Designing a Plugin System For Use Across Multiple Websites

by

Shane O Donovan

This thesis has been submitted in partial fulfillment for the degree of Bachelor of Science in Software Development

in the
Faculty of Engineering and Science
Department of Computer Science

November 2019

Declaration of Authorship

I, Shane O Donovan, declare that this thesis titled, 'Designing a Plugin System For Use Across Multiple Websites' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for an undergraduate degree at Cork Institute of Technology.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Cork Institute of Technology or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ________________________________

Date: ________________________________

CORK INSTITUTE OF TECHNOLOGY

Abstract

Faculty of Engineering and Science
Department of Computer Science

Bachelor of Science

by Shane O Donovan

Traditional websites offer limited options for users to expand on product functionality. There has been growing demand for plugin systems in recent years, made apparent by Trello [1], Zendesk [2] and Google Docs [3] all developing their own systems. These plugin systems enable users to add functionality to said sites via JavaScript, but the problem with existing systems is each site uses a different system. This means plugins are incompatible across websites and need to be completely rewritten for each site.

The goal for this project is to research how one would design a generic plugin system which can be easily implemented on any site. If one was to use the same plugin system on each site, only minimal code changes would be required for a plugin to function on another site as they share systems.

Plugins can insert components such as buttons onto a website, create pop-up dialogs and insert iframe widgets. Plugins can also receive data from the API of the site on which they are being embedded and have dialog contents change based on this data. Plugins are isolated so they cannot make breaking changes to HTML of the site on which they are embedded. Plugins will still likely need modifications to run on different sites, for example API endpoints on websites may differ, but the general idea is to minimise the number of required modifications for cross-site plugin compatibility.

Taking this generic approach means plugin developers need not maintain as much code as plugin code is largely the same for all sites and time spent porting plugins is reduced resulting in customers being presented with a wider variety of plugins. The system has two major selling points which are the ease at which it can be implemented on sites and its ability to support plugins compatible across several websites or products due to the same underlying system being used on each site.

The system's flexible design will be demonstrated by implementing it on two different sites and creating a plugin which functions on both sites.

Acknowledgements

I would like to thank my project supervisor Dr. Ruairi O'Reilly who provided suggestions and critiques throughout the research phase, aiding in steering the project in the right direction and putting up with my numerous idiosyncrasies.

I would additionally like to thank my workplace supervisor Conor Higgins who originally proposed that I research implementing a plugin system.

Contents

List of Figures

List of Tables

Abbreviations

DOM	**D**ocument **O**bject **M**odel
API	**A**pplication **P**rogram **I**nterface
CRM	**C**ustomer **R**elationship **M**anagement
IT	**I**nformation **T**echnology
UI	**U**ser **I**nterface
JS	**J**ava**S**cript
CSS	**C**ascading **S**tyle **S**heets
JSON	**J**ava**S**cript **O**bject **N**otation
SaaS	**S**oftware **as** **a** **S**ervice
CMS	**C**ontent **M**anagement **S**ystem
PHP	Hypertext Preprocessor
POT	**P**ortable **O**bject **T**emplate
SVN	Apache **S**ub**v**ersio**n**
SQL	**S**tructured **Q**uery **L**anguage
XSS	**C**ross-**s**ite **S**cripting
ZAF	**Z**endesk **A**pp **F**ramework
SDK	**S**oftware **D**evelopment **K**it
PoC	**P**roof **of** **C**oncept

Chapter 1

Introduction

1.1 Motivation

It is common when developing a product that a customer will request a feature that would prove beneficial to their company, but where for one reason or another, the feature ends up falling on the wayside and not being implemented, leaving the customer dissatisfied.

This situation can occur for a number of reasons.

- The customer may have requested a feature which would have benefited their own company but would not have served much if any value for other customers. Features which seem to developers like they would be of benefit to a greater portion of users are generally prioritized over ones which would benefit few.
- It could be a feature which is deemed of value by developers, but there may simply be an existing backlog of feature requests planned out for the next few months and the developers might not have the resources to implement it within the near future.

Over the years, there has been an increase in the number of integration systems being offered by websites for other products. For instance, Gmail Add-Ons, a plugin system for Gmail enabling integrations with third-party websites, was launched as recently as October 2017 [5]. This is further supported by Salesforce's acquisition of MuleSoft, a company developing a product enabling easier integration between applications and now used in Salesforce's own plugin system, for 6.5 billion dollars in March 2018 [6]. They wouldn't invest such a large sum of money into this integration product if they didn't see a profit opportunity, indicating there is significant demand for integration systems.

Rather than having several websites invest time and resources into adding the same requested functionality, a third-party plugin system would make it so the functionality could be written once for a particular service and then re-used across multiple websites. For example, in the case of project-management products, a plugin which adds functionality to Trello's boards could conceivably also function in Teamwork Products, as they both have a boards system, existing as a common section in both products, as demonstrated in Figure 1.1. This simplifies plugin development and reduces the likelihood of one reinventing the wheel when writing their plugin for different websites.

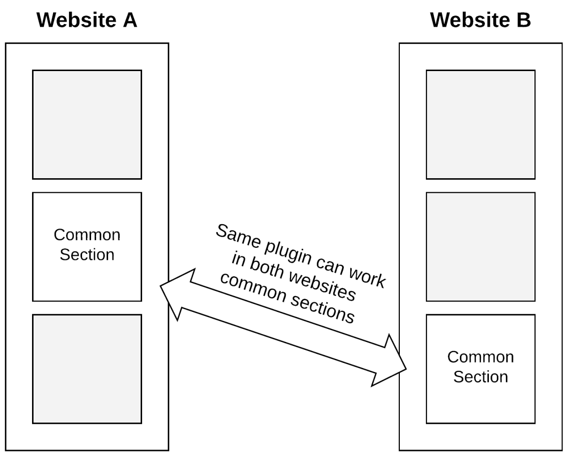


FIGURE 1.1: Common areas in separate products which can contain the same plugin

A website plugin system would enable these users to add the requested functionality themselves, meaning they need not wait for a developer to implement the functionality for them. They could then share this plugin with other users interested in the same functionality.

However, the security of such an implementation would be of importance, particularly among Enterprise customers with sensitive data. Third-party sites would not implement our plugin system if there was a chance plugins could perform unauthorized actions or introduce security vulnerabilities into the website. For example, Cross-Site Scripting (XSS) is a vulnerability which would allow plugins steal user information or perform unrestricted actions on behalf of a user on a website [7]. Therefore one would need a form of sandbox or isolation in order to reduce the impact of malicious or vulnerable plugins.

1.2 Executive Summary

The idea is to introduce a centralized plugin system, reducing the amount of code duplication across websites, providing users with better integrated products and supplying developers with more time to focus on improving upon their products' core features. Given competing products often have similar features, they could also feasibly share the same plugins, so both parties can make use of a wider range of plugins, which benefits a wider demographic of customers as plugins cater towards different customers.

The concept is to create a plugin system capable of performing the following actions:

- Displaying dynamic text on pre-approved sections of a website
- Displaying buttons and widgets on pre-approved sections of a website
- Saving per-plugin preferences or other data into a database for later retrieval
- Displaying basic pop-up dialogs, such as preference modals, upon clicking a button.
- Detecting when an event has occurred and performing an action when this happens.

A plugin's client-side code would be executed in a sandboxed environment, thereby preventing it from modifying the DOM or otherwise performing any malicious activities including sending unauthorized API requests outside the scope of the plugin's permissions.

1.3 Structure of This Document

- The goal for Chapter 2 is to discuss the rationale behind why one would want to implement a plugin system. We also explore the various existing plugin systems with regards to their functionality and implementation.

- We outline in Chapter 3 the functional and non-functional requirements in the plugin system we plan on implementing, along with our reasoning as to why said requirements are valuable. These requirements essentially comprise what we want our plugin system to be capable of, without using much technical terminology.
- Chapter 4 contains a technical plan on how we will implement the project along with what our other implementation options we considered. We will outline our overall structure for the system and predictions on where the resource drains and pitfalls in the project's development could lie.
- Finally, we conclude the paper with Chapter 5, in which we reflect on the findings in the report, outline the problems we encountered and discuss any additional functionality which we would have implemented had time permitted.

