestelmän satunnainen hidastelu.

Asiakkaan puolelta saatu palaute tehtävin teosta editorissa oli se, että peruskäyttö on ollut sujuvaa, mutta tietyissä kehittyneemmissä käyttötavoissa käyttäminen vaati hieman luovaa pohtimista, mikä voi olla tietyille käyttäjille hankalaa. Esimerkkinä tällaisista käyttötapauksista asiakas antoi esimerkiksi näkymättömien korttien, jotka toistavat ainoastaan äänen, käyttämisen ja generaattorin käyttämisen. Generaattori on erikoiskortti, joka antaa ulos muita kortteja. Myös editorissa on ollut joitain ominaisuuksia, jotka ovat vaatineet selittämistä joltakin toiselta käyttäjältä joka on järjestelmään aikaisemmin tutustunut. Tästä ei annettu esimerkkiä, mutta tällainen tunnistettu toiminallisuus on esimerkiksi tallennus-painikkeet, joista ei ole välttämättä ihan selvää mitä mikäkin tallentaa. Pelipakettien hallinnan kohdalla ei sen erikoisempaa mielipidettä osattu antaa, sillä sen käytössä ilmeisesti ei oltu kiinnitetty juurikaan huomiota mihinkään hankaluuksiin. Järjestystoimintoa ei oltu keretty kokeilemaan.

Generaattorikortit ja näkymättömät kortit ovat kuitenkin toiminnallisuuksia, jotka määritetään moduulissa ja joiden toimintaa pystyykin muuttamaan myös suoraan järjestelmässä itsessään. Esimerkiksi näkymättömille korteille voisi helposti lisätä moduulin kautta aivan oman toiminnon editorin sivupalkkiin. Sekä tekijä, että asiakas olivat samaa mieltä siitä, että editorissa on tiettyjä kummallisuuksia, jotka eivät ehkä ole aivan selkeitä. Vastauksia vertailemalla nähdään myös, se että molempien mielestä peruskäyttö editorissa on kuitenkin sujuvaa. Esikatselu ja testaaminen oli toiminut asiakkaalla sekä tekijällä lähes moitteetta, pois lukien se, että asiakas huomautti, että pelinäkymän kaikki painikkeet eivät ole toiminnassa ja tietyillä mobiiliselaimilla

saattoi olla ongelmia. Pelipakettien kohdalla tekijä piti prosessia hyvänä, mutta asiakkaan päässä asiassa ei oltu kiinnitetty huomiota. Tämä voinee omalla tavallaan kertoa siitä, että sen käyttö on ollut sen verran sujuvaa, ettei itse toimintaan ole edes kiinnitetty huomiota.

Asiakkaan päässä ei oltu keretty kokeilemaan versiointia ja voittoruutua, mutta näitä toimintoja esitellessä palaute oli, että toiminnot vaikuttavat hyviltä ja käteviltä. Voittoruudueditorin kanssa kommentoitiin myös, että editori vaikuttaa paremmalta kuin yksittäisten lomakekenttien kautta muokkaaminen. Tekijä itse on muokannut voittoruutuja ensin suoraan JSON-koodiin, sitten lomakkeen kautta ja lopulta editorilla ja hänen mielestä voittoruutueditori on ylivoimaisesti paras näistä tavoista.

Asiakkaan päässä järjestelmää ei ole vielä testattu oikeilla oppilailla, vaan kaikki järjestelmää käyttäneet henkilöt ovat olleet enemmän tai vähemmän teknisesti orientoituneita. Valterilla on ollut harjoittelussa yksi henkilö, joka on harjoittelussaan keskittynyt paljon järjestelmän käyttöön ja tehtävien tekemiseen. Noin tunnissa hänelle oli esitelty koko järjestelmän toiminta ja tämän jälkeen hän ei ole kuulemma kysynyt apua muuta kuin järjestelmässä olevien virheiden kohdalla. Yhden ihmisen otos on aivan liian pieni, jotta voitaisiin vetää todellisia johtopäätöksiä, mutta tämä tukee kuitenkin alustavasti tekijän näkemystä siitä, että järjestelmä käyttö on luontevaa.

Asiakas näkee järjestelmän hyvin lupaavana heidän käyttöönsä, mutta he tarvitsevat todellisia testejä todellisten pelaajien eli oppilaiden kanssa. Järjestelmä tarvitsee pidemmän testausvälin ja pelipuoli pitää saada käyttöön. Mikäli pelit ja tehtävät kuitenkin toimivat myös oppilailla, niin asiakas näkee, että järjestelmä on soveltuva oikeaan käyttöön oppimateriaalien pelillistämisessä. Asiakkaalla on ollut aikaisempiakin kokeiluja tämän kaltaisista järjestelmistä, mutta heidän kokeilemansa järjestelmä koettiin toimimattomaksi. Asiakas arvioi, että aikaisemmin kokeiltu järjestelmä yritti olla liikaa kaikkea ja tästä syystä oikein mitään ei saatu toimimaan. Tässä suhteessa suunniteltu modulaarisuus on todennäköisesti auttanut paljon, sillä LUEM-järjestelmää voidaan helposti laajentaa ja muuttaa tarpeen mukaan.

8 Pohdinta

Lopputuloksena projekti on saatu tilanteeseen, jossa hallintapuolelle tarvittavien uusien ominaisuuksien määrä on hyvin pieni ja hallintapuolella voidaan keskittyä korjaamaan virheitä ja parantamaan käytettävyyttä.

Tilastollisesti projekti on tässä vaiheessa kasvanut melko suureksi. Projektissa on 52 JavaScript-tiedostoa, 104 PHP-tiedostoa (kun lasketaan migraatiot mukaan), 15 CSS-tiedostoa ja 33 Twig-tiedostoa. JavaScript-koodia on kirjoitettu noin 6500 riviä, PHP:ta melkein 15000 riviä, CSS-sääntöjä noin 1100 riviä ja Twig-pohjia noin 1700 riviä. Tilastot on otettu kehitystyökalu PHPStormissa olevalla lisäosalla ja tuloksen ovat näkyvissä kuviossa 27. Nämä tilastot antavat osviittaa projektin laajuudesta. Näin ison kokonaisuuden kehittäminen yksin on hyvin haastavaa ja aikaa vievää. Järjestelmään onkin jäänyt jonkin verran teknistä velkaa, joka tulisi korjata, mutta jonka korjaamiseen ei ole ollut resursseja tämän opinnäyteyön puitteissa. Teknisen dokumentaation tärkeys korostuu myös, sillä projektiin tulevan kehittäjän olisi melko hankalaa päästä kärrylle kaikista osasista ja toiminnasta pelkällä tutkimalla. Teknisen dokumentaation hyödyt näkyivätkin osittain jo kehitystyön aikana, sillä oli tapauksia, jolloin jonkin ominaisuuden pariin palaaminen vaati teknisen dokumentaation lukemista muistin virkistämiseksi.

