CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Diseño y desarrollo de una herramienta de programación de laboratorios de biología portátiles: Bioblocks 3.0

Autor: Alejandro García Blanco

Director: Alfonso Rodríguez

MADRID, ENERO 2019

Índice

1. RESUMEN

1.1. RESUMEN (ESPAÑOL)

Bioblocks 3.0 tiene como objetivo ser una plataforma en la que el usuario pueda crear y compartir experimentos biológicos, orientada a un amplio rango de usuarios, desde la formación académica (colegios) hasta la creciente comunidad de biología de Do It Yourself (DIY). Este proyecto de TFG es la continuación de la anterior versión de Bioblocks (Bioblocks 2.0), desarrollada durante el curso 2017-2018 por estudiantes de la facultad y bajo el entorno Bitbloq. Debida a la mala experiencia con el desarrollo usando Bitbloq, tanto con la documentación y el soporte por parte de BQ como con el aspecto técnico de la plataforma, el primer objetivo de este proyecto ha sido buscar una plataforma alternativa sobre la que trabajar o, en su ausencia, diseñar y desarrollar las partes necesarias.

Llevado a cabo por un grupo de 4 alumnos (3 en el curso de los Trabajos de Fin de Grado correspondientes y el otro apoyando durante su Practicum), el proyecto ha tenido dos fases claramente separadas: 1) búsqueda de una solución alternativa a Bitbloq con la que desarrollar y 2) el diseño y programación de las distintas funcionalidades de la aplicación.

Para la primera fase, empecé evaluando la beta de Scratch versión 3, que resultó estar muy orientada a las animaciones de su público objetivo (niños), y cuyo sistema de extensiones era demasiado poco flexible para lo que queríamos usarlo. Buscando alternativas, encontré la plataforma PXT de Microsoft y a esto lo siguió un período de pruebas y conversaciones entre el equipo para determinar si utilizar PXT como la base del proyecto, que resultó en mi propuesta de utilizar un componente de PXT como base, PXT-Blockly (principalmente Blockly con el diseño de Scratch), y desarrollar nosotros tanto un backend como el resto del frontend con una serie de tecnologías, principalmente Vue JS en frontend y Node JS en el servidor.

Durante la segunda fase del proyecto me centré principalmente en la parte de conectar bloques de hardware de Blockly con un renderizado de los mismos y sus conexiones utilizando la librería Paper.JS, así como en el desarrollo de un entorno protegido en el que poder permitir la ejecución de extensiones diseñadas por la comunidad de forma segura, y la creación de una pequeña API para que dichas extensiones puedan añadir bloques a Blockly.

1.2. SUMMARY (ENGLISH)

Bioblocks 3.0's goal is to be a platform where the user can design a share biology experiments, targeting a broad range of users, anywhere from school to the increasingly large Do It Yourself (DIY) community. This final degree project is the continuation of a previous Bioblocks version, 2.0, developed during the 2017-2018 academic year by students of the university on the Bitbloq environment. Due to the overall bad experience with the environment, both in terms of the support received by BQ and in regards to the technical complexity of it, the first goal of the Bioblocks 3.0 project has been to find a different platform or, be it not found, design and develop the necessary parts.

The project has been developed by a group of 4 alumni (3 as part of their respective final degree projects, and one supporting as part of their Practicum), and it has consisted of two clearly separated phases: 1) the search for an alternative to Bitbloq with which to develop the project with, and 2) the design and development of the functionalities of Bioblocks afterwards.

For the first phase, I evaluated Scratch v3, which resulted to be too oriented to the creation of animation and for its target audience (kids), as well as featuring a very inflexible extension system for what we wanted to do. While looking for alternatives, I found Microsoft's PXT platform, which seemed to be a good alternative. What followed was a period of testing and talks within the team to agree on whether to use PXT as the base of the project. This work resulted in my proposal to use PXT-Blockly (a PXT component, a fork of Blockly to add the Scratch design) and develop a backend and the rest of the frontend primarily using Node JS and Vue JS (respectively).

During the second phase I focused specifically on connecting blocks designed in Blockly with canvas rendering based on the Paper.JS library, as well as on designing a sandbox environment in which to securely allow execution of user-provided code for extensions. Alongside this development, I designed a small API so that said user-made extensions can add blocks to Blockly.

2. ESTADO DEL ARTE

El espacio de la biotecnología no es uno nuevo, aunque sí rápidamente creciendo en popularidad, y ya existen soluciones comerciales similares a Bioblocks que han servido tanto de inspiración como de referencia para el proyecto.

La primera sería el Bento lab, un laboratorio biológico portátil orientado tanto a la investigación como a la educación, combinando una centrifugadora pequeña, un ciclador térmico, una unidad para realizar electroforesis basada en gel y un transiluminador, todo en el tamaño similar a un ordenador portátil pequeño, permitiendo así realizar variados experimentos con ADN. Bento responde a un mercado similar al que pretendemos cubrir nosotros, tratando de hacer accesible la experimentación al mayor número de personas, aunque a un coste superior a una posible solución futura basada en Bioblocks y hardware abierto.

Figura 1. Bento Lab (material de su web)

Así mismo, aunque más enfocado en despertar el interés por la ingeniería genética en estudiantes de a partir de 12 años, se encuentra el laboratorio *Amino DNA Playground*, de Amino Bio. En contraste con el Bento lab, Amino basa sus experimentos en la

biofluoresencia, algo comprensible puesto que es mucho más visual para los jóvenes. Amino incluye una guía en la que describe una serie de experimentos posibles.

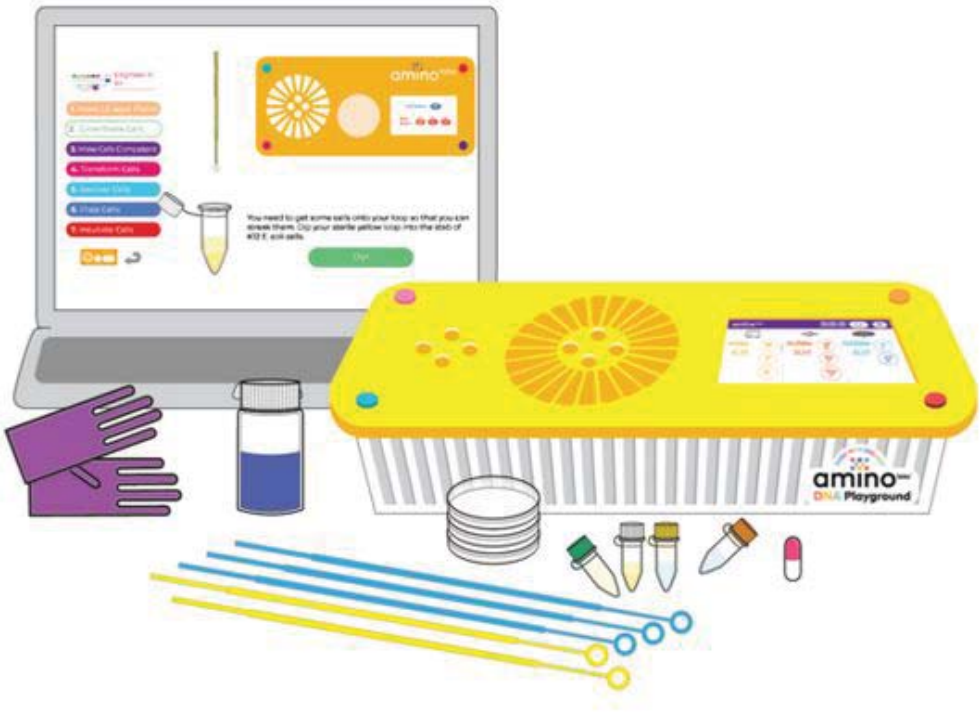

Figura 2. Amino DNA Playground (material de su web)

Al igual que el equipo de Bioblock 2.0 previamente, hemos utilizado como referencia en varias ocasiones las especificaciones del biolab diseñado para el iGEM 2016 por el equipo de la Universidad Politécnica de Valencia, compuesto por un electroporador, un colorímetro, una centrifugadora, una unidad de electroforesis en gel, un agitador con control de temperatura, un fitotrón, un ciclador térmico y un luminómetro, todo controlado por un Arduino Mega (y con varias fuentes de alimentación), y con un coste de menos de 800€, mientras que soluciones comerciales comparables superan los 10.000€. La decisión de utilizar hardware abierto (como es el Arduino Mega) va en consonancia con la filosofía abierta y para todos de nuestro proyecto.