Chapter 2

Background

Salesforce is a CRM software, with an application marketplace called AppExchange [8]. In 2016, according to Salesforce's then vice-president, more than 85 percent of Fortune 100 companies were using at least one AppExchange app [9]. Salesforce apps are like plugins in that they add addditional functionality to the product such as exporting and cloud storage integration. Salesforce's vice-president mentioned that customers use AppExchange to extend Salesforce functionality and broaden its use into other departments within their organizations, so plugins cater towards different needs of customers from different areas, for example IT, Sales, Support. AppExchange installs increased from 5 million to 6 million from roughly October 2017 to September 2018, despite the plugin system existing for the last 10 years. From this we can establish that the demand for plugin and integration systems is growing exponentially.

2.1 Downsides of Traditional Websites

Traditional websites typically offer no means of extensibility for customers, meaning essentially what you see is what you get. In the event that a feature a customer desires does not exist, they have limited options of adding said feature themselves. The only means in which they can hope for such a feature to be implemented is if they contact customer support and file a feature request.

This leads into the second issue, where the customer needs to wait for the request to be reviewed. Given there could be numerous other customers filing different feature

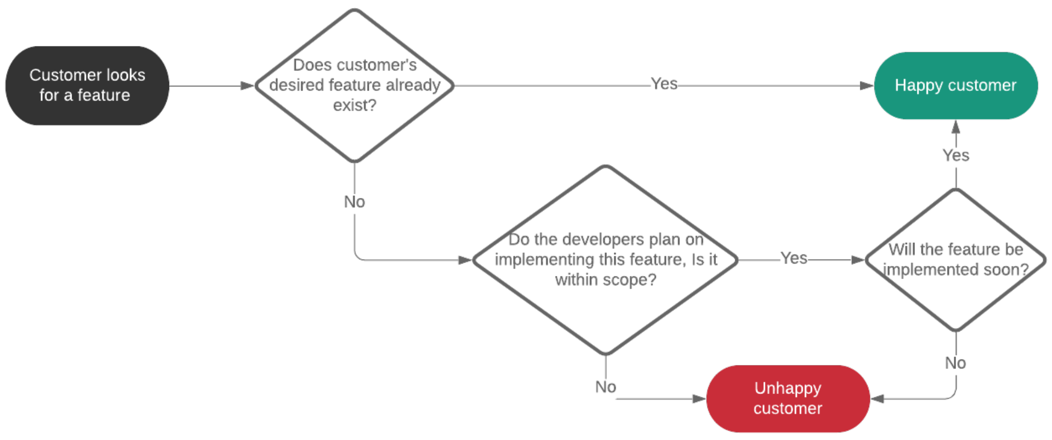

Customer looks for a feature
Does customer's desired feature already exist?
Yes
Happy customer
No
Do the developers plan on implementing this feature, Is it within scope?
Yes
Will the feature be implemented soon?
Yes
No
No
Unhappy customer

FIGURE 2.1: Traditional systems where plugins don't exist

requests, there can often be a request backlog spanning across several months. This means that a customer could potentially end up having to wait a significantly long period of time for their request to be considered and the feature to be implemented.

It is entirely within the realms of possibility that the website's developers may regard the client's requested feature to be of little value to other users, or they may even deem it to be outside of the product's scope or vision, in which case said developers may decide to forego implementing said feature entirely. This is an issue for clients as they miss out on a feature that they consider of significant value and they are left with no alternative.

The above process is illustrated in Figure 2.1.

2.2 Benefits of a Plugin System

2.2.1 Access to a Broader Range of Features

Through plugins, users have access a broader range of features offered by the product's developers. It is possible that these specific features may not match up with the vision of the product and hence end up not being developed, whereas the plugin system enables third parties to develop the features which the product's developers deem out of scope. Salesforce's plugin system is a good example as customers come from a variety of different backgrounds, with some belonging in sales, and others in support. It is unrealistic to expect a product's developers to cater to all of these backgrounds in its core features, which is why the plugin system serves of benefit to the customers as it satisfies an unmet need.

2.2.2 Enterprise - Paid Feature Requests

From a developer's perspective, if an enterprise customer requests a feature via the product's built-in request ticketing system, and it's not a feature which would benefit all users collectively, the developers could satisfy the customer by implementing this feature for them in the form of a plugin. Said developers might also benefit monetarily from this by making it a paid plugin and hence charging said enterprise customer for their services, so both parties stand to gain from its implementation.

2.2.3 More-Refined Product Functionality

Due to it being possible for plugins from other similar products to be used, this means that a given product's developers don't need to invest as much time developing plugins themselves, and can instead focus on refining their product's core functionality. This ends up benefiting customers as the overall product ends up being more polished and less bug-prone.

2.3 Comparing Existing Plugin Systems

When developing our own plugin system, it helps to have an understanding as to how existing plugin systems operate. Hence, we will explore existing solutions below, examining their implementation and the capabilities of said plugins.

2.3.1 Trello Power-Ups

Trello is a project-management solution which enables users to organise their projects' tasks. It allows users to create boards which represent tasks (cards) across columns, as demonstrated in Figure 2.2.

Trello Power-Ups is what Trello refers to its plugin system as [1]. It enables users to add additional UI elements and have those elements perform actions on the website on

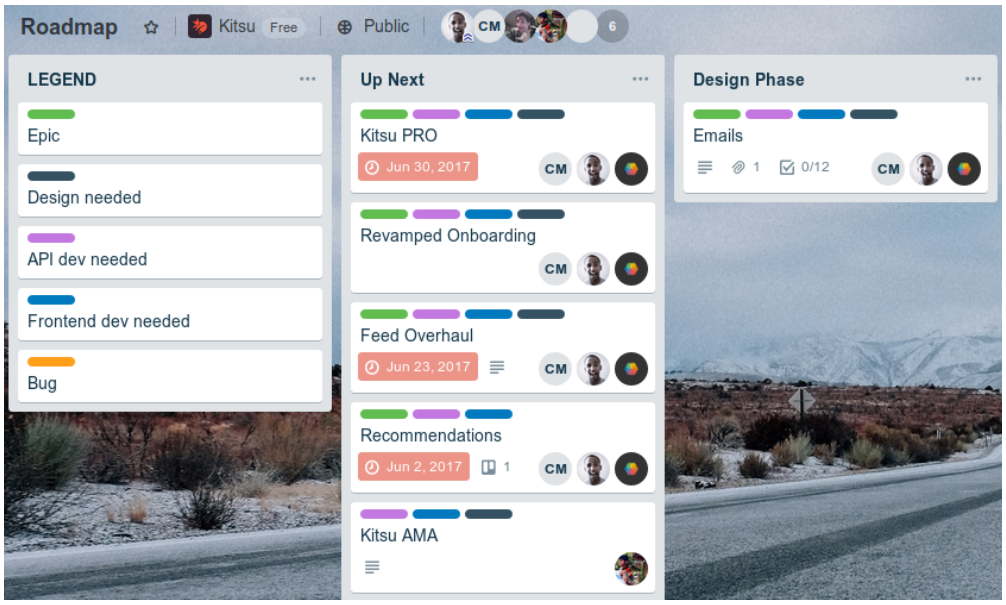


FIGURE 2.2: Project tasks represented using cards and columns

behalf of users. In practice, it's used to integrate other services with Trello, for example one plugins allows users to add an attachments from Giphy. It was launched in August 2016 [10] and its repository has since amassed over 80 public plugins [11].

2.3.1.1 Implementation

Due to the closed source nature of this feature, its full implementation is unclear. What is known, however, is that each power-up is hosted on its own website and is initialized by Trello in a hidden iframe [12]. Plugins must include a Trello JS library which handles the communication between the plugin iframe and Trello itself in order for them to function. Any communication between Trello and a power-up is performed via the JS postMessage API. This enables Trello to take advantage of iframe sandboxing techniques, serving as a plugin security layer, ensuring that plugins don't modify the DOM or otherwise perform any disallowed actions. Disallowed actions could possibly include using the logged in user's cookies to request an API endpoint that the plugin hasn't been authorized to access.

2.3.1.2 Capabilities

- Display buttons in certain pre-approved sections such as above a board. Each button can have a label and an icon, with different sections on the site requiring both light and dark icons, and other sections requiring light or dark specifically. The buttons match the styling that Trello uses, there is no CSS customization.

- Display pop-up modals containing a list of items, where clicking on each item performs a different action. There is also the option to create a modal containing a list of items and a search box, which may suit for larger lists. This modal offers no custom styling.
- Display iframe pop-up modals upon a button being pressed. Due to it being an iframe, the author has full control over it and can style it however they desire. If an iframe modal's content size changes due to JavaScript at a later point, this modal can be resized using a built-in Trello API function.
- Display 'alerts', which are essentially pop-up messages which show in the top-right corner and disappear after a certain period of time.
- Localize text via built-in support for string localization using locale-specific key/-value JSON configuration files.