Extension	Count	Size SUM	Size MIN	Size MAX	Size AVG	Lines
mp3 *(MP3 files)*	9x	8 467kB	7kB	1 544kB	940kB	67992
wav *(WAV files)*	1x	2 365kB	2 365kB	2 365kB	2 365kB	20892
php *(PHP files)*	104x	535kB	0kB	63kB	5kB	14838
mp4 *(MP4 files)*	2x	776kB	115kB	661kB	388kB	7559
js *(JS files)*	52x	244kB	0kB	24kB	4kB	6521
lock *(LOCK files)*	1x	119kB	119kB	119kB	119kB	3294
twig *(TWIG files)*	33x	80kB	0kB	14kB	2kB	1795
css *(CSS files)*	15x	22kB	0kB	5kB	1kB	1192
xml *(XML configuration file)*	23x	39kB	0kB	4kB	1kB	893
yml *(YML files)*	10x	11kB	0kB	3kB	1kB	349
json *(JSON files)*	6x	8kB	0kB	2kB	1kB	334
ogg *(OGG files)*	1x	33kB	33kB	33kB	33kB	141
htaccess *(HTACCESS files)*	3x	3kB	0kB	3kB	1kB	82
md *(MD files)*	2x	2kB	0kB	2kB	1kB	74
pem *(PEM files)*	2x	4kB	0kB	3kB	2kB	65
pem-back *(PEM-BACK files)*	1x	3kB	3kB	3kB	3kB	54
dist *(DIST files)*	2x	1kB	0kB	1kB	0kB	50
ico *(ICO files)*	1x	6kB	6kB	6kB	6kB	36
out *(OUT files)*	1x	0kB	0kB	0kB	0kB	5
txt *(Text files)*	1x	0kB	0kB	0kB	0kB	5
3149445751 *(3149445751 files)*	1x	0kB	0kB	0kB	0kB	0
Total:	271x	12 728kB		2 656kB		4 863kB

6: TODO Statistic 9: Version Control Terminal

Kuvio 27. Projektin tilastoja

Teknistä dokumentaatiota kirjoittaessa pääsi tutustumaan uudestaan jokaiseen järjestelmään osaan ja analysoimaan niitä huomattavasti teoreettisemmin kuin mitä kehittäessä on mahdollista tehdä. Kehittäessä riittää, että saa toiminnallisuuden pelaamaan nykyisen toteutuksen kanssa, mutta dokumentaatiota kirjoittaessa joutuu oikeasti miettimään, että onko ratkaisussa juuri mitään järkeä dokumentaation lukijalle. Tämä johtikin paikoitellen niin kutsuttuun refaktorointiin eli toteutuksen arkkitehtuurin uudistamiseen. Tekninen dokumentaatio muodostui kuitenkin sangen kattavaksi sekä selittäväksi ja siitä tuleekin olemaan selkeää hyötyä tulevaisuudessa. Tämä näkyi selkeästi niinä kertoina, kun tekijä joutui myöhemmin itse lukemaan dokumentaatiota. Itseharhan vuoksi tekninen dokumentaatio tulisi kuitenkin luettaa myös ulkopuolisella henkilöllä.

Dokumentaatiota kirjoittaessa ymmärrettiin myös, että tämän kokoisessa järjestelmässä pitäisi olla alusta alkaen mukana senioritason osaaja suunnittelemassa arkkitehtuuria ja vetämässä kehitystyötä. Järjestelmässä on hirveän monta teknillistä, taloudellista ja tietoturvallista osa-aluetta, joista kaikkia tulee huomioida kehityksessä. Muun muassa Valterin vaatimukset oppilaiden tietoturvalle ovat hyvin korkeat ja asettavat tiettyjä rajoitteita. Alkuperäisessä vaatimusmäärittelyssä ollut kaupallistaminen on myös jäänyt nykyisellään kokonaan pois. Kokemattomalle tekijälle sellaisen toteuttaminen on hyvin suuri tehtävä ja vaatii paljon resursseja. Se ei tosin myöskään ole Valterille ajan kohtainen, joten resursseja on voitu siirtää tärkeämpiin tehtäviin. Tämän lisäksi tulee projektin vetäjän huomioida erityisvaatimukset käytettävyydelle, sillä se on tärkeä osa järjestelmän mielekkyyttä. Oman mausteensa suunnittelutyöhön tuo myös modulaarisuus, joka itsessään on haastava tehtävä, mutta vaatii myös tarkkaa seuraamista, että muu kehitys ei riko modulaarisuutta.

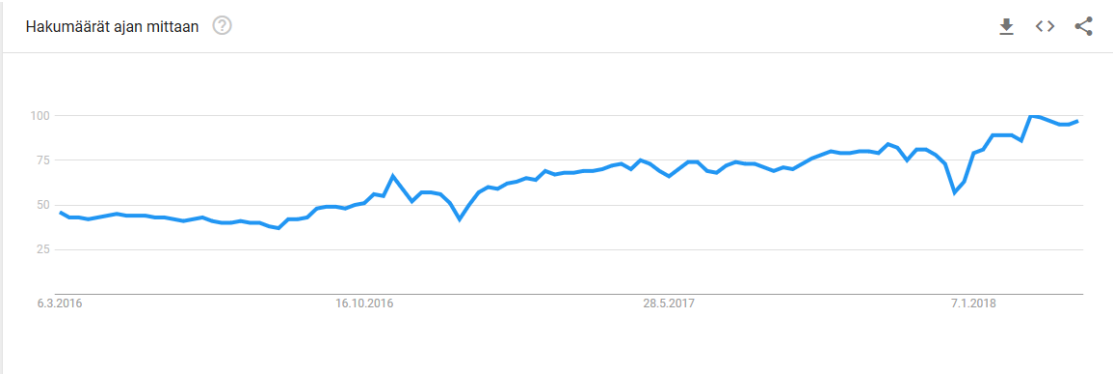
Tehtyjä ratkaisuja arvioimalla voi todeta, että ne eivät ole huonoja, mutta ainakin osan voisi tehdä myös paremmin. Tehdyt ratkaisut toimivat ja eivät esimerkiksi estä jatkokehitystä, mutta esimerkiksi moduulien suhteen rakenne voisi olla mielekkäämpi ja se uudistettiinkin työn aikana kertaalleen. Paikoitellen järjestelmään on myös jäänyt myös logiikkaa, joka rikkoo modulaarisuutta. Kaikki tunnistetut tapaukset on kommentoitu koodissa, mutta tämäkin osoittaa, että projekti vaatisi täysipäiväisen seniorikehittäjän valvomaan laatua, ettei näitä pääsisi tapahtumaan.