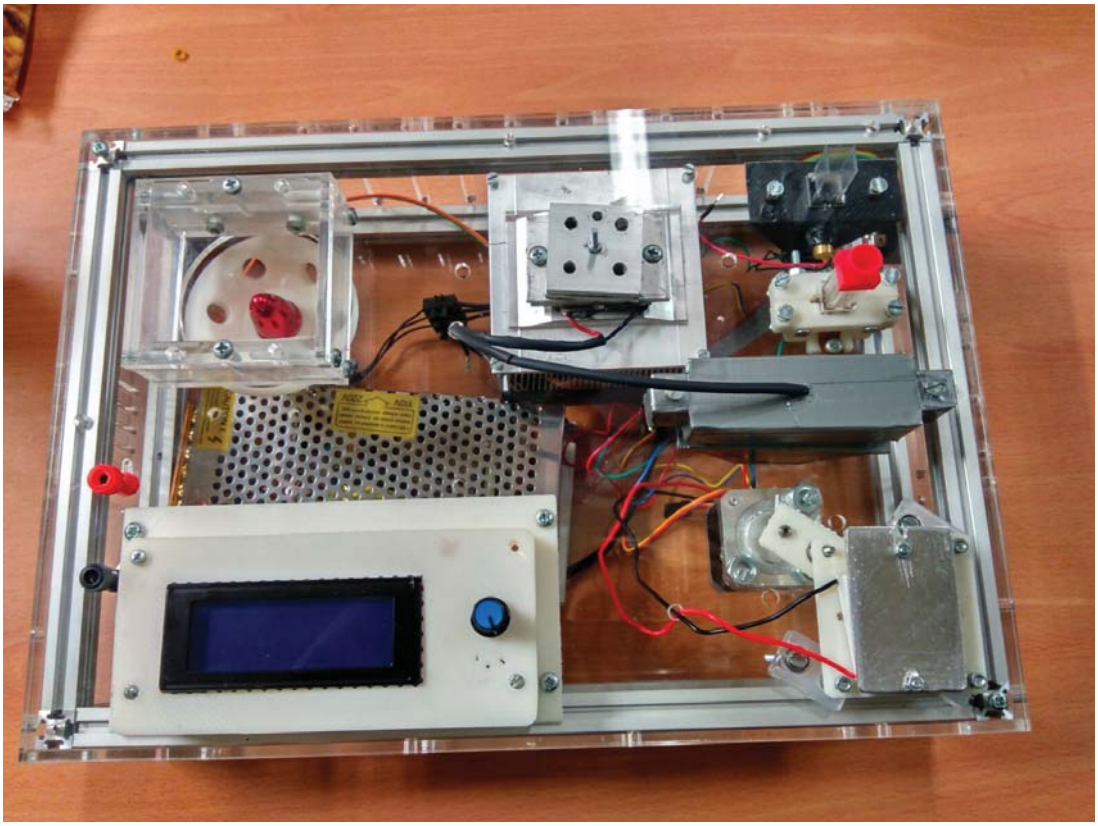

Figura 3. Biolab desarrollado por el equipo de estudiantes de la UPV en iGem 2016

Finalmente, es destacable mencionar a Blockly, que es el estándar de referencia en lo que a programación con bloques se refiere, y que ha llegado al aula en muchos colegios de España y en distintos países del mundo de una manera u otra. En realidad, Blockly no es conocido de por sí, pero es el motor que corre detrás de los exitosos proyectos de Scratch y MakeCode, de Microsoft.

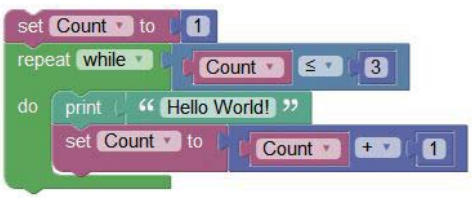

Figura 4. Programación de un “Hello world” en Blockly

Scratch, desarrollado por el MIT ha liberado recientemente su nueva y rediseñada versión 3, está centrado en permitir a los pequeños programar entornos multimedia interactivos (como animaciones, y pequeños juegos o vídeos). Su claro y amigable diseño de los bloques de Blockly ha sido directamente utilizado en el proyecto, aunque a través de la personalización de Blockly de Microsoft.

Por el contrario, Make Code está orientado para estudiantes un poco más mayores, ya la mayoría de los experimentos que se pueden encontrar en su portal están relacionados con la electrónica. Concretamente, el target "Maker" ha sido valorado en el proyecto como una posible base sobre la que trabajar. Aunque que por distintas razones no acabamos utilizándolo como tal, sí que fue fuente de inspiración muy significativa.

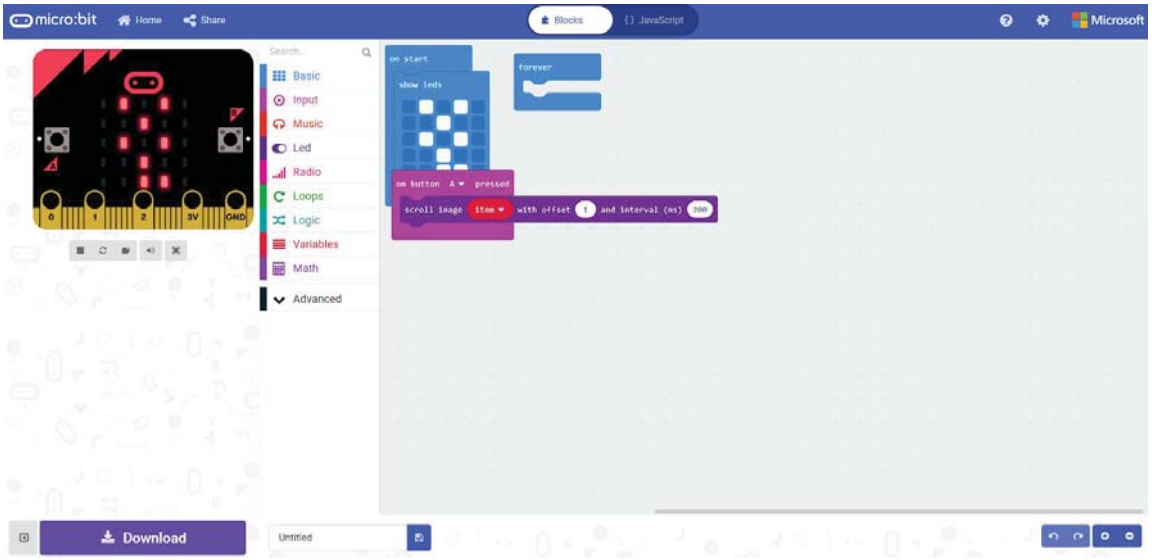

Figura 5. Uno de los target de MakeCode (microbit), con código de ejemplo

3. OBJETIVOS DEL PROYECTO

A continuación se describen los distintos objetivos de Bioblocks 3.0, tanto globales como los directamente relacionados con este trabajo de fin de grado (algunos ya mencionados brevemente en las secciones anteriores).

El primer y principal objetivo es producir un software libre y de código abierto que sirva tanto a estudiantes como a la comunidad DIY y soporte componentes de hardware abiertos, así como unificar la manera de describir tanto los experimentos biológicos como el hardware utilizado para ello, con la esperanza de facilitar e incentivar compartir este tipo de experimentos.

En términos de organización social, los usuarios deberían poder crear un proyecto privado, guardado en la nube, o compartirlo públicamente, así como ser parte de grupos de usuarios.

Los perfiles de usuarios a los que se orienta el proyecto son los siguientes: estudiantes en colegios, aficionados de la biotecnología, y graduados en biología/biotecnología, así como estudiantes de estas carreras. En base a esto, no podemos esperar que todos los usuarios tengan ningún tipo de conocimientos de programación, con lo que el proyecto debe hacer principalmente uso de herramientas visuales. Se puede encontrar la lista completa de perfiles de usuario que recopilé en un anexo.

Así mismo, debido al heterogéneo entorno que es la comunidad Do It Yourself, y a que este ecosistema está en continua evolución, con lo que cualquier set de componentes que diseñemos nosotros hoy puede quedarse rápidamente desactualizado, el sistema debe ser dinámico y permitir a los usuarios la inclusión de nuevos componentes de hardware. Para esta tarea, no obstante, los usuarios sí que deberán saber programación, puesto que la traducción del experimento a código Arduino para permitir ejecutarla en una placa Arduino es algo específico a la pieza de hardware y no se puede generalizar o abstraer significativamente.

En el aspecto técnico, y al estar basado en la experiencia negativa con Bitbloq de Bioblocks 2.0, el objetivo en esta área es encontrar otra plataforma en la que poder llevar a cabo los objetivos establecidos hasta aquí.

4. DESARROLLO

4.1. ANÁLISIS DE PLATAFORMAS ALTERNATIVAS A BITBLOQ

Bitbloq, pese a tener una interfaz de usuario bastante bonita, causó numerosos problemas durante el desarrollo de Bioblocks 2.0, por un lado debido a la falta de documentación y por otro a la falta de soporte por parte de BQ. En base a la actividad en el repositorio de Github, parece ser una aplicación mantenida por una o dos personas de cuando en cuando. Además de su compleja instalación, Bitbloq utiliza tecnologías ligeramente desfasadas, como Angular 1.x (o AngularJS, como se llama ahora), librería que hoy en día Google ya ha dejado desarrollar y dejará de tener cualquier tipo de soporte en dos años.