2.3.2 Slack Apps

Slack is a chat application. Slack Apps was introduced in December 2015 [13], and was essentially a plugin repository and system for Slack, enabling third-party users to extend Slack with their own desired functionality. Prior to this, Slack had its own official integration plugins with third-party services, but users were not capable of creating their own plugins and were limited to the existing supported integrations.

2.3.2.1 Implementation

Slack Apps works in a completely different manner to that of Trello Power-Ups. In Slack's case, in order for a plugin developer to display elements in the UI, they can simply go through a friendly interface allowing them to add the UI elements [14]. One doesn't need to write any code to add UI elements. When said elements are clicked, it will send a request to an API endpoint under the developer's control. So, where Trello Power-Ups are comprised of mostly client-side code, Slack Apps are coded entirely server-side.

2.3.2.2 Capabilities

- Display additional options in the drop-down per each chat message. Clicking on any one of these options will send a request to the external endpoint one specifies, giving the message details as context.
- Create custom chat slash commands in the UI which, when said, will send a request to an external endpoint containing any arguments.
- Publish messages in chat rooms by sending a request to a unique-per-channel Slack endpoint known as an incoming web-hook.
- Subscribe to events, meaning when a specific event occurs, for example, if a user joins a channel, Slack sends a request to an events-specific external endpoint specified by the app. The same endpoint is used for handling all kinds of events, and the event details are specified in the request body. In order for an event to be successfully received, the app needs to have that event permitted within its OAuth scope.
- Assign a bot user with a specified name using the UI, so the user can send messages to the bot, and if a bot event is subscribed to, for example when a user is @mentioned, the event will be sent to the events endpoint previously mentioned.

2.4 Downsides with Traditional Plugin Systems

2.4.1 Tied to One Site

Often plugin systems are only developed with a single website in mind. For example, the Trello Power-Ups system was developed specifically for use with Trello. One could not simply import the Trello Power-Ups library on another website and expect it to work.

2.4.2 Plugins are Difficult to Port

As plugin systems are often tied to a single website, this means that there ends up being a plugin system developed for each site. Therefore, if one wants to develop a plugin and have it work on two different sites, they must learn two different systems. This means plugin authors end up having to rewrite large portions of their code to support an additional system.

2.5 Products Which Could Benefit From a Plugin System

2.5.1 Teamwork Projects

Teamwork Projects is a SaaS project-management solution which facilitates users in better managing their project workloads. Users can create projects, assign tasks and comment on said tasks, post messages and create notebooks [15]. Additionally, it has boards functionality, shown in Figure 2.3, which essentially matches that of Trello's functionality.

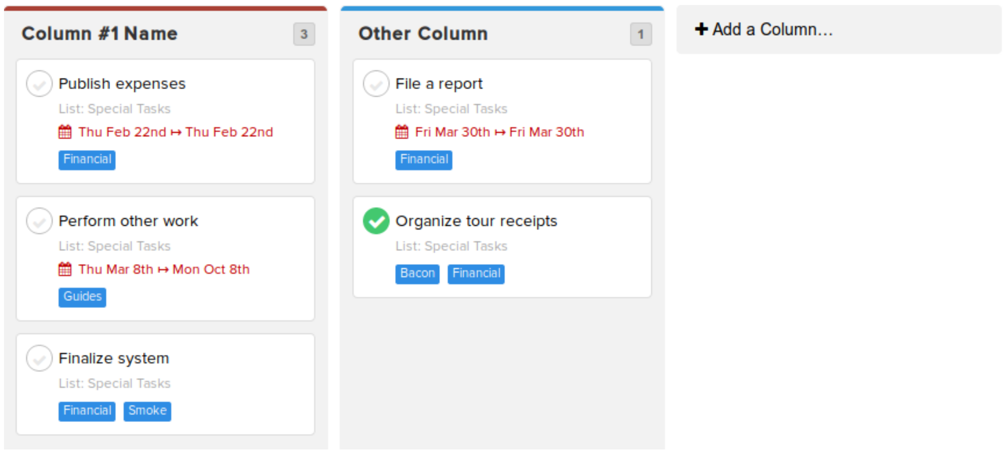


Figure 2.3: Teamwork Projects board system, cards and columns shown

Although Projects has built-in support for integration with various cloud file upload services such as Google Drive and OneDrive Business [16], and supports roughly 20 integrations in total, users are only given the choice of using these integrations and have no option to make their own plugins. Given Trello, a competing project-management product, has a plugin system with more than 80 plugins, and Teamwork Projects lacks such a system, this means Trello has a selling point which Projects lacks.

2.6 Exploring Other Plugin Architectures

2.6.1 Content-Management System (CMS) Plugins

Content-management systems such as WordPress enable people to create their own websites, without requiring writing as much code as would normally be necessary to make such a site. In the case of WordPress, site design can be customised by writing custom themes via WordPress Themes, and extra functionality can be added via WordPress' plugin system called WordPress Plugins [17].

2.6.1.1 Themes

WordPress themes are separate to its plugin system, but it is worth exploring how the themes system operates as it may provide an insight into how systems are designed with customisability in mind.

Themes are comprised of PHP and CSS files. Some PHP files are used for templating, while others contain functions. Web-pages are split into sections, with each section having its own corresponding PHP theme file [18]. Themes can overwrite each of these sections with their own PHP files. If one wanted to change the order in which elements are represented, they would copy the default theme PHP file and rearrange the PHP code within. API functions facilitate retrieving post information which can be injected by the PHP template. Some of the PHP theme file names and where they are positioned is displayed in Figure 2.4.

2.6.1.2 Plugins

As of 2016, WordPress had more than 44,000 plugins, with downloads altogether totalling over 1.2 billion [19]. Plugins consist of at least one PHP file and can also contain JS files, images and localization files. They can have the following capabilities:

- Create a database table in which plugin-specific data can be stored and retrieved.
- Localize strings via built-in support for key-value string localization. These strings are stored in PHP-proprietary POT files rather than JSON. Both are similar in concept.

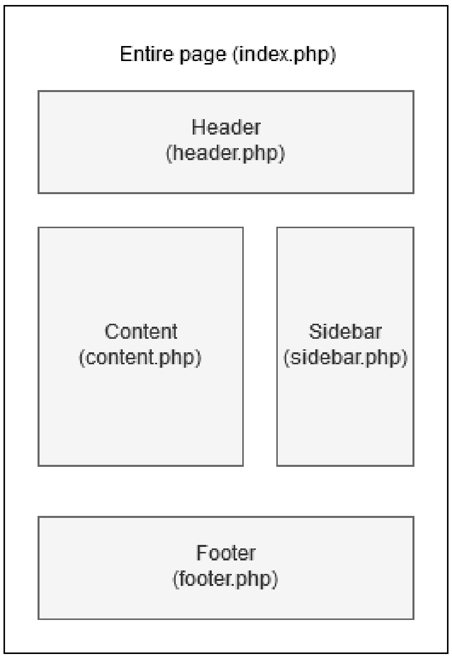


FIGURE 2.4: WordPress default template page structure

- Auto-update functionality exists for official plugins, where they are stored on an SVN (version-control system) and so the plugin developer uploading new revisions notifies its users of an available update.
- Run PHP and SQL directly on the website, and add widgets comprised of HTML and JS files without requiring them to be inside an iframe. This differs to Trello's Power-Ups feature which runs code under a sandboxed iframe environment. While WordPress' solution enables developers more freedom, it also poses as a security risk as there is a possibility of a malicious plugin introducing an SQL injection and/or XSS vulnerability in a website. Website defacements are commonly caused by insecure WordPress plugins. As of writing, 22.8% of vulnerabilities logged in WPScan's WordPress vulnerability database are caused by plugins [20].
- Listen for events, so when such an event occurs, the plugin's PHP code will perform an action in the background. For example, if a user creates a post and the plugin is configured to listen for this event, the plugin will execute said code.
- WordPress plugins use the WordPress PHP API, which permits displaying widgets in widget areas and adding additional WordPress setting sections and items

2.7 Exploring Varying Plugin Complexities

Some plugins make use of more advanced functionality than others. Increasingly complex plugin functionality require a wider range of features to be supported by the plugin system itself. Different customers may have use-cases for plugins, with some requiring simple read-only widgets, and other customers requiring plugins containing logic and event handling. We investigate below various plugins of differing complexity in an effort to understand what customer use-cases we should satisfy when developing the system.