Alla oleva taulukko 4 kuvaa järjestelmästä tunnistettua teknistä velkaa.

Taulukko 4. Tekninen velka

Velka	**Selite**
Babel-kääntäjä	Nykyisellään Babel-kääntäjää käytetään ulkoisena script-tagina, mikä on tavattoman hidasta. Vaatisi laajan refaktoroinnin jossa otetaan käyttöön ES6-syntaksin import- & export-lausekkeet
Pelaajan tilit	Tämän suhteen toteutus jäi kesken johtuen teknisistä vaikeuksista. Vaatisi arviolta 50 tuntia työtä lisää.
Databank-systeemi	Kaikki kohdat, joissa käytetty databank-systeemiä, pitäisi refaktoroida käyttämään Doctrinea controller-funktioissa. Näitä on paljon.
Reittien epämääräisyys	Pitäisi luoda järkevät säännöt siitä, että miten reitit nimetään ja millä tavalla. Osa API-reiteistä on esimerkiksi siirretty /api… tapaan, mutta ei kaikki. Reittejä on järjestelmässä 64, joten iso työ tarkistaa kaikki.
Lokitus	Lokeja kirjoitetaan hyvin vähän, jos ollenkaan. Ihan ensimmäiseksi pitäisi luoda yhtenäinen tapa lukea lokeja.
Tiedostojen, muuttujien ja funktioiden nimet	Nimissä on käytetty sekaisin a-b ja aB nimeämistapaa. Pitäisi yhtenäistää.
Testit	Järjestelmässä ei ole testejä.

Projektissa käytetyt teknologiat kuten Symfony, React, Doctrine ja Twig ovat toimineet hyvin myös jatkokehityksessä. Näiden suhteen on tehty projektia aloittaessa hyviä valintoja. Näitä tekniikoita analysoimalla voidaan todeta, että tekniset valinnat eivät aiheuta projektissa ongelmia, vaan päinvastoin tukevat projektia myös tulevaisuudessa. Muun muassa React on sittemmin vain kasvattanut suosiotaan. Symfonystä sen sijaan on julkaistu uusi versio, mutta muuten sen suosio on pysynyt sangen tasaisena. Symfony on edelleen toiseksi suosituin PHP-ohjelmistokehys. Kuvioissa 28 ja 29 näkyy Reactin ja Symfonyn hakumäärien kasvu Googlessa.



Kuvio 28. Reactin suosion kasvu Googlessa

Kuvio 29. Symfonyn suosion kasvu Googlessa

Järjestelmän modulaarisuus on onnistunut teknisiin tavoitteisiin nähden hyvin. Tehtävät pyörivät täysin moduulien ympärillä ja uusien moduulien luonti järjestelmään on mahdollista, mikä tarkoittaa, että järjestelmä on helposti laajennettavissa. Teknisen dokumentaation ja moduulityökalun avulla moduulit ovat hallittavissa ja luotavissa huomattavasti helpommin kuin mitä aikaisemmin. Näin on korjattu tiettyjä moduuleissa ilmenneitä ongelmia. Tehtäviä rajoittavia ongelmia moduuleista ei ole toistaiseksi löydetty. Kuitenkin, kuten luvussa 5.5 todetaan, niin moduuleissa voisi toimia jokin standardisoitu rakenne kuten XML tai JSON paremmin ja se korjaisi tiettyjä ongelmia mitä nykyisessä toteutuksessa on.

Modulaarisuuden kehitys oli hyvin haasteellista, mutta myös hyvin palkitsevaa ja opettavaa. Modulaarisen järjestelmän kehitys pakottaa miettimään asioita laajasti ja miettimään niin käytettyjen teknologioiden teoreettista puolta kuin myös järjestelmän käyttäjien käyttökokemusta. Näillä kaikilla on merkitystä, kun mitataan lopputuloksen onnistuneisuutta. Modulaarisuutta suunnitellessa tuli muun muassa ottaa

huomioon, että miten PHP:ssa voidaan dynaamisesti ladata luokkia tekstidatan perusteella. Samoin tuli ottaa huomioon moduulin lataamisen vaikutus esimerkiksi muistinkäyttöön.

Moduulien käytettävyyttä ei ole päästy testaamaan ulkopuolisilla kehittäjillä, joten on hankalaa antaa arviota sen käytettävyyttä kehittäjän näkökulmasta. Uuden moduulin lisääminen järjestelmään on kuitenkin hyvin yksinkertaista, riittää kun moduulitiedoston saa oikeaan kansioon. Koodipuolella kehittäjille taas on annettu hyvin vapaat kädet, sillä moduulissa oleva koodi ajetaan sellaisenaan ja ympäröivä järjestelmä antaa vain rajapintoja muihin osiin ja toimintoihin. Aiemmin mainittiin, ettei modulaarisuus ole ainakaan vielä rajoittanut tehtävien tekoa ja tämä on todennäköisesti juurikin tämän vapauden ansiota.