En resumidas cuentas, después de varios intentos problemáticos de montar Bitbloq nosotros y por la mala experiencia en la que resultó para Bioblocks 2.0, sin duda descartamos Bitbloq como plataforma. La primera alternativa sugerida en una reunión fue Scratch, plataforma de código abierto y cuya versión 3.0 estaba en beta en su momento (ha sido recientemente liberada al público, el 2 de Enero de 2019), y tiene características relativamente similares a Bitbloq, tales como un sistema de usuarios, una lista de bloques, un área donde pintar el hardware resultante.

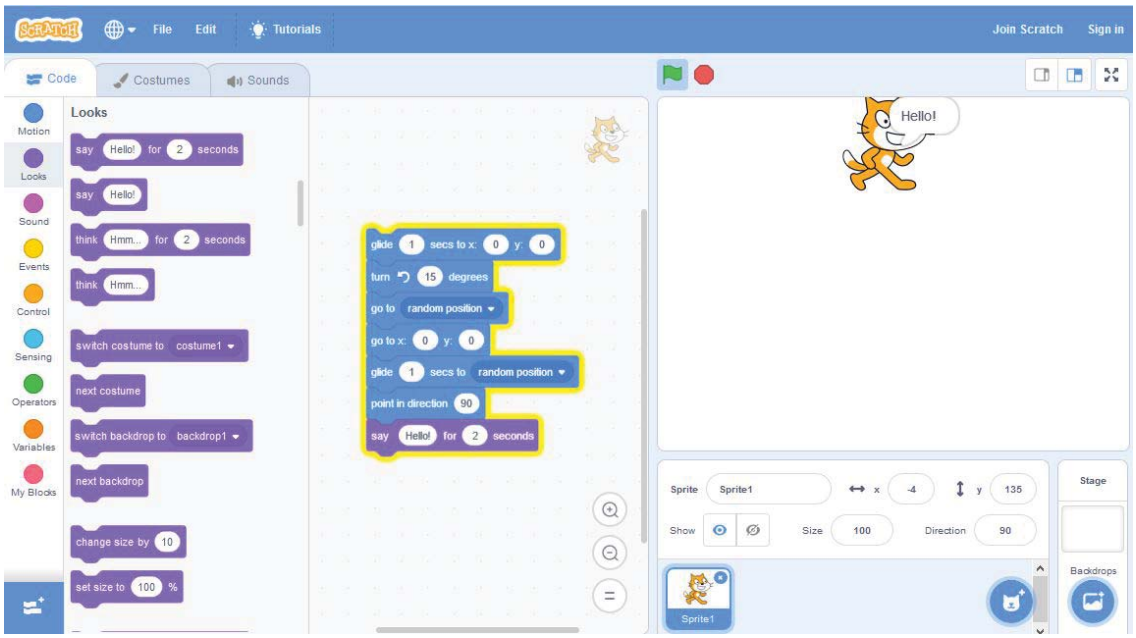

Figura 6. Versión 3 de Scratch

No obstante, la capacidad de extender el área de trabajo de Scratch 3 con las extensiones (o ScratchX), pero resultaron ser demasiado simples para lo que queríamos hacer, siendo capaces principalmente de añadir bloques, pero no cambiar el renderizado del resto de la página. Adicionalmente, Scratch incluye varios sistemas que nos resultarían completamente innecesarios y complicarían el desarrollo, como audio o animaciones. Por si todo esto no fuera poco, el backend de Scratch no es de código abierto, dejando claro de forma definitiva que Scratch no era una buena alternativa. No obstante, incluyo en los anexos detalles sobre cómo montar los repositorios de Scratch por si fuera de interés.

Descartado Scratch también y en busca de alternativas, encontré un proyecto prometedor de Microsoft: PXT (Programming eXperience Toolkit). Una vez presentado al equipo, estuvimos de acuerdo en que en efecto parecía una buena opción, y que habría que estudiarlo. Poco después, Alfonso encontró Maker, un target de PXT en MakeCode.

Inmediatamente Maker fue de interés: disponía de una parte en la que se renderizaba el hardware y sus conexiones, otra en la que se podía declarar el experimento, y un sistema de extensiones (accesible desde la sección *Advanced*, ver **Figura 7**).

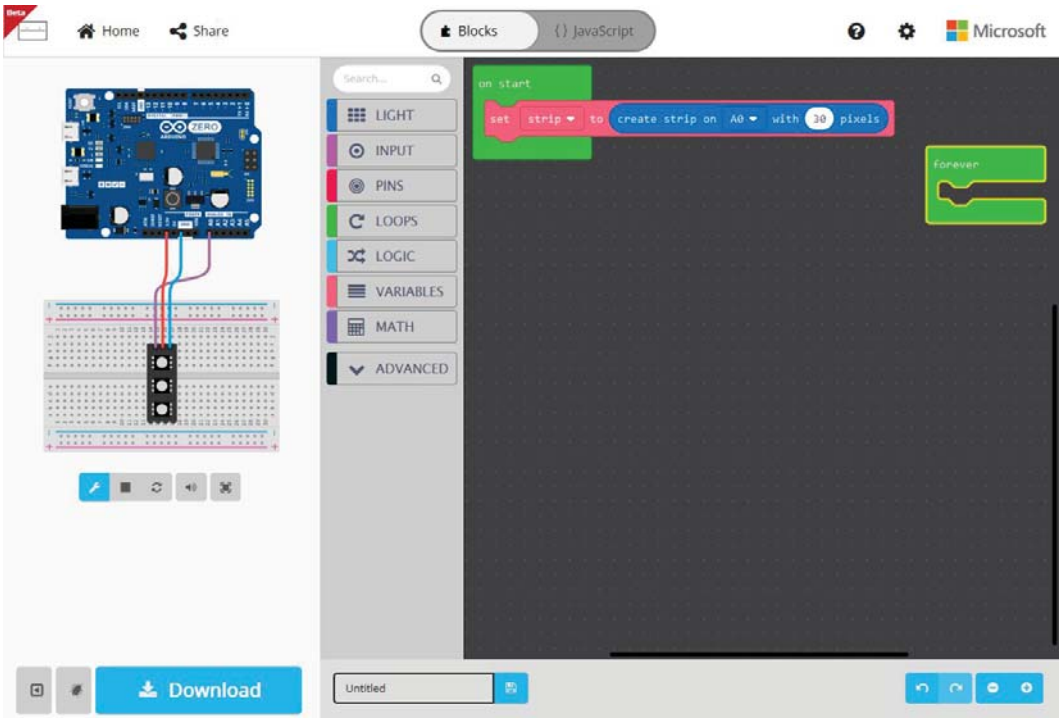

Figura 7. Maker MakeCode (Beta), con una tira LED de ejemplo

Durante varias semanas me dediqué a navegar el código y a entender el funcionamiento de las partes de PXT, principalmente el sistema de extensiones y el simulador, así como concretamente el *target* maker. Después de esto, mi conclusión fue que no deberíamos utilizar PXT, aunque sí utilizar uno de sus componentes, PXT-Blockly como base. El resto de esta sección se dedicará a listar los posibles problemas que encontré y los fuertes que considero nos aportaría PXT, pero en mi opinión (con la que el resto del equipo coincidió), las desventajas de usar PXT superan a las ventajas.

Empezaré por las ventajas, que sin duda hay unas cuantas. Para empezar, PXT es un framework maduro usado por una compañía del calibre de Microsoft, que ya tiene un diseño gráfico claro que podemos seguir, que tiene un sistema de extensiones ya diseñado, y el target Maker tiene un simulador a la izquierda en el que podríamos basar nuestro renderizado de los componentes hardware.

PXT tiene también un sistema de conversión de bloques a código Javascript, una herramienta que permite *flashear* la placa Arduino, e incluye un mecanismo para guardar y restaurar proyectos (aunque basado en local, no en la nube). Como pequeño plus adicional, desarrolladores que conozcan PXT podrían en teoría trabajar fácilmente con Bioblocks.

No obstante, el simulador que provee *Maker* se nos quedaría pequeño para el uso que queremos darle, con lo que habría que cambiar el diseño de la página y el objetivo de dicho simulador, puesto que no queremos simular nada, solo pintar. De hecho, para llegar al nivel de personalización que queremos, tendríamos que o bien hacer cosas no soportadas desde las extensiones (que además funcionan a base de *whitelisting* de repositorios de Github), o bien un fork de PXT, que nos haría divergir y para retener a desarrolladores del entorno PXT, nos bastaría de forma general usar tecnologías similares. Así mismo, PXT es un software muy complejo con un simulador potente, con debugging Typescript, animaciones, sonido, un editor de código que refleja los cambios en los bloques, y más, todas características completamente innecesarias y que se interpondrían entre nosotros y el producto objetivo. Añadiendo a todo esto, PXT carece de un backend open source (y tampoco está especialmente preparado para uno), y utiliza los servidores de Microsoft para compilar código, lo que no nos da mucha independencia.

En definitiva, PXT es buen software, pero basarse en PXT-Blockly es más interesante, permitiéndonos control casi completo sobre el producto final y manteniendo el diseño de bloques de Scratch, superior al que provee Blockly por defecto, que era una de las intenciones.