2.7.1 Simple - Static Plugin

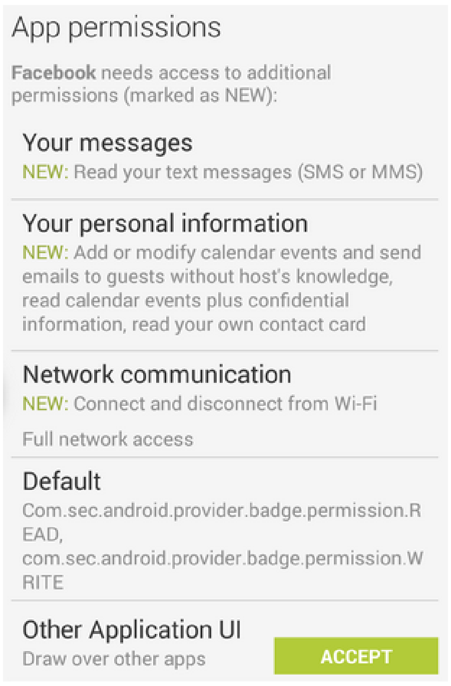
These plugins are entirely static. They have no back-end element, and instead are essentially logic-less widgets. One such example of this is a plugin which adds a static read-only calendar to a website. It would not support internal/external API requests, so one could not view/add/edit events through an API.

2.7.2 Medium - External API Usage

External API access is a feature which would be of medium complexity. Plugins could send and retrieve data to third-party APIs unrelated to the site on which they are being hosted. For example, if an external site hosts calendar events, one could retrieve the events and then display them on the site which implements the plugin system.

2.7.3 Hard - Internal API Usage

A hard-to-implement plugin could be one which can send API requests to the website the plugin is being ran on using the logged in user's cookies. It could have a permission scope system whereby the user would be required to permit the plugin to send the desired API requests. For example, given the proposed system, if a plugin wanted to create a task in Teamwork Projects, the user would need to approve of this permission when installing the plugin, and in the event of a plugin update requiring additional permissions, the user would need to confirm the new permissions, similar to how Android's system operated in older firmware versions.


App permissions

Facebook needs access to additional permissions (marked as NEW):

Your messages
NEW: Read your text messages (SMS or MMS)

Your personal information
NEW: Add or modify calendar events and send emails to guests without host's knowledge, read calendar events plus confidential information, read your own contact card

Network communication
NEW: Connect and disconnect from Wi-Fi
Full network access

Default
Com.sec.android.provider.badge.permission.READ, com.sec.android.provider.badge.permission.WRITE

Other Application UI
Draw over other apps

FIGURE 2.5: Android's system notified the user if an app required extra privileges when updating

2.8 Existing Plugin Sandbox Solutions

Knowing how other products achieved sandboxing will provide us with an insight into what viable sandboxing solutions we should consider when implementing the proposed system. Sandboxing is necessary if one is permitting plugins execute JavaScript code as otherwise plugins would effectively be offering a form of cross-site scripting [7].

2.8.1 Zendesk

Zendesk desk is a custom support and ticketing service. Its developers looked into a number of solutions when designing Zendesk Apps, a plugin system for Zendesk [2]. This system allows users to embed their own desired functionality into Zendesk products. Plugins are essentially user-configurable iframe widgets which can be embedded into the site. Users can share their plugins via the Apps Marketplace. This is similar to how our proposed system will work.

When developing the plugin system, Zendesk researched various sandbox methods so as to prevent untrusted plugin code from making unauthorised API requests on behalf of the user among other reasons.

Zendesk looked into the following sandbox solutions:

- Google Caja - This is a Google library which allows sites to embed untrusted JavaScript and CSS code in a secure manner. Zendesk's reasoning behind not using it was its then-uncertain future and its implementation would require making fundamental changes to Zendesk's system's design.
- Web Workers - These are background threads one can create in JavaScript. Background threads offer a sandboxed environment but lack their own DOM. The DOM is an area in which they can create HTML elements. The lack of a DOM resulted in Zendesk not considering this approach.
- Web Components - This feature allows one to create custom HTML tags which can replace chunks of code. Zendesk's comment was that it would help with sandboxing JavaScript, but the feature was not yet standardised so it was not a viable option at the time.
- Iframes - This is the approach Zendesk ended up using in its system. On Zendesk's website itself exists a Zendesk App Framework (ZAF) JS file. Plugins all include a separate ZAF SDK JS file which communicates with the website's ZAF JS file. The two JS files allow cross-frame communication using the JS postMessage API. This API allows iframes (plugins) and their parent window (Zendesk) to communicate with each other. The postMessage API is not straightforward to use however, so the two scripts act as an abstraction of this API so plugin developers do not have to concern themselves with postMessage and can instead use functions from the user-friendly plugin SDK in their plugins and these functions will then execute on the Zendesk website. This approach ensures only a fixed set of functions can be executed on the site by plugins and HTML5 iframe sandboxing prevents Zendesk's DOM from being modified by the plugin.

2.8.2 Trello

We referred to Trello's system before, but this time its sandboxing is being looked at specifically. Trello's Power-Ups system involves several components. Plugins are hidden iframes embedded on the page. Each plugin has its own iframe. Trello takes advantage of HTML5 iframe sandboxing to ensure malicious plugins cannot affect the DOM of

Trello itself. This solution also prevents plugins from affecting or interacting with each other.

The concept is similar to Zendesk's iframe solution. All plugin-to-Trello communication is performed via the JavaScript postMessage API. This API allows messages to be sent from an iframe to its parent window regardless of if they are on different domains. There is a client.js library which all plugins must include. On Trello's side, it has its own script which handles receiving messages sent by the plugins via their respective client.js lbiraries.

Chapter 3

Problem - Designing a Plugin System For Use Across Multiple Websites

3.1 Problem Statement

Traditional websites tend not offer third parties a means of integrating their product's functionality. However, there has been a rise in recent years in plugin systems which facilitate this need by allowing third-parties easily add functionality to an external website. Examples include 'Gmail Add-Ons', 'Trello Power-Ups', and 'Slack Apps', and none of the plugins created with these sites are compatible with each other, meaning plugins would need to be rewritten for each site.

Gmail's system, 'Gmail Add-Ons', offers integrations with Trello, Asana, Teamwork Projects and a number of other services. It allows users to perform actions on these external services while viewing their emails. Outlook.com on the other hand has no such system, providing Gmail with a unique selling point, reducing the likelihood of someone using Outlook.com.

Even if Outlook.com had a plugin system, it would likely be written specifically for use with Outlook.com in a similar manner to that of Trello and Slack, meaning plugins would not be compatible across different websites, so Gmail plugins would not function on Outlook.com and would instead need to be rewritten as a separate Outlook.com plugin, duplicating plugin development efforts. This inevitably results in some plugins being compatible with one client and not the other, and the product with the most market share tends to end up having a greater selection of plugins, so users on lesser-known products end up suffering. In our example, Gmail has 27% market share and Outlook products have 12% [21]. This means Gmail would have the upper hand and Outlook could lose users due to either not offering any integrations in the event of no system existing or offering fewer than Gmail in the event of them using separate systems.

3.2 Objectives

Ultimately the goal is to create a third-party website which provides plugins for multiple websites. These plugins would add buttons and widgets to an external website, providing said site with additional functionality. Plugins could be written once and with minor modifications would be capable of running on a separate website which also happens to support the third-party plugin system. The benefit of this approach is, plugin developers can reduce development times by using the same plugin code across multiple websites. This would also benefit users, as the reduction in time spent porting a plugin to another website increases the likelihood of a plugin supporting more websites. This means a wider variety of plugins for users. The purpose of the plugins themselves is to allow users to provide an avenue for third-parties to add functionality to a website where said functionality would not otherwise exist.

The difference between Pluggy and a traditional system such as Trello Power-Ups is that Pluggy would be easily implemented on any site, and its plugins compatible with multiple sites meaning code need not be totally rewritten for compatibility. An attempt will also be made to expand on the features that Trellos system offers in order to facilitate more advanced plugins.