Alkuperäinen tietokantarakenne on toiminut kelvollisesti. Käytössä olevista kahdestakymmenestä taulusta kolme on mietitty myöhemmin uudestaan. Käyttämättömistä tauluista oikeuksille varattu taulu on poistettu kokonaan, sillä saman toiminnallisuuden pystyy tekemään Symfonyn Guard-komponentilla paljon helpommin, varmemmin ja paremmin. Pelaajien tunnistetaulu muutettiin yksinkertaiseksi moni moneentauluksi, sillä tämä ratkaisu toimii paremmin muun muassa Doctrinen kanssa. Tilastojen taulussa luovuttiin yrityksestä mallintaa kaikki mahdolliset tilastoitavat asiat teknisesti oikealla tavalla ja siirryttiin käyttämään JSON-tekstikenttää ja tilastojen hallitsijaa. Tällä tavalla tilastotaulu on yleiskäyttöisempi ja helpompi ymmärtää. Muihin tauluihin on tullut lisäyksiä, mutta niiden perusrakenne ja relaatiot ovat pysyneet samana, sillä ne ovat toimineet myös jatkokehityksen aikana hyvin. Olisi kuitenkin järkevää poistaa käyttämättömät taulut ja käyttämättömät Doctrinen entity-tiedostot heti kun niiden suunniteltu toiminallisuus on saatu dokumentoitua. On todennäköistä, että niiden olemassaolo aiheuttanee hämmennystä ulkopuoliselle tarkistelijalle.

Alkuperäinen projekti oli tekijöilleen hyvin kehittävä kokemus ja auttoi myös tekijää myöhemmin työelämässä. Projektiin palaaminen puolen vuoden työkokemuksella valaisi kuitenkin huomattavasti sitä tasoeroa mikä oli projektin alkuperäisessä tekovaiheessa ja siihen palatessa. Tämän uuden kokemuksen avulla järjestelmää pystyttiin tarkistelemaan nyt kriittisemmin ja aikaisempia ratkaisuja tarpeen mukaan korjaamaan. Samalla jatkokehityksen aikana opittiin uusia asioita järjestelmässä olevista

kokonaisuuksista. Esimerkiksi modulaarisuuden kanssa on ymmärretty paremmin PHP:n sisäistä toimintaa, opittu sen edistyneempiä toimintoja, kuten luokkien rakenteellisesta parsimista ja opittu muistinkäytöstä. Samoin editorin jatkokehitys on pakottanut tutkimaan Phaser-kirjaston sisäistä toimintaa paljon ja kehityksen aikana onkin jopa raportoitu Phaserin kehitystiimille uusia ohjelmointivirheitä. Reactia käytetään projektista melko eri lailla mitä alalla yleisesti. Yhdistämällä tämä työelämässä opittuun, on projekti opettanut paljon Reactin toiminnasta ja siitä, miten sitä voidaan hyödyntää missäkin tilanteessa. Kuten aiemmin mainittiin, teknisen dokumentaation kirjoittaminen on ollut iso kasvun paikka, ei ainoastaan teknisen dokumentaation kirjoittamistaidoilla, vaan kyvylle analysoida ratkaisuja täysin toteutuksesta irrallaan ja nähdä niiden hyvät ja huonot puolet. Lopuksi, projekti on opettanut Babel ja Grunt JavaScript-työkalujen toiminnasta. Babelista jäi teknistä velkaa, mutta sen opiskeleminen antoi selkeän suunnitelman sen korjaamiseksi. Grunt-työkalua käytetään Phserissa sisäisesti ja koska projektissa on kustomoitu Phaser käytössä, tuli se sitä kautta tutuksi. Aikaisemmin kustomointi oli tehty suoraan Phaserin pakattuun koodin, mutta projektissa siirryttiin käyttämään Phaserin tarjoamaa kehitysversiota, josta luodaan erikseen versio järjestelmän käytettäväksi.

Tulevaisuus näyttää kuinka paljon Valteri saa LUEM-järjestelmästä irti. Tämän opinnäytetyön aikana oli vielä sovittu, että projektia aikaisemmin jatkanut toinen tekijä tekee myös opinnäytetyönsä LUEM-järjestelmästä. Projektissa kehitetty dokumentaatio mahdollistaa uusien kehittäjien mukaan ottamisen projektiin järkevästi ja moduulien luontiin tehdyt ohjeet mahdollistavat sekä vanhojen moduulien ylläpidon, että uusien moduulien luonnin ilman alkuperäisten tekijöiden tukea. Projektin suurimpia saavutuksia lieneekin, ettei se ole enää niin riippuvainen sen alkuperäisistä tekijöistä, vaan voi elää omana kehitysprojektina asiakkaalla. Projektissa korjattu teknine velka ja tehdyt toiminnot tukevat myös tätä. Tekijällä itsellään ei ole enää kahden vuoden jälkeen niin hirveästi paloa jatkaa projektin päävastaavana. Järjestelmää tultaneen parantamaan tietyissä määrin myös opinnäytetyön jälkeen muun muassa dokumentaation osalta ja tiettyjen virheiden korjauksien suhteen, mutta nämä toimenpiteet tähtäävät siihen, että asiakas voisi ottaa projektin vastuun itselleen.

Nykyisellään järjestelmää käytetään Valterin sisäisesti koekäytössä. Tietyt opettajat tekevät järjestelmällä testipelejä ja peluuttavat niitä kohdehenkilöillä. Samoin järjestelmää esitellään muulle henkilökunnalle. Tekijän mielestä järjestelmässä on paljon potentiaalia ollakseen se kattava oppimateriaalien pelillistämisalusta mitä alun perin lähdettiin suunnittelemaan ja toivookin, että järjestelmästä otetaan täysi hyöty irti.