4.2. TECNOLOGÍAS Y FRAMEWORKS INVOLUCRADOS

Por un lado, las tecnologías principales involucradas en el proyecto Bioblocks 3.0 son HTML, CSS y Javascript, la tríada clásica para las aplicaciones web; el framework Vue, que permite diseñar una página de forma imperativa en función de los datos; el framework de diseño Element UI para Vue, que provee una serie de elementos prediseñados; Blockly, por supuesto, para la parte de programación por bloques; Git, para el versionado del código y su publicación en Github; Node.JS para el servidor, y concretamente Express para proveer la API del backend; MySQL como BBDD, para el almacenamiento de datos; y Arduino (aunque principalmente a futuro), para la conversión. Aunque no es exactamente parte del stack actual, también se utilizaría NGINX como proxy inverso en un entorno de producción para poder definir una Content Security Policy. De estas tecnologías, únicamente no he tenido ninguna interacción directa durante mi trabajo de fin de grado con Express y MySQL, aunque respecto a esta última, sí he colaborado en el diseño de la base de datos.

Por otro lado, en el análisis de las distintas herramientas he tenido además contacto en mayor o menor medida con: Gulp, un framework para ejecutar tareas de Node.JS; con Python, utilizado en general para scripts de compilación de las herramientas; Typescript, en uso en la mayoría del código interno de PXT, y C++, en el caso de PXT y sus extensiones.

4.3. DISEÑO DE ALTO NIVEL DE BIOBLOCKS 3.0

Una vez elegida la base de PXT-Blockly, necesitábamos diseñar un sistema que incluyese todos los demás componentes necesarios para poder desarrollar las funcionalidades deseadas (un servidor, base de datos, etc.). Mi propuesta original para la arquitectura del sistema fue la siguiente:

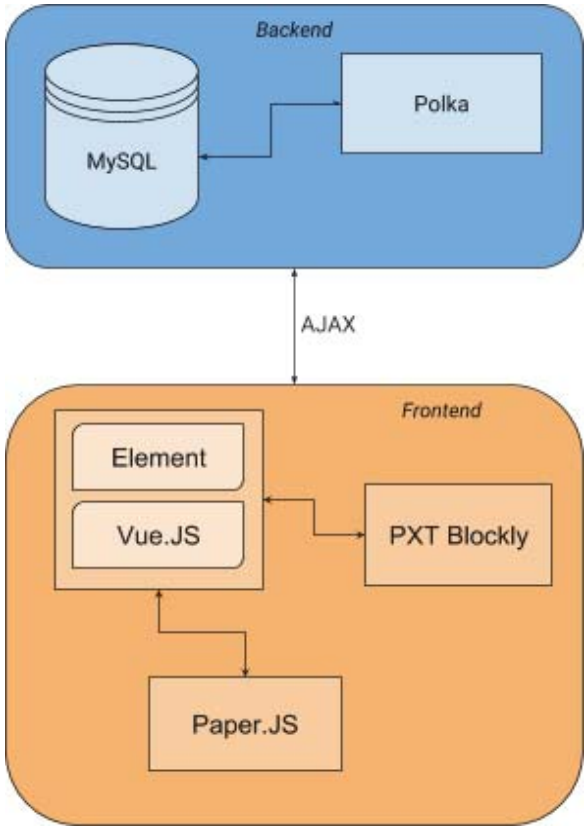

Figura 8. Componentes de mi propuesta original de arquitectura

En este diseño, el *frontend* sería una SPA (*Single Page Application*), que, aparte de ser una tendencia actual en el mundo web, está justificada en este caso: la experiencia del usuario sería lo más fluída posible, y además se trata de una aplicación con un número limitado de vistas. También valoré la posibilidad de una HPA (*Hybrid Page Application*), pero no ganábamos mucho teniendo solo algunas páginas servidas por el servidor más allá de separar la lógica, con lo que la única alternativa real era la más clásica MPA (*Multi Page Application*). En una MPA, el servidor tiene la mayor parte de la lógica de la aplicación y

normalmente también el procesado de los templates, y en nuestro caso esto tampoco presenta un especial problema - es una alternativa perfectamente válida, que goza de menor popularidad en los últimos años por el fuerte tirón de los frameworks React y Vue.

En este caso, acabé sugiriendo una SPA siguiendo las tendencias de desarrollo recientes y puesto que Bitbloq utilizaba React, que es otro framework de SPA. El equipo estuvo de acuerdo con esta sugerencia y con el uso de Vue en vez de React, principalmente basado en que en mi (muy limitada) experiencia con ambos, Vue tiene una curva de aprendizaje mucho menos pronunciada que React, y sigue más el modelo clásico web de HTML + CSS + JS, frente a la obsesión de React de integrar todo en Javascript, hasta los templates.

El resto del *frontend* estaría compuesto por PXT Blockly, que proveería la lógica de programación con bloques; Paper.JS, un framework para el renderizado dinámico en canvas de HTML5, que se encargaría de pintar la parte hardware; y Element, un framework para Vue que aporta una serie de componentes Vue con un diseño gráfico elegante, eliminando así la necesidad de aportar el CSS de estos nosotros mismos. La comunicación con el servidor se llevaría acabo a través de AJAX (no hay muchas alternativas en el contexto de un navegador).

El *backend* constaría principalmente de MySQL y Polka. MySQL es una base de datos relacional, que pese a no ser la usada por Bioblocks 2.0 (Bitbloq) parecía la más lógica para nuestro caso de uso, tanto por ser un tipo de base de datos familiar a todos los componentes del equipo (se imparte en el grado) como por el tipo de datos que deseamos guardar en esta: usuarios, grupos, proyectos, permisos… todos ellos inherentemente relacionales. Por su parte, Polka es un framework muy ligero y súper rápido para el diseño de APIs, que es principalmente todo lo que necesita hacer el servidor en el caso de una SPA, personalmente hice varias pruebas con Polka y en efecto cumplía sus promesas, además de ser muy similar a Express (el framework más popular para manejo de peticiones en el servidor).

Puesto que todos estuvimos de acuerdo, creé unos proyectos básicos con los componentes mencionados en Github (bioblocks-backend y bioblocks-browser) para que pudiésemos empezar a trabajar en la aplicación. No obstante, un par de semanas más tarde, el compañero que se encargaba principalmente del backend comentó que, puesto que él ya había realizado algún proyecto con Express y estaba teniendo algunos problemas con la integración de un middleware en Polka, iba a cambiar a Express. Ya que ambos eran comparables, por mi parte no hubo problema.

Con este pequeño cambio, la arquitectura finalmente quedó como sigue:

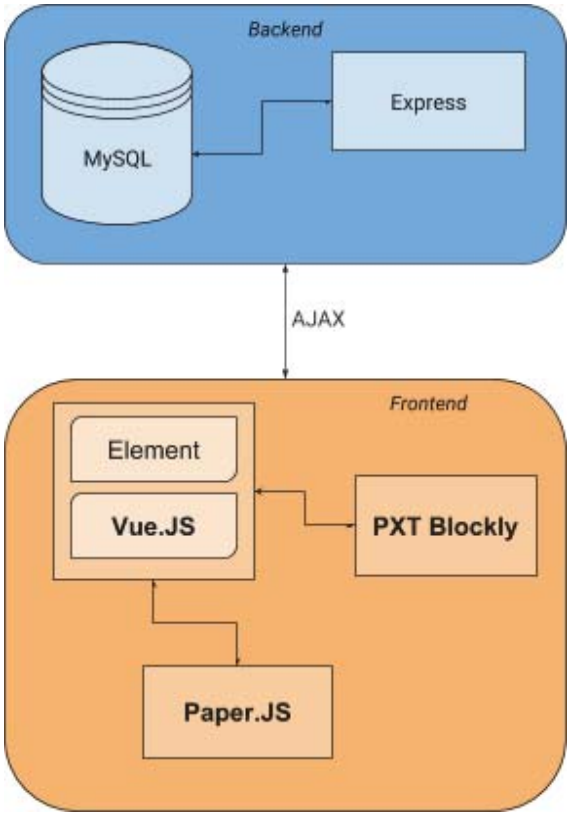

Figura 9. Componentes de la arquitectura final. En negrita, en los que me centré.

De estos componentes, en la figura están marcados en negrita aquellos en los que me centré para mis tareas durante el resto del trabajo de fin de grado: el diseño de un renderizado de los bloques de hardware que forman parte del experimento y sus conexiones, y la creación de un entorno seguro para permitir la inclusión de extensiones de usuario. Estas dos tareas se centran las dos subsecciones siguientes de esta memoria.

Cabe destacar que, aunque el componente MySQL no está marcado, sí que colaboré en el proceso del diseño de la base de datos, además de ayudar a mis compañeros con dudas/problemas con las tecnologías en uso, principalmente sobre Vue y Git. Conforme avanzó el proyecto, también dediqué varios días a echar una mano con cosas relacionadas con Blockly, como los generadores de código de Blockly o la adición de un nuevo bloque.