3.3 Stakeholders

3.3.1 Third Party Service Providers & Their Users

If a plugin for a website is providing integration with another service, said service stands to benefit as its users are able to access its functionality seamlessly without changing website, thereby improving customer engagement. For instance, Teamwork Projects recently created a Slack app which enables creating tasks from Slack messages and also posts messages to Slack when tasks are created. Projects' announcement blog post for this feature has the highest reply count of any post in over 6 months, indicating users see value in it.

3.3.2 The Website Integrating the System & Its Users

The website implementing the plugin system could gain from this as it would give it a unique selling point over other competing products which lack such a system. For example, if a Projects app exists for Slack but not Flock, a messaging client which competes with Slack, this could cause Flock to lose customers who require said integration. Therefore it would be in Flock's best interest to share plugin system with Slack. Likewise, it would be in Slack's interest to share these plugins as plugins created for Flock would be compatible with Slack, widening the range of supported plugins.

3.4 Example Use Cases

3.4.1 Chat Clients

One could design a plugin which adds a context menu item next to a Slack message enabling users to create tasks in Teamwork Projects from a Slack message. Upon researching, we discovered Slack added support for apps to add this functionality as recently as May 22nd 2018 [22]. In any case, other chat clients could benefit from the feature's addition via our system as much the plugin's code could be re-used across all clients. This would require plugins authenticate with an external service (Asana in Figure 3.1) and send API requests to said service.

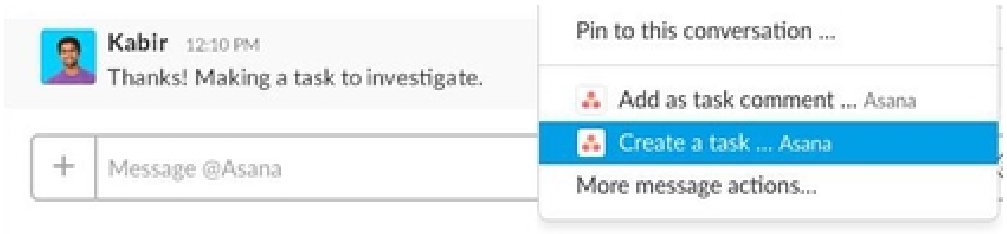

Kabir 12:10 PM
Thanks! Making a task to investigate.
Message @Asana
Pin to this conversation ...
Add as task comment ... Asana
Create a task ... Asana
More message actions...

FIGURE 3.1: Slack's action system showing Asana actions [4]

3.4.2 Project Management - Bulk Export Content

One feature which was requested by a user for Teamwork Projects was the ability to bulk export all notebooks. Notebooks are effectively large text documents which may also have comments posted by users. It is possible to export notebooks individually but not in bulk for performance reasons. Technically this plugin system could enable users to add this bulk export functionality themselves despite it not being approved. This would involve having the plugin display an option item which when clicked would request the list of notebooks from the API, individually export all notebooks and then merge them into a single ZIP file. Said option item is illustrated in Figure 3.2.

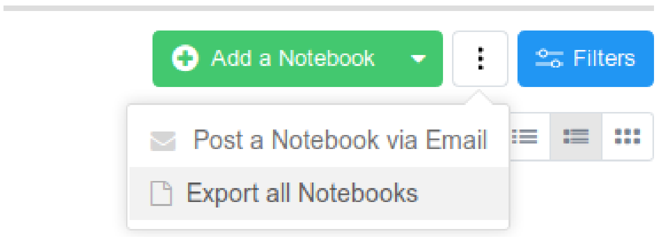


FIGURE 3.2: The option menu shown in the notebooks section would facilitate adding extra items

3.4.3 Project Management - Board Cards Export

This plugin system would allow for the addition of an 'Export to CSV' dropdown option to boards in Trello and Teamwork Projects. Trello in fact does offer CSV exports, but only for business customers. There is a Trello Power-Up plugin named 'Booklet by Vince' which facilitates exporting boards to CSV for free users [23]. In the proposed system the goal would be to design the plugin system so such a plugin could function on both Trello and Projects.

3.5 Functional Requirements

- Website operators should be able to define several areas in which they permit plugins to display content. **Rationale**: Without this, the plugin system would not know where buttons or text should be placed. There are only so many places on the site where it would make sense for placement to be permitted.

- Plugins should have the ability to add buttons in said areas, and said buttons should take the styling of the website on which they're being displayed. **Rationale**: We want plugin-placed buttons to fit in seamlessly when placed next to non-plugin buttons, hence their styles should match and custom styling would go against this decision and introduce additional design complexity.

- When a button is clicked, plugins should be able to display a pop-up list, where clicking on each item in the list performs an action. This list would take on styling of the website operator's choice. **Rationale**: Trello offers this functionality in Power-Ups. It is used by Trello's GitHub plugin to list repositories and is also used for showing Power-Up actions when a Power-Up's button is clicked [24]. If one is to expect third-party operators to implement this plugin system then it needs to contain similar functionality to that of existing systems.

- Plugins should be capable of displaying pop-ups containing HTML elements styled however they desire. **Rationale**: The other requirements allow one to add buttons and text but they don't allow custom CSS or embedding of forms. Plugin developers may want more flexibility in how they can design their plugins Allowing iframe pop-ups would allow any HTML elements to appear in the pop-up with custom styling, thereby providing this desired flexibility. Additionally, this is used by the plugin management dialog embedded in all third-party websites.

- Plugins should have the ability to request external API endpoints. **Rationale**: If someone wanted to integrate their service into a website which is using the plugin system, they would need to be able to access their service's REST API. This API is external from the plugin's perspective. For example, if one wanted to retrieve calendar data from an external service in order to display it in a plugin, this requirement would be needed to be fulfilled first.

- Plugins should be capable of modifying/retrieving data from the API of the website on which they are being displayed. **Rationale**: A plugin which adds CSV export functionality to a website would need to be able to retrieve the data from the website so it could be exported. Giving plugins this functionality would also help facilitate the hard plugin usage scenario mentioned in Chapter 2.

- Plugins should be able to programmatically determine the site on which they are hosted. **Rationale**: If one is making a plugin compatible with several sites, the REST API endpoints on the various sites may differ so the plugin would need to be able to check what site it is dealing with so it could request the correct endpoints.

- Websites which allow plugin functionality should be allowed restrict internal REST API access on a per-plugin basis. **Rationale**: Plugins should not be able to make unauthorized API requests. Websites implementing the plugin system will not want every user-created plugin to be capable of resetting the active user's login details, for instance.

- Plugins should be able to modify the properties of any UI elements they have previously rendered. For instance, a button's label should be able to be changed with minimal effort. Likewise, it should be possible to customise a dialog's contents even after it is rendered, and changes should show instantly. **Rationale**: Plugins may want to fetch data from the REST API and update the UI afterwards. In this case the UI may have previously been rendered, but the plugin may want to re-render a portion of it.

3.6 Non-Functional Requirements

- A plugin written for one website should require minimal configuration to make it compatible with another website. **Rationale**: Plugin developers may not bother to make a plugin compatible with lesser known websites if supporting the additional sites meant making substantial changes to their plugin's code. Plugins need to be compatible with as many sites as possible as that is the main selling point of the proposed system.

- Website operators should be able to implement the plugin system on their website with minimal changes to their existing code. **Rationale**: If the plugin system was difficult to implement and required a lot of changes on the providers part, said providers would be less inclined to bother implementing it and would instead end up designing their own plugin solution.

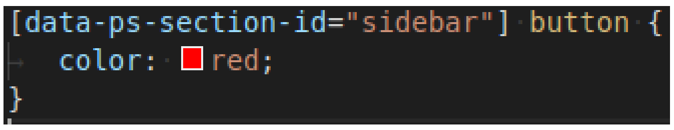
- Plugins should be secure, incapable of directly modifying the DOM (the third-party website's webpage contents). Plugin code should only ever execute under a sandboxed environment. **Rationale**: It is unlikely third-parties would implement the system if plugins could introduce vulnerabilities in their website. They need to be able to trust that any malicious plugin code is isolated and said code will not affect other plugins or site functionallity.
- Plugins should provide an API supporting translations, meaning they display text in the user's preferred language. **Rationale**: This would be of particular benefit for larger websites offering services for users located across the globe as not everyone speaks English. It would ensure the plugin's language matched the user's language which they specified in their profile on the third-party website.
- The plugin library should have minimal load times. **Rationale**: Websites and all plugins using Pluggy will be including its libraries. If these libraries are large and load slowly, it will impact the performance of the website in general and people will not be likely to use the system.