Lähteet

Atomicity. N.d. Technopedian kuvaus atomisuudesta. Viitattu 15.6.2017. https://www.techopedia.com/definition/24729/atomicity

Black, P. 2004. Transitiotion function. NIST organisaation määritelmä siirtymäfunktiolle, verkkoartikkeli. Päivitettu 17.12.2004. Viitattu 12.12.2017. https://xlinux.nist.gov/dads/HTML/transitionfn.html

Black, P. 2016. Finite state machine. NIST organisaation määritelmä tilakoneelle, nettiartikkeli. Päivitettu 6.6.2016. Viitattu 12.12.2017. https://xlinux.nist.gov/dads/HTML/finiteStateMachine.html

Buna, S. 2017. Yes, React is taking over front-end development. The question is why. React-kirjaston suosiota luotaava blogipostaus. Viitattu 23.1.2018. https://medium.freecodecamp.org/yes-react-is-taking-over-front-end-development-the-question-is-why-40837af8ab76

Ford, T. N.d. 4 Pros and Cons to Gamified Learning. Pelillistämisen hyötyjä luotaava nettiartikkeli. Viitattu 5.2.2018. https://tophat.com/blog/gamified-learning/

Hamp, S. 2015. Vastaus kysymykseen “Why do many web developers hate jQuery?” Hashnode nimiellä verkkosivulla. Viitattu 12.3.2018. https://hashnode.com/post/why-do-many-web-developers-hate-jquery-ciibz8fp801g9j3xtgx19utpe

Ibrahim, N. 2010. Stack Overflow vastaus liittyem modulaarisuuteen. Päivitetty 5.5.2015. Viitattu 10.12.2017. https://stackoverflow.com/questions/2768104/how-to-create-a-flexible-plug-in-architecture

Jenkins, J. 2016. Top 4 Gamification Problems To Avoid. Artikkeli eLearning Industry verkkosivulla. Viitattu 5.2.2018. https://elearningindustry.com/top-4-gamification-problems-avoid

Khan, S. 2013. Object oriented programming revisited. Blogipostaus. Viitattu 10.12.2017. https://codeitsfun.com/2013/05/16/object-oriented-programming-revisited/

Kirkham, P. 28.2.2009. Stack Overflow vastaus siihen, mitä on taikuus koodissa, Päivitetty 26.5.215. Viitattu 11.12.2017. https://stackoverflow.com/questions/598318/when-talking-about-programming-languages-what-is-the-definition-of-magic

MariaDB versus MySQL – Features. 2010. MariaDB:n verkkosivun artikkeli MariaDB:n eroista MySQL:ään. Päivitetty 15.9.2017. Viitattu 26.12.2017. https://mariadb.com/kb/en/library/mariadb-vs-mysql-features/

Marquis, J. 2012. The Trouble with Gamification. Blogipostaus pelillistämisen vaikeuksista. Viitattu 6.2.2018. https://www.onlineuniversities.com/blog/2012/07/the-trouble-gamification/

Poolet, M. 1999. SQL by Design: Why You Need Database Normalization. Verkkoartikkeli siitä, miksi normalisointia tarvitaan. Viitattu 16.6.2017. http://www.itprotoday.com/microsoft-sql-server/sql-design-why-you-need-database-normalization

Relational Database overview, N.d. Artikkeli osana Oraclen arkkitehtuurikuvaus JDBC-rajapinnasta. Viitattu 26.12.2017. https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html

Ronan, A. 2015. The Ultimate Guide to Gamifying Your Classroom. Internet-artikkeli, julkaistu 30.7.2015. Viitattu 30.10.2017. http://www.edudemic.com/ultimate-guide-gamifying-classroom/

Sebrechts, J. 2013. Stack Overflow vastaus NoSQL:n skaalautuvuudesta suhteessa SQL-ratkaisuihin. Muokattu 16.10.2017. Viitattu 26.12.2017. https://softwareengineering.stackexchange.com/questions/194340/why-are-nosql-databases-more-scalable-than-sql

Shalom, N. 2012. Stack Overflow vastaus tietokantojen skaalautuvuudesta. Muokattu 6.2.2017. Viitattu 26.12.2017. https://stackoverflow.com/questions/11707879/difference-between-scaling-horizontally-and-vertically-for-databases

Tezer, O. 2014. SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems. Blogipostaus eri SQL-tietokantojen vertailusta. Viitattu 26.12.2017. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems

Tietokannan normalisoinnin perusteiden kuvaus, N.d. Microsoftin tukiartikkeli normalisointitekniikkaan aloittelijoille. Muokattu 4.7.2017. Viitattu 16.6.2017 https://support.microsoft.com/fi-fi/help/283878/description-of-the-database-normalization-basics

What is a Relational Database? N.d. Tietokantoihin liittyvää oppimateriaalia opettajille tarjoava nettisivu. Viitattu 16.4.2018. https://gcsecomputing.org.uk/theory/relational-databases/

What is atomicity in dbms. 2014. Stack Overflow keskustelu atomisuuden merkityksestä. Viitattu 16.6.2017. https://stackoverflow.com/questions/24029620/what-is-atomicity-in-dbms

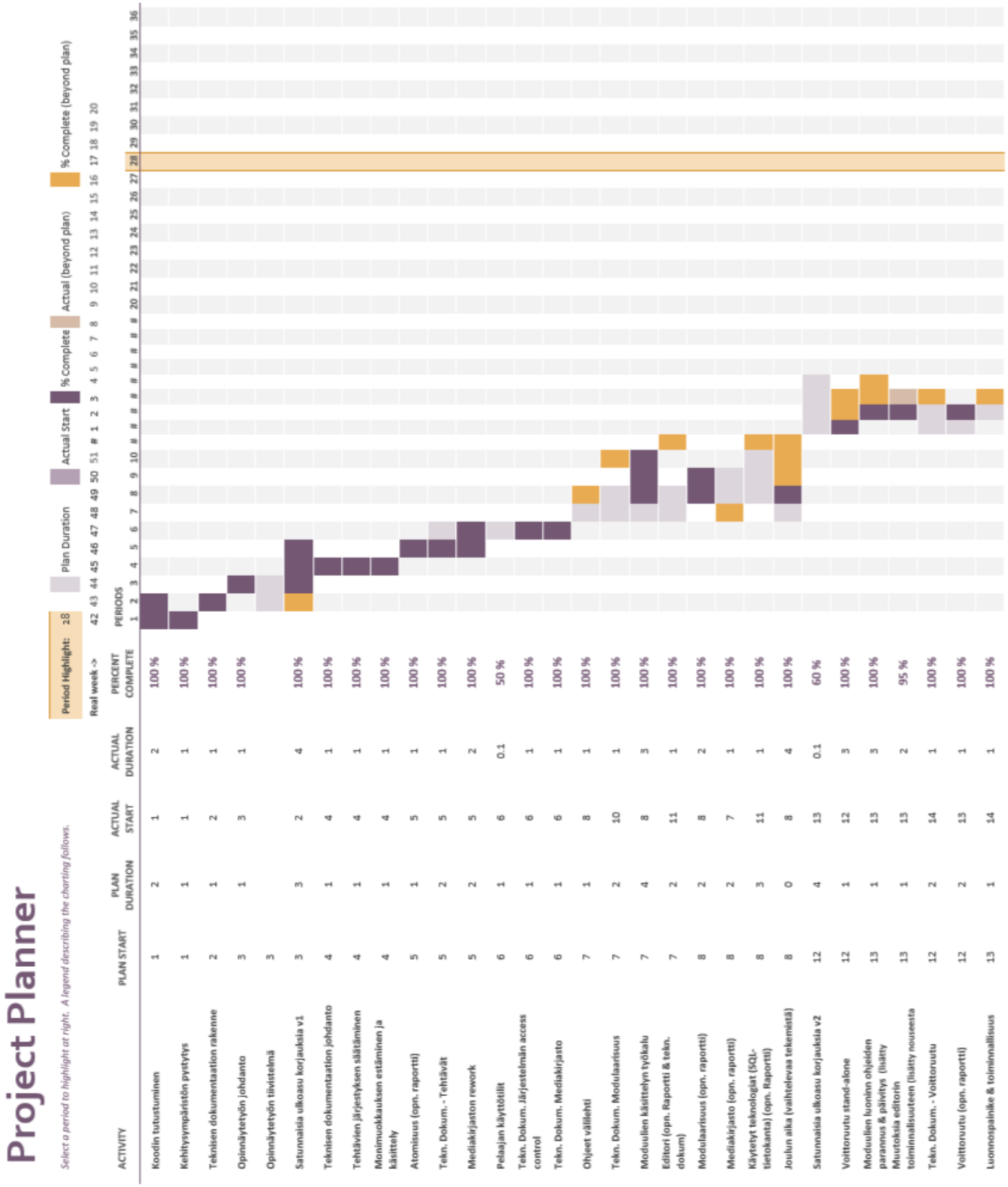
What is data integrity. 2016. Internet-artikkeli tietokannan tekniikoita selittävällä nettisivulla. Viitattu 16.6.2017. http://database.guide/what-is-data-integrity/