4.4. DESCRIPCIÓN Y VISUALIZACIÓN DE HARDWARE

La cuestión de cómo definir el hardware varió varias veces durante el proyecto, e inicialmente teníamos la idea de definir el hardware en una sección aparte, de forma similar a como se hace en Bitbloq, conectando los componentes manualmente. Esta idea no la llegamos a prototipar y se acabó descartando, puesto que desde que dimos con MakeCode Maker, esa pasó a ser nuestra principal inspiración, y consideramos que era una mejor idea definir el hardware mediante bloques (como hace Maker), siendo consistentes así también con el experimento. En base a eso, mi idea de implementación era la siguiente: a fin de no mezclar experimentos y definición de hardware, existirían dos *workspaces* de Blockly, uno para definir el hardware y otro para definir simplemente los pasos del experimento. Con este objetivo, traté de recrear unos bloques similares a los de Maker en Blockly, que resultaron en lo siguiente:

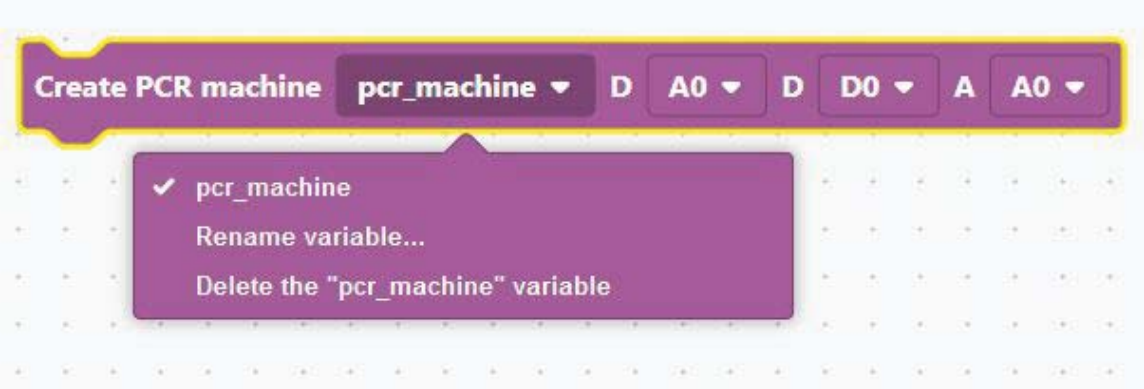

Figura 10. Primer prototipo de un bloque de creación de hardware una PCR

Desgraciadamente, me topé con varios problemas: En la parte técnica, el manejo de variables de Blockly no era tan simple como pensaba, además de tener que duplicar y sincronizar las mismas entre los dos *workspaces*. Desde el punto de vista de experiencia de usuario, era también molesto tener que asegurarse de que la PCR definida en la parte de hardware era la PCR exactamente definida en la parte de experimento, al tener que seleccionar la misma variable manualmente en ambas vistas. Por estas razones, eliminé las variables y la separación entre un workspace de software y un workspace de hardware, con la idea de unificar la definición de hardware como la definición del experimento, como se puede observar en la siguiente figura:

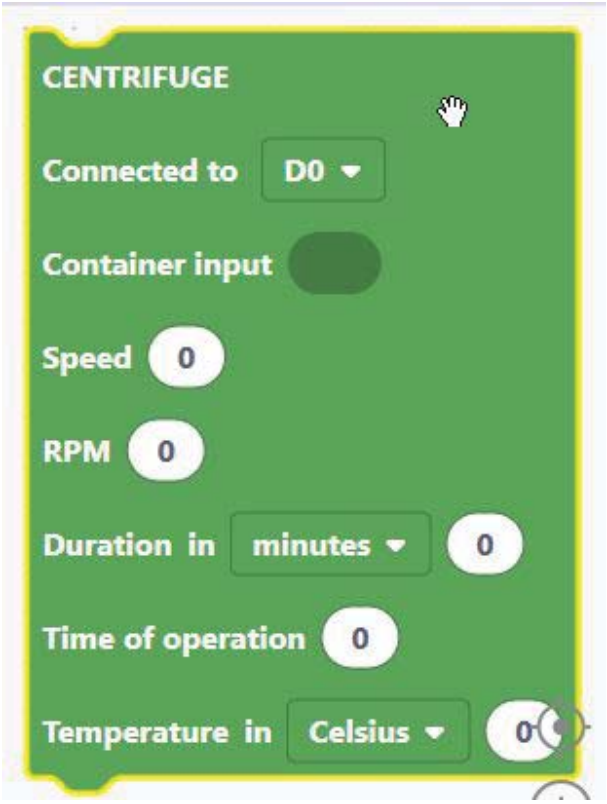

Figura 11. Descripción de conexiones hardware integrada en el bloque

A partir de aquí, diseñé un pequeño prototipo de renderizado con Paper.JS, cuyo funcionamiento se describe a continuación: Se creaba un canvas sobre el que Paper.JS puediese pintar, y se rellenaba con una imagen de un Arduino Mega en el centro, sobre la cual manualmente coloqué distintos puntos de conexión invisibles correspondientes a las coordenadas de los pines (se han pintado de colores rojo y verde en la figura).

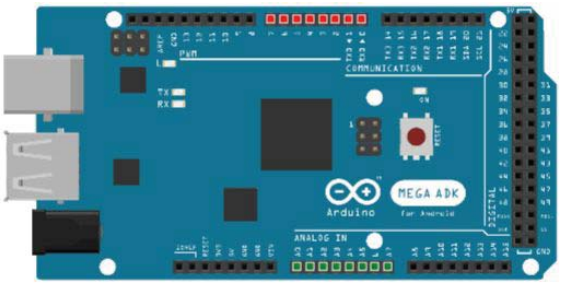

Figura 12. Arduino Mega renderizado junto con una lista de pines

Desgraciadamente, esto no solo fue un proceso tedioso, sino que además había que rehacerlo completamente cada vez que se cambiaba la escala de la imagen, algo que tuve que hacer varias veces con cambios del tamaño del área de trabajo por distintas razones. Seguramente una mejor opción habría sido utilizar un SVG de el Arduino Mega, pero aunque encontré numerosos SVG para Arduino Uno, el único SVG para Mega que encontré funcionaba perfectamente en Google Chrome pero presentaba problemas de renderizado en Firefox (ver figura).

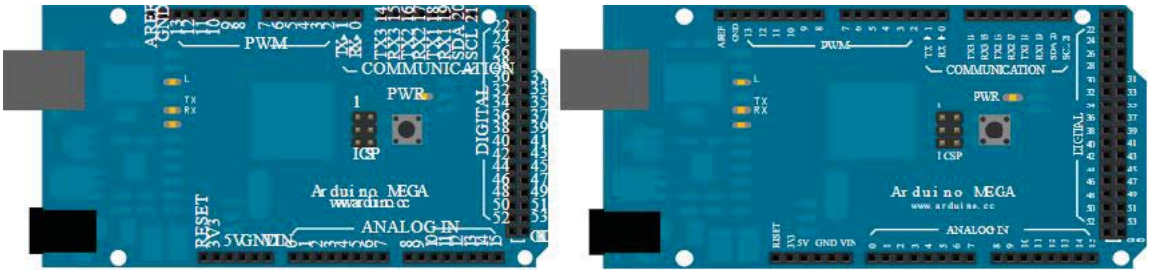

Figura 13. Renderizado del SVG en Firefox (izquierda) frente a Chrome (derecha)

Dejando estos problemas de lado, Blockly provee un método para su *workspace*, `workspace.addChangeListener`, que nos permite escuchar y recibir eventos sobre los cambios en el sistema. Utilizando esta funcionalidad, diseñé una función para pintar un nuevo rectángulo en el canvas al recibir un evento CREATE, identificado por su ID de bloque (que es única para un workspace dado). En función al estado del campo de pin del bloque en Blockly, se generaba un Path (objeto de PaperJS que representa un camino en el canvas), con origen en el centro inferior del rectángulo, y destino en el pin dado. Al recibir un evento CHANGE de un bloque registrado en la lista de componentes hardware, se actualizaba el destino del Path de forma que apuntase a otro de los nodos invisibles del Arduino Mega. El prototipo inicial se puede observar en la Figura 14.

Poco después de esta implementación, caí en la complejidad de tener que mantener el estado de todos los componentes hardware y sus conexiones, que complicaría muy significativamente la lógica en el caso de querer mover un bloque por cualquier razón, y opté por reescribir la lógica de forma más imperativa, esto es: los nuevos bloques y cada actualización de los mismos actualizan un objeto intermedio que simplemente describe las conexiones actuales, y con cada cambio se vuelven a renderizar **todos** los componentes.