3.7 Plugin Components

3.7.1 Buttons

Plugins may want to add buttons to drop-down menus and sidebars in existing sites. However, these buttons need to match the styling of the buttons on the site they are being shown on. Different sites may have different styles for buttons, and various sections on sites may use different styling for buttons. For instance, a sidebar may contain ordinary looking buttons while a drop-down menu may contain menu items. Menu items are buttons but styled differently. The difference between the two is CSS styling and HTML markup, but on a fundamental level they both have labels, click callbacks and sometimes icons. Plugins are not permitted to supply custom HTML markup on how they would like buttons to appear, as this would be allowing any plugin to introduce

XSS vulnerabilities on sites using the system. However, the site itself can configure button CSS on a per-plugin and per-section basis as the system inserts HTML5 data attributes such as data-ps-plugin-id and data-ps-section-id and these can be used in the button CSS selectors. Each plugin has its own unique ID and likewise sections also have unique IDs. This custom CSS allows sites to style sidebar buttons differently to that of menu items. In Figure 3.3 we change sidebar buttons to show red text.



```
[data-ps-section-id="sidebar"] button {
    color: 🟥 red;
}
```

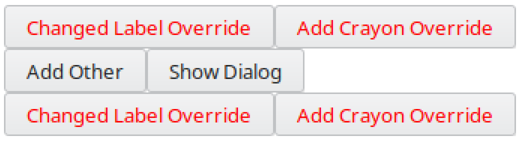
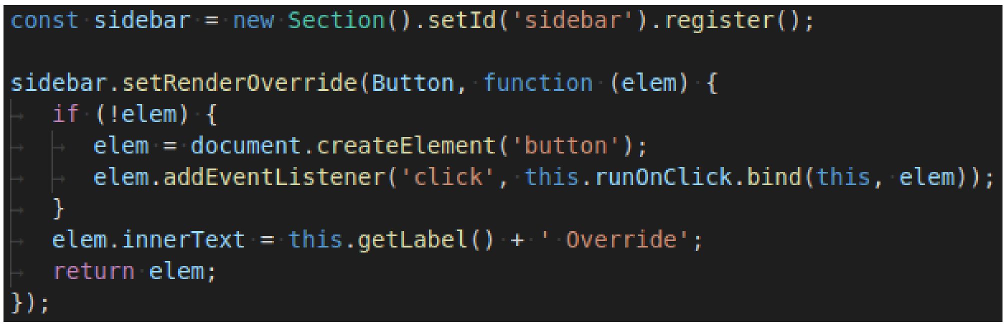


FIGURE 3.3: Left: What a CSS section override may look like. Right: What the rendered section may look like with applied overrides.

If the site wants buttons to be using a menu item structure such as <li><a></li> instead of simple <button> elements, the system is configured in such a way as to allow the site operators to change the rendering code on a per-section basis, allowing the site to specify custom HTML markup. This custom HTML can be set by the site using a function named setRenderOverride. The result of the overridden markup is shown in Figure 3.3 - note the 'Override' suffixes on the buttons. The code performing the overriding is shown in Figure 3.4.



```
const sidebar = new Section().setId('sidebar').register();

sidebar.setRenderOverride(Button, function (elem) {
    if (!elem) {
        elem = document.createElement('button');
        elem.addEventListener('click', this.runOnClick.bind(this, elem));
    }
    elem.innerText = this.getLabel() + ' Override';
    return elem;
});
```

FIGURE 3.4: What JS code which overrides HTML markup rendering may look like

3.7.2 Dialogs

Trello offers two different dialog components:

- Modals: Modal dialogs. These are dialogs which show a grey overlay covering the page around the dialog, prevent interacting with outside elements.

- Board bars: Non-modal dialogs which stick to the bottom of the page and take up the full width. Elements outside of the dialog can be interacted with and there is no overlay.

The main difference between the two is a modal prevents the user from interacting with outside elements, whereas a board bar doesn't prevent outside interaction. Modals are centered, whereas board bars consume full-width and stick to the bottom of the page.

To cater towards users of both types of dialogs, we can create a generic dialog whose horizontal and vertical positioning, width and height can be set. The dialog can also be easily toggled between modal and non-modal allowing for similar functionality to that of both modals and board bars.

As a bonus, Trellos dialogs allow only one callback which fires when the dialog closes, whereas the designed system will allow for when the dialog is opened (onShow), its iframe is loaded (onFrameLoad) or the dialog closes (onHide), providing plugins more events to work with. There are practical purposes for allowing these extra events, for example one might want to have the dialogs title show Loading... while its contents are loading. The iframe load event would allow for this.

Uses of dialogs include:

- Users may want to create plugins which show cookie consent and GDPR notices on the sites page. Some sites implement this as a non-modal dialog on the bottom of the page, whereas other sites implement it as a modal dialog on the center of the page, hence the plugin system needs to allow creating dialogs in either style.
- For the plugin directory, the aim is to make it so clicking on a button on the third-party site shows a popup iframe with plugins one can enable, centered on the screen. A vertically and horizontally centered fullscreen modal dialog would be used in this case.

Chapter 4

Implementation Approach

4.1 Possible Approaches

There are a number of different means in which one could implement a plugin system, but not all of them are viable. In order to show the process of deciding which approach we went with, we outline what our various options were and any issues we noticed with said options.

4.1.1 Pure JavaScript

We could have plugins execute JavaScript directly on the web-page on which they are enabled. This JavaScript would be used for instance in order to add plugin buttons to the third-party site. However, this design would allow malicious plugins execute arbitrary JavaScript code, steal user cookies or perform unauthorized API requests. There would also be the possibility of legitimate plugins which just happen to contain security vulnerabilities. Given one cannot trust the contents of a plugin, it is not viable to simply run the plugin directly on the website. Third party providers would not trust our plugin system if one designed it this way. Hence one needs to run plugin code in a sandboxed environment in which plugins are given limitations preventing unauthorized API usage.

4.1.2 Web Workers

We could work around this security issue by placing the code within a web worker and preventing access to sensitive JavaScript API functions or variables. Web workers are essentially background threads one can create in JavaScript. They do not have permission to access the DOM meaning they cannot add elements without permission, view cookies or cause page redirection. This seems like a viable solution and because it prevents access to the browser's window, it serves as a reasonable security mechanism and prevents plugins redirecting the web page or inserting arbitrary HTML elements into the page.

However, one major issue is it remains possible to execute AJAX requests which perform any action on the third-party website using the logged in user's cookies. This issue would make it difficult to restrict access to certain endpoints using a permission system. One could work around the issue by designing the third party website using cookie-less sessions, but it is assumed one has no control over the third party website's design. Given this issue, the web worker approach does not appear to be viable from a security perspective.

4.1.3 Iframes

Another potential option would be to have each plugin be contained in its own iframe. The benefit of this approach is plugins would would have virtually no chance of performing any unauthorized actions. Iframe sandboxing would prevent plugins from modifying the DOM, retrieving cookies, making alert dialogs or otherwise causing disruption to the end-user's experience. The downside would be that it adds a layer of complexity which would not exist in the case of a pure JS approach.

The concept is to have each plugin's code be contained in a hidden iframe. This iframe communicates with the third party website telling it to add buttons. When said buttons are clicked on the third party website, the website notifies the plugin and the button's onClick callback is executed by the plugin within the iframe. Hence document.write() would not cause any issues as it would only affect the invisible iframe as opposed to the third party website itself. The plugin is contained. Each plugin/iframe would also need to be hosted on a separate subdomain, so as not to have plugins leak localStorage or cookies to each other.

Going with the above approach introduces a limitation however. Plugin-called functions which return data from the third-party page would need to be callback-based. Functions could not return immediately and would instead need to go through the process described above. For example, if a plugin wanted to obtain details from the web page instead of using the site's API, the plugin might call getUserInfo. The web page is notified the plugin wants details and the website then notifies said plugin with the user's details. This is done to work around sandbox restrictions being introduced by the web browser as plugin iframes are on a separate domain to third-party websites and cross-domain interaction is forbidden by the same-origin policy [25]. One can work around the issue of functions not returning values by instead using callbacks or promises, so this is just an additional complexity to bear in mind. It still nonetheless remains a better option than either pure JavaScript or iframes from a security perspective.

4.2 Google Caja

Google Caja is a plugin sandboxing solution developed by Google, used in its own products such as Google Docs. Google developed it as an easier means of accomplishing the iframe method mentioned above. It makes use of what Google refers to as 'virtual iframes' and acts as a JavaScript compiler. It appears to send plugin code to a Caja server which then returns the sandboxed equivalent of said code, where the code is effectively given its own DOM so it cannot modify elements outside it. This is similar to how iframes operate.