What is Symfony. N.d. Symfonyn sivuilla oleva kuvaus mitä Symfony tekee ja miksi se on olemassa. Viitattu 13.3.2018. https://symfony.com/what-is-symfony

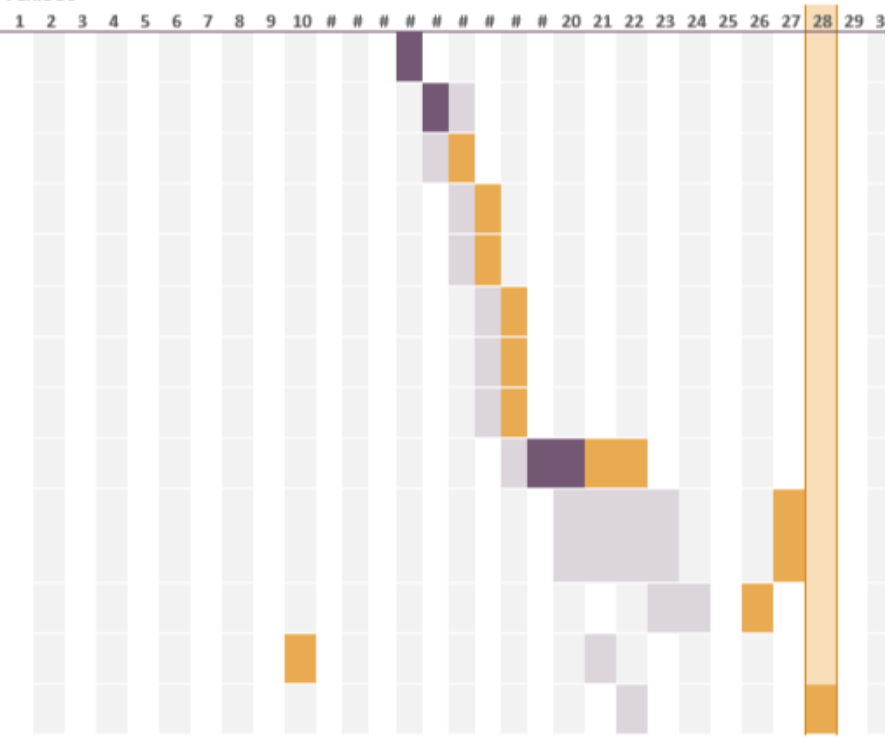
Why should I use a framework. N.d. Symfonyn sivuilla oleva artikkeli ohjelmistokehyksien hyödyistä. Viitattu 16.6.2017. https://symfony.com/why-use-a-framework