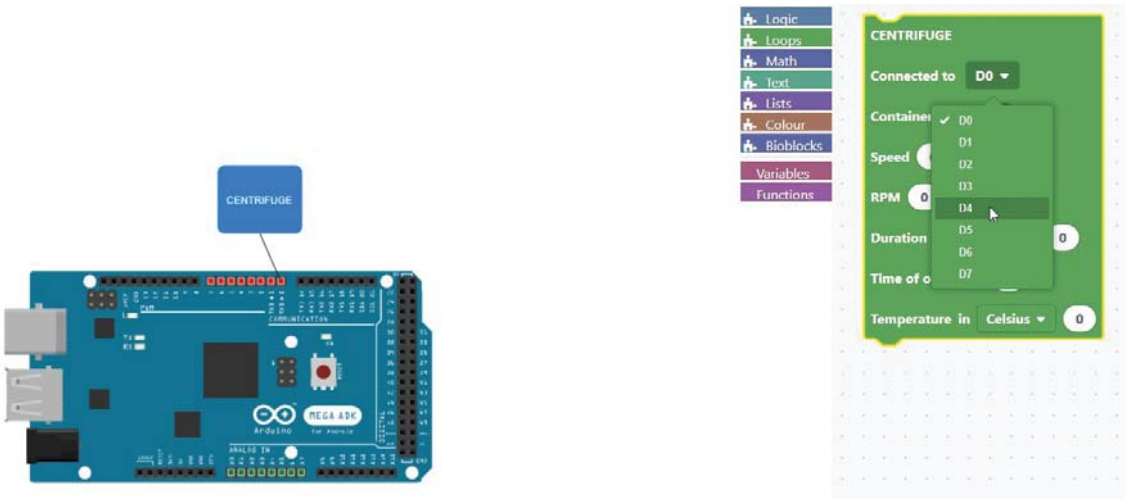

Figura 14. Primer prototipo de visualización de hardware

En esta figura también se puede observar que se encuentran tanto hardware como software en la misma vista, permitiendo así editar el componente y ver el resultado en tiempo real. Aunque estaba bastante contento con esta distribución inicialmente, las vistas se separaron completamente en versiones siguientes, puesto que por un lado cambiamos los componentes Blockly a una distribución horizontal, y por otro lado esta distribución era muy poco práctica en pantallas más pequeñas (como las de un tablet).

Existen varios problemas con esta versión, que no he llegado a resolver en el trabajo de fin de grado, los más importantes son: 1) identificación de componentes duplicados (por ejemplo, si hay dos centrifugadoras, deberían pintarse de forma distinta y/o marcarse de alguna manera), 2) distribución de los componentes en el entorno (por ejemplo, si hay 7 componentes, ¿cómo se distribuyen espacialmente?), 3) inflexibilidad de la solución (por ejemplo, no permitiría cambiar la placa fácilmente) y 4) definición de las “dependencias” de cada componente (por ejemplo, una fuente de alimentación, si fuese el caso). En el capítulo de líneas futuras de trabajo exploro ideas sobre posibles mejoras que resuelvan estos problemas.

4.5. EXTENSIONES DE HARDWARE: ENTORNO

Las extensiones de usuario están orientadas a permitir que los usuarios añadan nuevos componentes hardware, debido al contexto heterogéneo y cambiante de la biotecnología mencionado en previos capítulos. Para conseguir este objetivo hay que permitir a las extensiones que incluyan código que convierta el bloque en código (en el fondo, la conversión no es más que una forma de compilación de la programación visual a Arduino, y la compilación requiere un compilador, parte del cual deberán proveer los usuarios).

Sin embargo, el código Javascript de terceros debe ser ejecutado de forma aislada al resto de la aplicación, porque de lo contrario existen una serie de riesgos, que permitirían al atacante (consideremos atacante al usuario que provee código malicioso en su extensión) hacer cosas tales como: reemplazar el sitio completamente, de forma que pareciese otro; redireccionar el sitio web a otro similar al nuestro o otro sitio conocido (p.ej Google), con el objetivo de robar información a los usuarios; mostrar los clásicos pop ups de "*¡¡Se ha detectado un virus en tu ordenador!!*" y ofrecer descargas de virus para "solucionarlo"; o minar criptomonedas.

Tanto Scratch como PXT solventan este problema concreto ofreciendo una lista aprobada manualmente de extensiones, pero no es una buena solución práctica para nuestro problema ya que ralentizaría el proceso de creación de extensiones y nos obligaría a tener un equipo de moderación de extensiones cualificado (o de lo contrario, el aprobado manual no serviría).

Así pues, mi primer objetivo fue buscar una manera de crear un entorno en el que ejecutar código de forma segura, una tarea algo complicada dado que los navegadores normalmente no están pensados para estas tareas. Lo primero que exploré fueron librerías que ofreciesen la funcionalidad, de las cuales las más destacables son Caja (de Google), y jailed. Jailed no parecía funcionar bien en mis pruebas, ni siquiera las más simples siguiendo el README del repositorio. Entre estos problemas y que el soporte a la librería parecía un poco abandonado si nos guiamos por la actividad en su repositorio Github, acabé descartándolo. La alternativa de Google, por otro lado, era un paradigma completamente distinto, que requería un servidor adicional y un flujo más intrincado, en el que se alteraba el código en el servidor para limitar las funcionalidades del código. Unas pequeñas pruebas con PXT Blockly revelaron posibles conflictos entre la librería y Caja, y decidí también descartarla en busca de mejores alternativas.

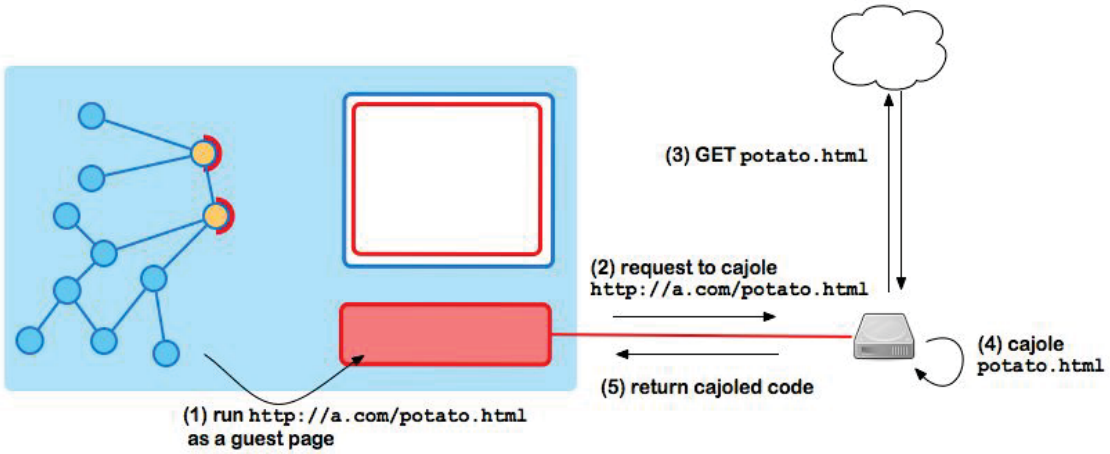

Figura 15. Funcionamiento de Caja (Google)

Descartadas esas dos alternativas, y a pesar de la existencia de otras extensiones menos populares, decidí dejar de buscar librerías, puesto que en temas de seguridad es mejor no utilizar soluciones poco probadas.

Por lo tanto, lo único que quedaba por probar era aplicar todas las medidas posibles que ofrezca el navegador para aislar código. Lo que más se acerca a esto son los iframes, elementos de la página que permiten embeber otras páginas y que ofrecen una interfaz que permite la comunicación bidireccional con las mismas, así como mitigaciones de prácticamente todos los problemas cuando se habilita su modo *sandbox*. No obstante, esto tiene dos problemas: las extensiones no son parte de **otra** página, y el modo sandbox es demasiado restrictivo. Afortunadamente los navegadores permiten reducir las restricciones, pero dar los permisos que necesitamos al iframe mantiéndolo en el mismo origen también nos expone a que un usuario malicioso pueda "escapar" del iframe, derrotando nuestras medidas de seguridad.

La solución final es el resultado de la mezcla de tres mecanismos de seguridad distintos: Primero, tanto blockly como las extensiones de usuario se tienen que servir desde **otro origen** (un origen se considera distinto si el puerto, dominio o subdominios varían), de forma que el navegador le aplique medidas de seguridad especiales, tales como evitar acceso a información del padre. La razón de servir tanto Blockly como las extensiones de usuario juntas es para simplificar la comunicación entre ambos y permitiendo un poco más de potencia a las extensiones sin muchas desventajas. Segundo, el iframe debe estar configurado como una ***sandbox***, pero con los permisos de *allow-scripts* y

allow-same-origin, que le permiten utilizar Javascript (necesario para las extensiones) y los mecanismos de Cookies y Local Storage (independientes del sitio web), respectivamente. Para comunicarse con el resto de la página, el iframe utilizará postMessage. El conjunto se describe gráficamente a continuación:

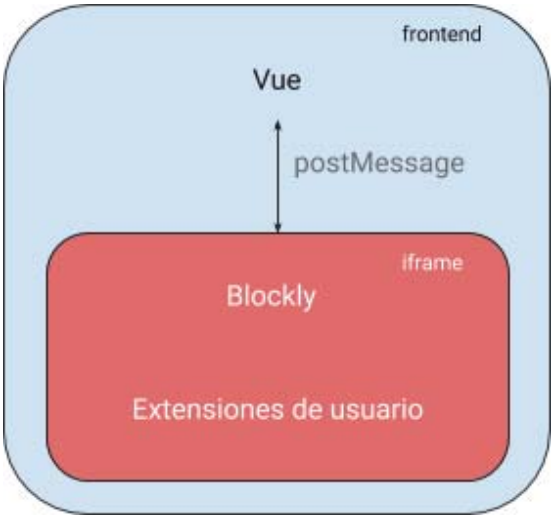

Figura 16. Entorno seguro para extensiones de usuario

Finalmente, la tercera medida es la aplicación de una CSP, (*Content Security Policy*), que limite los sitios web a los que el iframe puede conectarse remotamente, eliminado así los ataques de minado de criptomonedas al no poder transmitir información al exterior. Esto se puede conseguir utilizando NGINX (que, a pesar de ser una herramienta Linux, también tiene una versión Windows) con el siguiente bloque de servidor:

```
 server {
      listen       80;
      server_name  localhost; # Or whatever it needs to be, of course

      location / { # proxy requests to Node JS listening on 8080
         proxy_pass   http://127.0.0.1:8080;
         add_header  Content-Security-Policy "default-src 'self'
'unsafe-eval' 'unsafe-inline' *.bootstrapcdn.com; img-src 'self' data:";
         add_header X-Upstream $upstream_addr always;
      }
   }
```

Debido a que el mantenimiento de un servidor NGINX durante el desarrollo puede ser un poco tedioso, esta medida no es necesaria en el entorno de desarrollo, pero la aplicación de una CSP **debe** ser incluida en un despliegue de producción. Una vez aplicados todos mecanismos de seguridad, las amenazas mencionadas antes se neutralizan, únicamente permitiendo a los atacantes realizar vandalismo en el entorno de trabajo de blockly (por ejemplo, sobrecargando la pestaña del navegador con un bucle infinito).

Para dejar esta área un poco más zanjada, desarrollé una pequeña API para que las extensiones puedan añadir bloques y categorías fácilmente a los bloques, que provee las funciones `createCategory`, `registerBlock` (que funciona tanto para bloques en JSON como bloques complejos), y `registerGenerator`. Un ejemplo de uso autodocumentado se puede encontrar en los anexos.

4.6. INTEGRACIÓN CON EL TRABAJO DE MIS COMPAÑEROS

Una vez completadas las tareas previamente mencionadas, realizadas independientemente del trabajo de mis compañeros para ser lo más independiente posible (puesto que a ellos no les afectaba mucho mi progreso, y no había problema en integrarlo al final), procedí a integrarlas entre sí y con el trabajo de mis compañeros, concretamente con la parte de frontend, puesto que la relación entre backend y frontend no estaba establecida (no había un login, y por lo tanto no existe una sesión de usuario ni una manera de compartir datos específicos a un usuario).

En esta sección detallaré los dos problemas más significativos que encontré en el proceso de dicha integración, estando ambos relacionados con el renderizado de hardware. No hubo especiales problemas a la hora de integrar los temas de seguridad, puesto que se encuentran relativamente autocontenidas.

A la hora de integrar Vue y Paper.JS en la vista de Bioblocks, el inicializado que hacía de Paper.JS, que funcionaba correctamente por sí mismo, no parecía funcionar en la integración. Después de muchas pruebas, descubrí que un canvas de Paper.JS no se puede inicializar si está asociado a un bloque con v-if (por como genera Vue el DOM), y se inicializa con un tamaño de canvas incorrecto de 0x0 si está fuera de la vista, con lo que, para resolver ese problema, pasé a inicializar el canvas la primera vez que el usuario abre la pestaña de Hardware. Además, la inicialización se realiza con un retraso de 100ms puesto que el evento de cambio de pestaña se lanza previo a que el DOM se haya modificado. Aun con todo, siguen existiendo algunas situaciones en las que Paper.JS no renderizará todo el entorno correctamente, como podría ser el caso de una resolución de pantalla muy pequeña.

Figura 17. Integración de PXT-Blockly con el cliente Vue

Por otro lado, en la integración del desarrollo del iframe (iframe que ahora contenía PXT Blockly entero) con la visualización del hardware, me di cuenta de que el método de comunicación postMessage no era capaz de serializar ni el workspace ni los bloques Blockly, puesto que eran objetos muy complejos. Mi solución para este problema fue cambiar de nuevo el renderizado hardware, de forma que los datos que saliesen del iframe por parte de Blockly fuesen exclusivamente una asociación de bloques y pines.

4.7. POSIBLES LÍNEAS FUTURAS DE DESARROLLO

En esta sección exploro posibles mejoras al trabajo que he realizado durante este TFG.

4.7.1. EN LA PARTE HARDWARE

En lo relacionado con la parte hardware, mi opinión es que en general falta darle otra vuelta y quizás repensar ciertos aspectos. Es más, creo que si mi implementación ha servido para algo, es para dejar claro que como está no es la mejor manera de hacerlo (como poco técnicamente), y si me tocase continuar, no la reutilizaría (aunque sí algunas de las ideas detrás de esta).

Primero habría que aclarar en el concepto hasta qué punto se desean mostrar las conexiones del hardware (duda que me surgió al ver las especificaciones de los equipos con fuente de alimentación del equipo de Valencia en iGEM), y más concretamente si se deben soportar conexiones adicionales entre componentes. Concretar esto definiría de forma clara si se puede optar por un diseño centrado en una placa a la que se conectan cosas (como Bitbloq), o si todos los componentes se pueden interconectar entre ellos.

Por otro lado, creo que el modelo actual de incluir el/los pines en el bloque del experimento tampoco es ideal en el momento en que se quieran definir más piezas de hardware interconectados. Si aún así se considerase viable, hay problemas que no tengo ideas de como resolver elegantemente, como puede ser la distribución de las piezas de hardware en el espacio alrededor del Arduino Mega. Por ejemplo, dado el caso en el que tengamos un experimento complejo que haga referencia a 6 componentes con 3 fuentes de alimentación, ¿qué algoritmo determina como se colocan estos en el espacio? (este es un problema que Bitbloq no tiene, al colocar el usuario manualmente los elementos en el canvas).

Otro problema que identifiqué recientemente fue el siguiente: dado el caso en el que un experimento utilice la misma herramienta más de una vez (por ejemplo, realizar centrifugación varias veces), ¿cómo identificamos que la centrifugadora es la misma?

Puesto que los bloques tienen toda la información, al contrario que en Maker, que solo crea el elemento una vez y luego hace modificaciones incrementales, lo que sería equivalente a instrucciones como “Modifica velocidad de la centrifugadora <variable> a <valor>”, que

no existen en nuestro caso. En este sentido, veo dos opciones para la interacción hardware/experimento:

- Utilizar el modelo de Maker de una definición y modificaciones incrementales o
- Volver a la idea inicial de definir el hardware por separado (lo que ayudaría para hardware más complejo), sea con bloques o no, y asociarlo a variables específicas del experimento, que permita declarar con qué componente estás trabajando.

Para la ejecución de la segunda opción mediante bloques (que yo intenté y no conseguí llevar a cabo) se podrían variables tipadas combinadas con mirroring basado en eventos (demo), algo que desafortunadamente descubrí mientras investigaba un problema con los eventos de Blockly, y cuando ya tenía demasiado avanzada la solución actual como para dar la vuelta en el tiempo que me quedaba.

En lo que respecta a la parte puramente técnica a la hora de pintar el hardware, es posible que Paper.JS sea demasiado bajo nivel, necesitando la implementación de numerosas características para que tenga un buen diseño (tanto a nivel de usuario como en experiencia de usuario) y optaría por una abstracción superior basada en alguna librería de trabajo con nodos (como podrían ser XOD o Rete.js, aunque XOD parece más orientado a ser IDE). De hecho, creo que una implementación de nodos con etiquetas en los distintos puntos sería ideal, porque sería extremadamente flexible. El único punto en el que una solución puramente basada en nodos no funcionaría perfectamente sería con la placa Arduino, se perderían algunos puntos en cuestión de diseño e intuitividad si no estuviese basada en una imagen, pero tampoco me parece algo terrible por el momento (y a las malas, en un futuro siempre se podría optar por la ruta más complicada de Paper.JS)

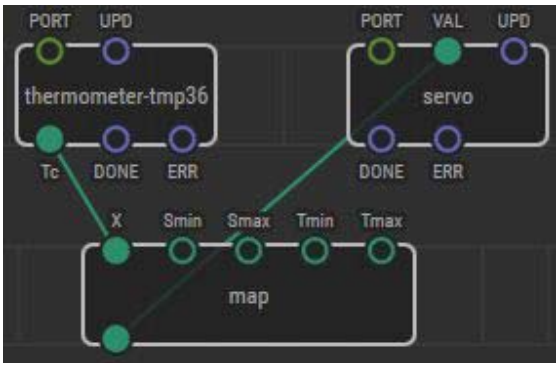

Figura 18. Ejemplo de interconexión de nodos en XOD

4.7.2. MEJORAS A LA HORA DE SERVIR PXT-BLOCKLY

Esta subsección es puramente técnica, pero necesaria desde mi punto de vista. Actualmente se sirve la carpeta de un nivel superior a pxt-blockly en el puerto 9000, que es poco ideal. La razón técnica de esto es que pxt-blockly en modo de desarrollo requiere la closure-library de Google, que no puede estar dentro del repositorio de PXT Blockly. Este no sería el caso en una versión de producción (que utilizaría los ficheros generados por build.py). En este caso, se me ocurren dos soluciones:

- Usar NGINX, y servir en el puerto 9000 exclusivamente lo necesario para que funcione Blockly, sin tener que servir dinámicamente.
- Tener un script que monitorice los distintos ficheros relevantes que se podrían modificar durante el desarrollo y los copie a una carpeta de distribución específica.