The main benefits Caja has over the iframe method are:

- Because Caja is not making use of iframes, any and all elements in the sandbox inherit the styling of the parent site.
- Caja allows the parent site to easily pass functions which can be used by the sandbox. It does so without requiring either the sandbox or parent site to use a postMessage mechanism.

The downside is that Caja's compiler approach introduces the potential for security vulnerabilities. One security researcher found 6 different means of bypassing Caja's sandbox [26, 27], some of which were caused by ES6 additions, and was therefore capable of executing XSS on any Google sites which included Caja. Caja allows plugins greater control over the content they display, but introduces a greater chance of XSS. Additionally, given the goal is to allow plugins show buttons in several different sections, and all of these sections could theoretically be visible on a page at a given time, it makes more sense to go with the hidden iframe approach.

4.3 Technologies

A number of technologies are required in order to develop a system which meets our design requirements. Mentioned below are the various technologies we plan on using throughout the development cycle of the plugin system and its test site.

4.3.1 Hyper Text Markup Language (HTML5)

HTML5 is the language which will be used when developing the layout of the test site. Essentially we write code in this language and it's converted from textual form into visual elements. Buttons and text are examples of HTML elements which a page comprises of. They are displayed on the test site and are created by plugins. We will be working HTML v5 features specifically such as iframe sandboxing which is explained later.

4.3.2 Cascading Style Sheets (CSS3)

CSS3 is a language which allows one write text in order to change the appearance of HTML elements [28]. It will be used for changing the appearance of plugin-inserted widgets and the list of enabled plugins. Third-party websites will use CSS to specify how they would like plugin-created buttons to appear.

4.3.3 Iframes

Iframes are essentially HTML elements which allow one to embed a third-party web page into their own website. We wrap our plugins in an iframe as JavaScript code in an iframe executes in its own context and will be less capable of the third-party page on which it is being embedded. In simple terms, it reduces the likelihood of malicious plugins executing JavaScript code on our website. Iframes are also used in the pop up modals we permit plugins to create, allowing for any elements in any styling to be displayed in them.

4.3.4 Iframe Sandboxing

By default iframes are prevented from executing code on third party websites on which they are embedded. However, iframes (our plugins) would still remain capable of performing actions such as redirecting the third-party website to another URL or creating pop up browser windows which would irritate users. To prevent these situations, a HTML5 feature known as iframe sandboxing is used [29]. It acts as a security layer, preventing this embedded frame from performing actions which would be deemed a security risk to the site on which the page is being embedded. It is essentially an attribute one can add to an iframe element. Once added to the iframe, all security-sensitive actions including submitting forms and in-frame JavaScript execution are prevented, with us having the option of individually whitelisting them at our discretion. We want frames to be capable of running JavaScript code as otherwise plugins would not function, but we do not want them being able to create pop ups or redirect the web page. The iframe sandbox attribute makes this easy to accomplish.

4.3.5 JavaScript (JS)

JavaScript is a language which facilitates executing code. HTML allows one to create a static web page, but for any interactivity one must write JavaScript code. The plugin system we are developing is predominantly JavaScript-based. We will use JavaScript when performing communication between the plugin and the third-party website. The third party website will leverage JS is used in order for the third-party website to communicate with the plugin iframe.

The following JavaScript features will be required when implementing the plugin system:

4.3.5.1 postMessage API

Our plugins are contained within iframes on a separate domain to the third-party websites. The only means in which they can interact with the website is via what is known as the postMessage API. Essentially this allows windows on the same page to send messages to each other [30]. When a plugin wants to create a button, it will send a message to the server, and the server will create the button outside the iframe sandbox. Likewise, when the button is pressed, the third-party website sends a message to the plugin frame and the plugin executes the button callback from within its sandbox. The postMessage API is what facilitates this entire messaging procedure and is a core component in our system as we would not otherwise be able to easily communicate between the plugin frame and third-party page.

4.3.5.2 MutationObserver

When designing our plugin system, we need to factor in the possibility of a section not existing at the time of plugin initialisation, meaning widgets cannot immediately be added to said section. Websites such as Teamwork Projects use front-end frameworks which load a page once and choose to dynamically render elements on the page when areas on the navigation are clicked. This means that our functions to add buttons to sections will need to monitor for the existence of such a section. Whenever sections come into existence, only then should attempts be made to add the buttons to them. MutationObserver is a JavaScript API which essentially allows one to listen to changes made to the DOM, such as when a HTML element is created or modified [31]. It is what we will use to detect new sections. Dynamic sections are actually a common scenario, as websites running front end frameworks

4.3.5.3 LocalStorage

LocalStorage is a place where plugins can store plain-text key-value data in web browsers [32]. It is used in the plugin system in order to keep track of which plugins a user has enabled for a site. The user ID and enabled plugin IDs are stored in the third-party site's browser-specific local storage when plugins are enabled. Whenever the third-party site is loaded, the system checks for plugin IDs in localStorage and loads them if any are present.

4.3.6 Golang

Golang is a programming language developed by Google engineers, often used to design REST APIs. Benefits of using Golang over NodeJS include Golang having reduced memory usage and compile-time error checks which facilitate in creating more stable software. The plugin system uses a REST API developed in Golang. This API can list plugins available for specific websites and allows users to upload, edit or delete their plugins.

4.3.7 GORM

GORM is an Object-Relation Mapper (ORM) library for Golang [33]. This means it allows Golang communicate with SQL databases without needing one to write SQL. Instead, one can use simple Golang objects instead of tables and modifying said objects or creating new ones performs SQL queries without the developer needing to write them. This is used in the plugin system as from personal experience it is easier to use an ORM than deal with raw SQL. It keeps track of user-uploaded plugins and plugin metadata.

4.4 Components

4.4.1 Plugins

Plugins are invisible iframes inserted by the plugin system library (explained below) into the third party website. Plugins include the plugin library described in the next section. Their iframe design serves as a sandbox, meaning whenever a button is interacted with, for instance, the click callback will execute within the iframe rather than the parent site. This solidifies third party trust in the plugin system as plugins cannot cause any site-breaking effects as a plugin's unsanitised code is never executed on the third party

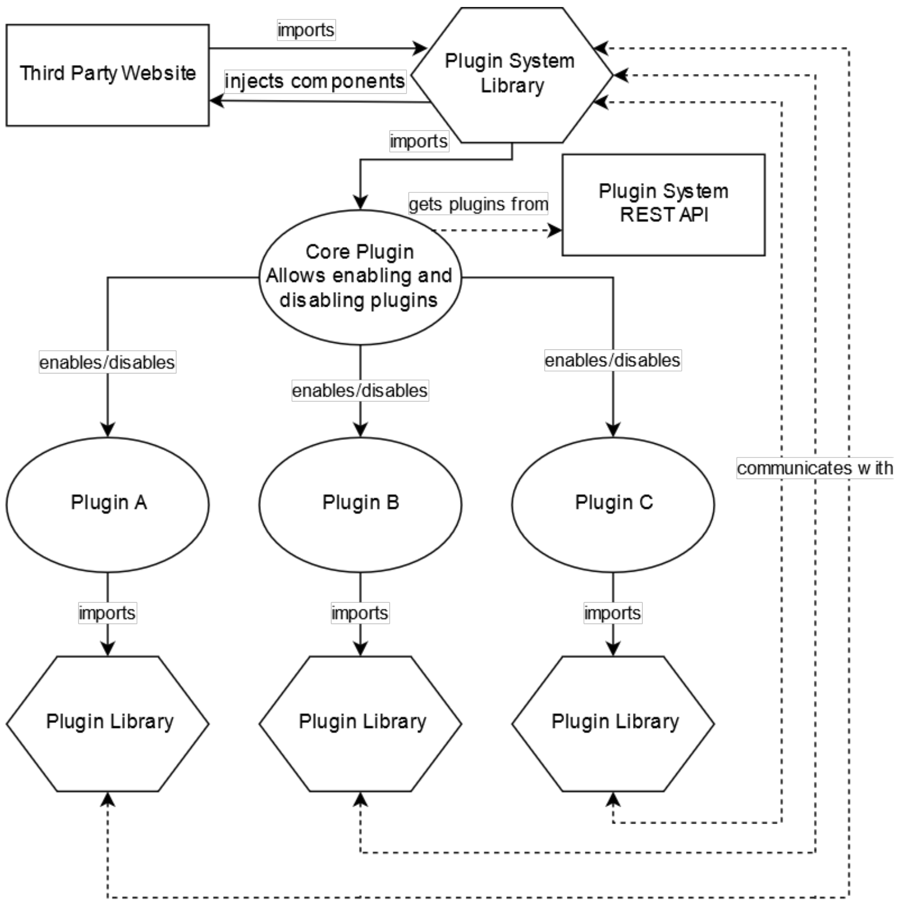


FIGURE 4.1: Rough outline of the various plugin system components

site. Instead helper functions such as addButton and addWidget are exposed to these iframe plugins, allowing interacting with the site in a more restricted and secure manner.