Liitteet

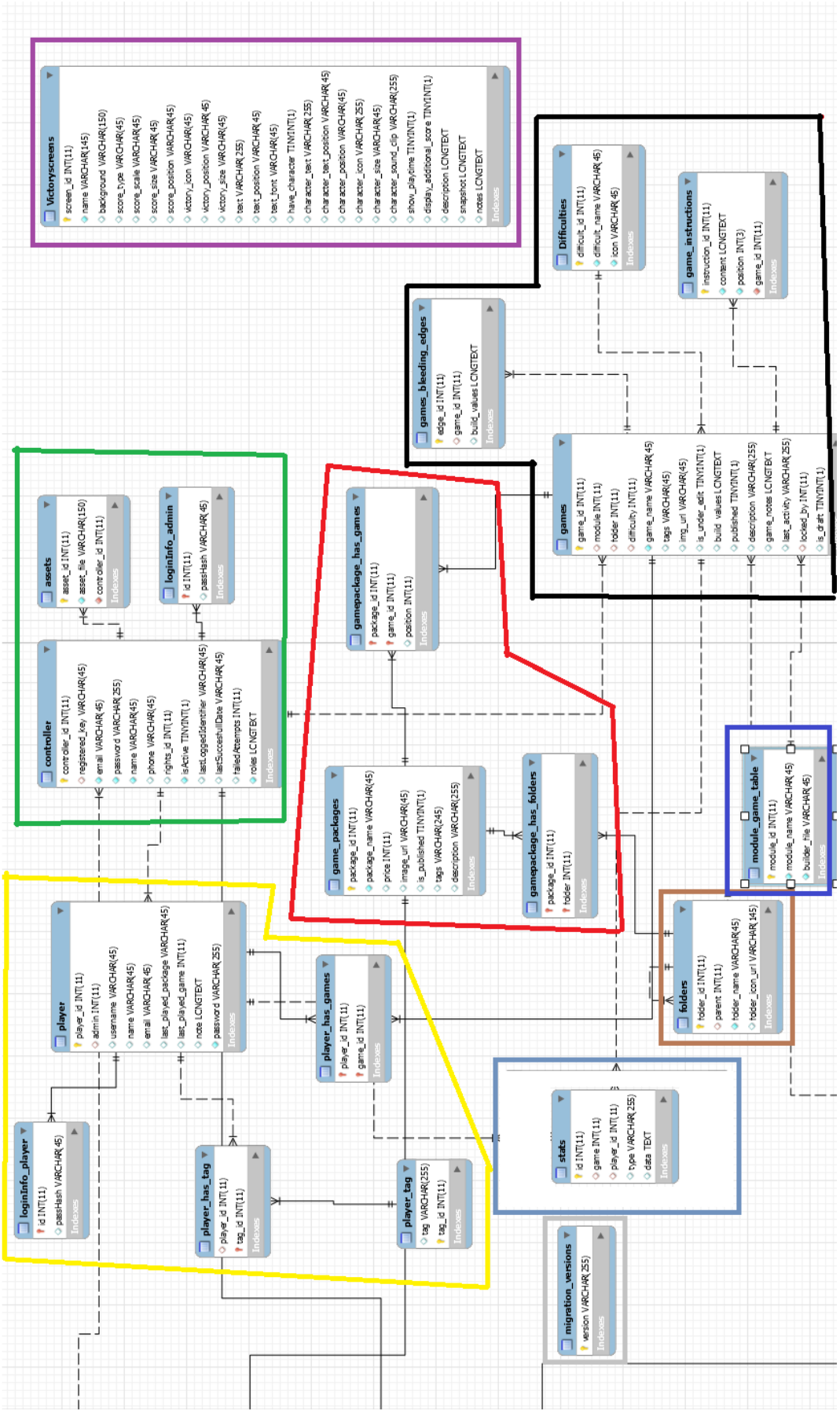
Liite 1. Alustava suunnitelma



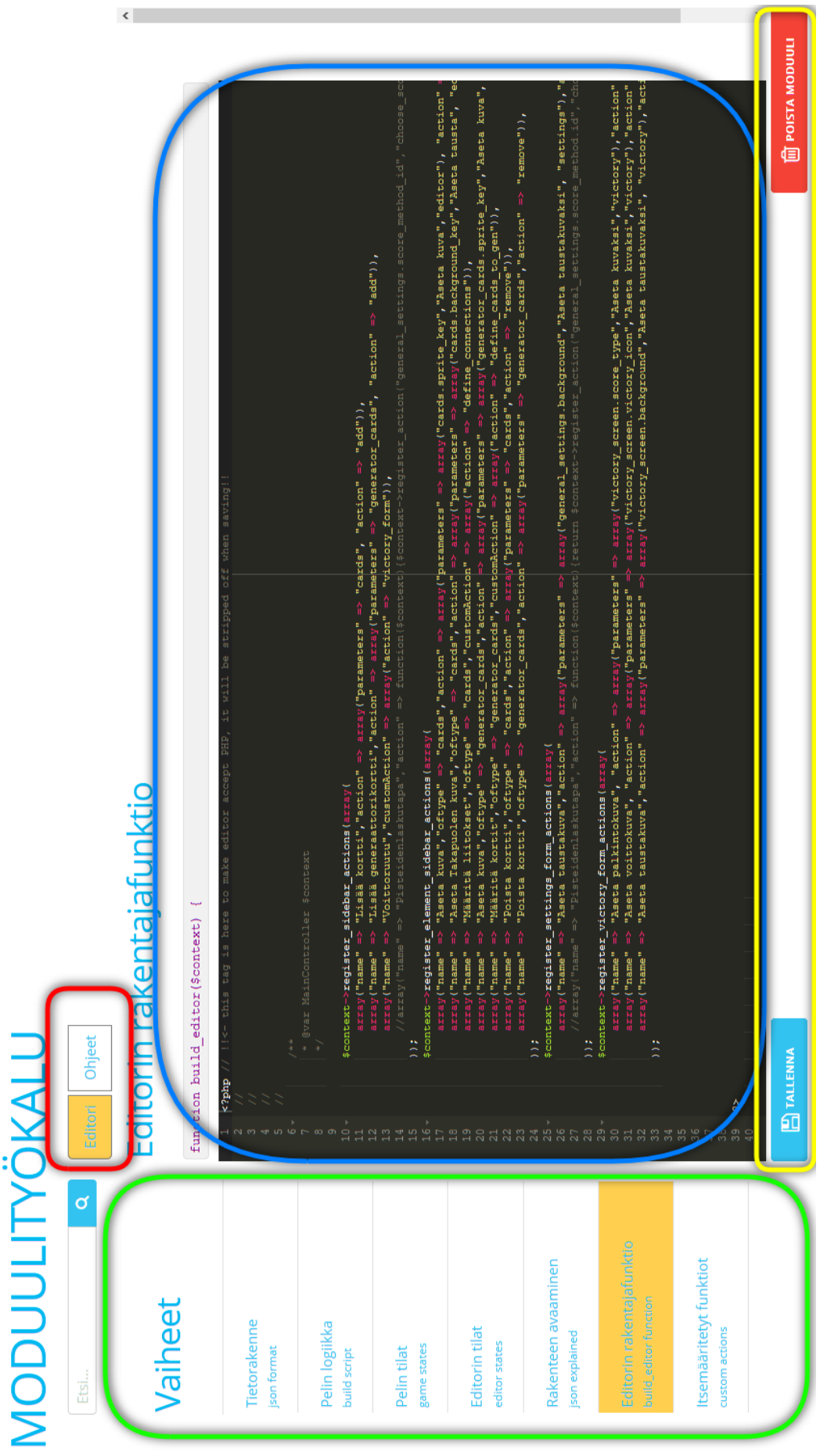
ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Editorin vasen palkki	14	1	14	1	100 %
Versiointi	15	2	15	1	100 %
Kuvaus LUEM-järjestelmästä - yleiskuvaus (opn. raportti)	15	1	16	1	100 %
Ongelmana miten sähköistää oppimateriaalit	16	1	17	1	100 %
Pelien hallinta (opn. raportti)	16	1	17	1	100 %
Tekn. Dokum. -Pelien hallinta	17	1	18	1	100 %
Tehtävien hallinta (opn. raportti)	17	1	18	1	100 %
Pelipuoli (opn. raportti)	17	1	18	1	100 %
Käytetyt teknologiat (Kaikki loput) (opn. Raportti)	18	3	19	4	100 %
Toteutetun ratkaisun soveltuvuus käytännössä (mukaan lukien testaukset)(opn. raportti)	20	4	27	1	100 %
Pohdinta (opn. raportti)	23	2	26	1	100 %
Tekn. Dokum. Moduulien kehittäminen	21	1	10	1	100 %
Tekn. Dokum. tietokantarakenne	22	1	28	1	100 %