En realidad, hay una tercera alternativa: que solo se pueda usar la versión compilada con build.py, pero creo que no creo que sea aceptable para el flujo de desarrollo tener que recompilar blockly para probar cada vez que se haya hecho un pequeño cambio.

Cabe desatacar también que intenté pero descarté tanto incluir un enlace simbólico desde bioblocks/public, principalmente porque los enlaces simbólicos causan problemas con Git para Windows y requieren permisos de administrador Windows.

5. CONCLUSIONES

El proyecto ha sido bastante interesante y creo que sobre todo una buena base para el futuro, tanto literalmente en lo que respecta a PXT-Blockly y el resto del trabajo desarrollado por el equipo como figurativamente mediante el análisis y la serie de soluciones evaluadas y/o descartadas (junto con las razones para esto), todo pese a que no hayamos alcanzado el objetivo de tener un prototipo completo que nos propusimos al inicio. Así mismo, durante este trabajo he ganado conocimiento de todo el ecosistema de biotecnología, que es fascinante, apoyado en parte por cursar la asignatura de biotecnología en paralelo.

Quizás tardamos demasiado en definir una plataforma, lo que ha terminado limitando el tiempo que hemos tenido para desarrollar software sobre ella, aunque por supuesto era inevitable que se tardase un cierto tiempo mientras se analizaban y discutían las distintas posibilidades.

En lo que se refiere a expectativas, cuando vi que se me asignó el proyecto no esperaba que derivase en la relación de partes diseño/análisis y programación que ha acabado teniendo, pero creo que ha sido algo positivo, puesto que llevo unos cuantos años programando y recientemente estoy tratando de mejorar precisamente mis habilidades con diseño de sistema y análisis, para lo cual sin duda me ha ayudado el trabajo.

En retrospectiva, creo que debería haber hecho aún más pruebas de concepto, más simples, especialmente para probar la usabilidad de ciertas ideas - me di cuenta de algunos fallos en mis ideas (**especialmente** en la parte hardware) después de gastar esfuerzo en implementarlas, pero sí que estoy bastante contento en general con las decisiones de diseño que sugerí.

Creo que Bioblocks es un proyecto con muy buenos objetivos que espero consiga cumplir, así como que la comunidad lo adopte y que todo el mundo que así lo desee pueda disfrutar y aprender biotecnología. Me alegra haber podido contribuir durante estos meses.

6. ANEXOS

6.1. ANEXO: SETUP DE REPOSITORIOS DE SCRATCH

Adjunto un poco de documentación en lo que respecta a mis intentos de clonar e instalar los repositorios de Scratch en Windows, al menos parcial, por si alguien desease explorarlos en un futuro.

Para todos los repositorios, recomiendo no usar una versión superior a Node 8, Node 10 (o quizás el npm que viene con este) presentó problemas a la hora de ejecutar algunos comandos. El repositorio **scratch-gui** funciona sin problemas siguiendo las instrucciones presentes en el README que se puede encontrar en el mismo repositorio, con lo que no documento nada adicional en esta memoria.

6.1.1. SCRATCH-WWW

Este repositorio es el *Standalone web client for Scratch*. Necesita Python 2.7 para la instalación, versiones siguientes no funcionan. Antes de ejecutar **npm install**, en caso de que no se haya corregido todavía (se trata de un error de webpack-cli en 2018), hay que modificar el fichero package.json y fijar webpack a la versión 4.19.1, de lo contrario no instalará.

Además, para poder ser capaz de ejecutar **npm run build** hay que modificar build.py para que tire un poco. Buscar la asignación de test_args y reemplazarla por lo suguiente:

```
test_args = ["java", closure_dir + "\\" + closure_compiler + "\\compiler.jar", os.path.join("build", "test_input.js")]
```

, donde compiler.jar es el closure compiler de Google, que habrá que descargarse previamente. Nótese que si no hay un JDK instalado local se usa la versión del compilador en remoto y en ocasiones puede llegar a tardar 10-15 minutos.

Una vez compilado, es posible encontrar problemas con la política de seguridad (que se puede saltar o bien deshabilitándola en el navegador, poco recomendado, o bien teniendo un proxy intermedio - parece que existe también un fichero crossdomain.xml, aunque con ese no he probado).

Pese a mencionar ser “standalone”, hace llamadas a la API de Scratch online al arrancarse, así que la afirmación pierde un poco de valor.

6.1.2. SCRATCH-BLOCKS

Este repositorio hace referencia a la lógica de bloques de scratch, junto con su diseño. Desgraciadamente, no documenté adecuadamente los pasos para corregir el script de build en Windows, que no me funcionó directamente, pero sí que puedo confirmar la necesidad de tener la herramienta make en el PC. Para poder tenerla en Windows, se puede utilizar Git Bash y obtener *make* como indica el siguiente Gist de Github: https://gist.github.com/evanwill/0207876c3243bbb6863e65ec5dc3f058

La alternativa a no usar Git Bash para *make* es instalar *mingw* de forma independiente o *cygwin*. Una vez se tenga make en el sistema, se debe ejecutar `make translations` , a continuación `npm install`, y finalmente con `npm start` debería funcionar.

6.2. ANEXO: LISTA DE PERFILES DE USUARIO DE BIOBLOCKS

A continuación sigue una lista de distintos perfiles de usuario de Bioblocks que recopilé y como esperamos que utilicen algunas de las características del software (consta de algunas contribuciones por parte de Jorge Díez, especialmente en las últimas dos categorías). La lista se encuentra en inglés puesto que la documentación del proyecto se ha realizado en inglés después de la inclusión de un estudiante extranjero al mismo.

School students

- They are not going to change hardware components
- Will mostly have private experiments, shared within a group or with at least the teacher
- Will likely have a pre-designed hardware or some sort template for it

Biotechnology enthusiasts

- They want to set up their biolab in the system and be able to share it and describe experiments for it
- Will use private experiments, but also share some publicly
- They will be potentially setting up new hardware, the ecosystem is very heterogeneous. May need to be able to add said hardware to our system.

Undergraduate and Graduate students (Biology/Biotechnology)

- Some may add hardware components if they have some knowledge in programming
- May also follow private experiments set up by their professors, and may make alterations to such experiments to satisfy their own needs

Teachers and Professors

- Specially interested in the capabilities of setting up pre-made experiments for their students (both in terms of hardware and software)
- Will likely need the ability to include new hardware components
- Will expect having their work saved remotely

6.3. ANEXO: EJEMPLO DE API

```
//////////////////////////////
// Creating a new category
//////////////////////////////
createCategory({
   id: "examplecat",
   name: "Example",
   color: 390,
   iconClass: "blocklyTreeIconCustom"
})

//////////////////////////////////
// Registering a JSON-type block
//////////////////////////////////
registerBlock({
   json: {
       "type": "pcr_example", // This is the block type as usual
       "message0": "Create PCR attached to %1",
       "args0": [
           {
               "type": "field_dropdown",
               "name": "PIN",
               "options": [
                   ["A0", "A0"],
                   ["A1", "A1"],
               ] // ...
           }
       ]
       // (etc)
   },
```

```
   /*
    *  You can explicitly design how the block will appear and whether it has
children or not.
    *  Note that XML is *optional*, if not provided, a default one will be
built based on the
   *  type, like the one here:
   */
  // xml: `<block type="pcr_example"></block>`
   category: "examplecat", // Category represents the category the block's
XML will be added to. You may use an existing category or create a new one
});

// Registering a generator for transforming pcr_example into Arduino
registerGenerator("pcr_example", function(block) {
   // Transform the block to a string of code here as needed.
   // The block instance will always be a parameter to this callback
   return 'char hello[] = "world";';
});

////////////////////////////////
// Registering a complex block
////////////////////////////////
const Centrifuge = {
   init: function() {
       this.setInputsInline(true); // Horizontal: Yes
       this.setPreviousStatement(true); // Can stick to something before
       this.setNextStatement(true); // Can have a modifier on the bottom
       this.setColour(120); // Green

       // Title
       this.appendDummyInput()
         .setAlign(Blockly.ALIGN_CENTRE)
         .appendField("CENTRIFUGE");
       this.setTooltip('');
```

```
        // Define the block
        this.appendValueInput("SPEED")
        .setCheck("Number")
        .appendField("Speed (RPM)");

        // ...
    }
};

registerBlock({
    type: 'centrifuge2', // This is the block type for Blockly
    obj: Centrifuge,
    // xml: `<block type="centrifuge2"></block>`
    category: "examplecat",
});
```

Este documento esta firmado por

Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
Fecha/Hora	Mon Jan 14 22:07:07 CET 2019
Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
Numero de Serie	630
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)