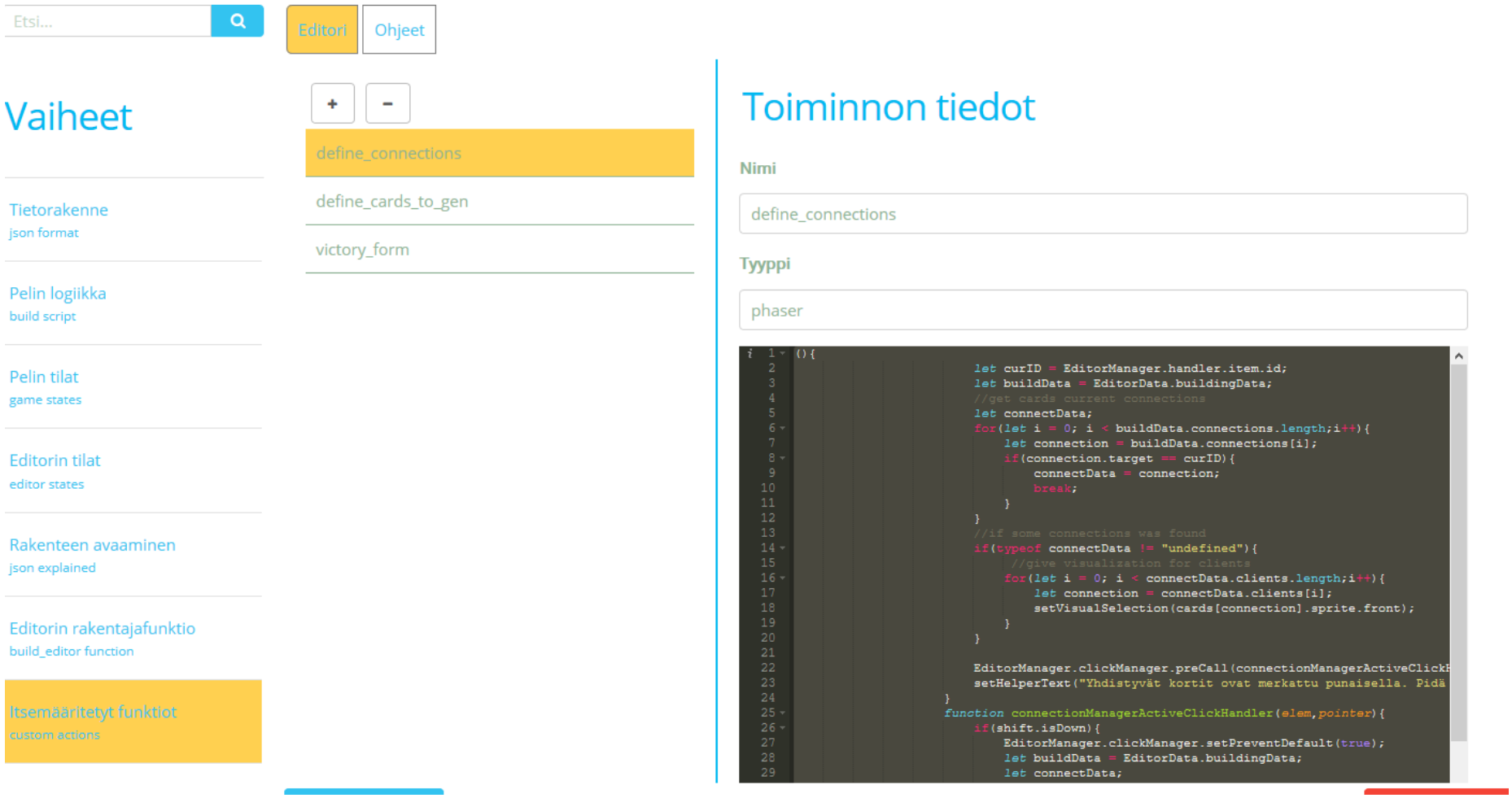
Liite 2. Tietokannan rakenne



Liite 3. Moduulityökalun päänäkymä



Liite 4. Moduulityökalun JSON-työkalu


Etsi...
Editori
Ohjeet
Vaiheet
Tietorakenne
json format
Pelin logiikka
build script
Pelin tilat
game states
Editorin tilat
editor states
Rakenteen avaaminen
json explained
Editorin rakentajafunktio
build_editor function
Itsemääritetyt funktiot
custom actions
+
-
define_connections
define_cards_to_gen
victory_form
Toiminnon tiedot
Nimi
define_connections
Tyyppi
phaser
(){
let curID = EditorManager.handler.item.id;
let buildData = EditorData.buildingData;
//get cards current connections
let connectData;
for(let i = 0; i < buildData.connections.length;i++){
let connection = buildData.connections[i];
if(connection.target == curID){
connectData = connection;
break;
}
}
//if some connections was found
if(typeof connectData != "undefined"){
//give visualization for clients
for(let i = 0; i < connectData.clients.length;i++){
let connection = connectData.clients[i];
setVisualSelection(cards[connection].sprite.front);
}
}
EditorManager.clickManager.preCall(connectionManagerActiveClickH
setHelperText("Yhdistyvät kortit ovat merkattu punaisella. Pidä
}
function connectionManagerActiveClickHandler(elem,pointer){
if(shift.isDown){
EditorManager.clickManager.setPreventDefault(true);
let buildData = EditorData.buildingData;
let connectData;