4.4.2 Plugin Library

Each plugin includes this JavaScript library. Plugins call functions in this library's API, such as addButton, and then the plugin library sends a postMessage from the plugin's iframe to the third-party website. This message is intercepted by the plugin system library included on the third-party site. The plugin system library has its own addButton function that adds the button to the website, outside of the plugin's iframe sandbox. So, in a way, the functions in the plugin library are like an interface whose functions invoke the plugin system library functions using the postMessage API. It is just a way of accomplishing the postMessage communication without requiring plugin developers to be as aware of it.

4.4.3 Plugin System Library

The plugin system library is a JavaScript library included by all websites implementing the plugin system. When the system library is loaded, this library injects any plugins (iframes) the user has enabled. Enabled plugins could be tracked via either localStorage or a REST API. This library has functions which allow plugins create custom widgets and buttons in the website itself, outside of the plugin sandboxes. When a button is clicked, the plugin system library notifies the plugin library (explained below) included by the plugin, which in turn executes the plugin click callbacks from within the plugin's iframe.

4.4.4 Third Party Website

The third party website includes the plugin system library previously described. The website would then add HTML attributes such as data-plugin-sectionId and data-plugin-sectionType to the container elements it wants to allow buttons or widgets to be added to. The sectionType might be 'buttons' for instance, in which case only plugin-created buttons would be permitted in this section. The third party website would also specify what appearance plugin-created buttons would take.

4.4.5 Plugin System REST API

The plugin system has its own website on which user-created plugins are hosted. This website has a REST API which allows one to upload plugins as ZIP files and retrieve a list of plugins compatible with a specific website. The API allows logging in and registering using standard login, and once logged in one can upload, edit and delete plugins. Time permitted, a nice to have feature is if the API could remember which plugins are enabled when logged in so the user wouldn't have to enable plugins on a per-browser basis.

4.5 System Characteristics

4.5.1 Framework Agnostic

Front-end frameworks will not be used in the plugin system design itself. The plugin system is programmed entirely in JavaScript and contains no HTML. Plugins on the other hand may use any JavaScript framework so long as the plugin code has been compiled to JavaScript by the plugin developer.

4.5.2 Easy For Third-Parties to Implement

If a third-party website simply wants to allow plugins which add buttons or widgets, said website need only include the plugin system library and add data-plugin-sectionId and data-plugin-sectionType to the areas where custom elements may be inserted, where sectionType would be either 'buttons' or 'widgets'. If desired, the third-party site operator can specify the styling for the buttons and other plugin-created widgets. As the widgets are embedded on the site itself and not in the plugin iframe, site operators can simply adjust the styling for them using regular CSS stylesheets on their site.

4.5.3 Capable of Handling Dynamically-Generated Sections

Sites making use of frameworks such as VueJS and ReactJS often dynamically create sections when buttons are pressed. When a plugin attempts to insert a widget into the element, the system will need to check that the section exists, and when the plugin comes in or out of existence due to dynamic rendering used in front end frameworks the plugin system must detect these DOM changes and insert the requested plugin-created elements.

4.6 Methodology

4.6.1 REST API Authentication

The plugin system's REST API allows fetching a list of available plugins for a given site. Users can register an account or log in, in which case they would be allowed to upload new plugins or edit or delete their own plugins.

Originally the plan was for the REST API to allow OAuth login and keep track of enabled plugins in the API. However, then it was discovered that despite OAuth being a standard, every site implements it differently. This would have meant the plugin system API would need to have code written on a per-site basis to handle every OAuth system and would have introduced a lot of maintenance and taken more time than the project would permit. It was not a practical approach.

Additionally, each third-party site implementing the site would have needed an OAuth implementation, which in hindsight would have been setting the barrier for entry too high for basic plugin usage.

Instead, it was decided to require users to maintain separate login details for the plugin system itself, and make use of localStorage for keeping track of enabled plugins. Enabling a plugin stores its ID in localStorage so enabled plugins are remembered so long as the browser's data isn't manually cleared by the user. This means enabled plugins would be enabled on a per-browser basis. If time permits, logging in will also keep allow one keep track of enabled plugins across different browsers meaning it would only use localStorage if not logged in.

4.6.2 Plugin Load Procedure

Described below is roughly how plugins load on the page:

- User visits third-party website which includes the plugin system library.
- Plugin system library checks which plugins are enabled for the website and injects their iframes into the page
- Once a plugin loads in iframe, it adds buttons, etc. to the third-party website via the plugin library and plugin system library. The plugin library tells the plugin system library it wants to add a button, and the plugin system library adds it to the website's DOM and acts as a proxy for click events, forwarding them to the plugin library so the handlers are ran in the iframe context.

4.6.3 How Buttons Work

Buttons are inserted and clickable via the following process:

- Plugin tells website to add button via window.postMessage JS API. This is abstracted by the plugin library, so developers only need to know the addButton function and its arguments. It ultimately runs postMessage and the plugin system library catches the message and runs its own addButton function no the website itself.
- Website creates button with the details the plugin specified (id, label, icon)
- User clicks the newly created button on the website
- Website via the plugin system library tells the plugin which created the button that it was pressed
- Plugin executes onClick callback, so it's ran from within the frame meaning document.write() won't affect the web page and will instead only affect our hidden frame.

4.6.4 How to Implement in Third-Party Sites

The system's primary selling point is that it can be easily implemented across a number of sites with minimal configuration required. Therefore, to demonstrate the system's flexibility, one must show it functioning on a third-party site to prove the concept works.

Technically one could spend time designing an interface which lets our system's plugins work in existing plugin systems such as Trello Power-Ups. However, this would be too complex given the project's time constraints.

A relatively easy means of proving the system works would be if one was to make a Tampermonkey user-script. User-scripts allow users execute their own JavaScript code which executes when specified websites, enabling users add their own functionality to sites they have no control over [34]. Tampermonkey is an extension for most modern browsers which supports loading these user-scripts. If one had control over the third-party site, they would simply modify the source code to import the plugin system library on the site and add the plugin HTML attributes previously mentioned to the various customisable sections. Given one does not have such control, one could instead create a Tampermonkey user-script which executes when loading the site, imports the plugin system library and adds the plugin attributes using JavaScript. Plugins would then load normally as if the site operators had added the system to their site themselves, as the plugin system library has been included.

4.6.5 How to Enable or Disable a Plugin

If a site implements the plugin system, one could make it so that there is one core plugin which is enabled no matter what, even if the user has no specific plugins enabled. This plugin would add a button to third-party site, and clicking said button shows a popup modal with a list of plugins which function on the site.

A benefit of designing the plugins list as a plugin is third-party site operators could customise its design to their liking by forking its code. It also serves as an example of what plugins can be created using the system, providing aspiring plugin developers with insight into how they would design their own plugins.

This plugins list is obtained from the plugin system's REST API. Next to each item in the plugin list is a button which says 'enable' or 'disable' depending on if the plugin is already loaded for the logged in user. Enabling or disabling a plugin adds or removes plugin-created elements such as buttons in real time, without requiring the current tab to be refreshed.

4.7 REST API Endpoints

Plugins can be listed without requiring the user to be logged in, but all other endpoints (editing plugins, uploading new plugins, deleting plugins) require logging in.

Table 4.1: Plugin system REST API endpoints

Method	Route	Params	Purpose
POST	/plugins	siteIds: what sites plugin works on zippedFile: plugin files, zipped	Publish a new plugin (Must be logged in)
GET	/plugins	siteIds	Get a list of plugins given site
PUT	/plugins/:id	siteIds, zippedFile	Update an existing plugin: - change its supported sites - change its files (Must be logged in)
DELETE	/plugins/:id	*N/a*	Delete a specific plugin (Must be logged in)
GET	/plugins/:id	*N/a*	Get a specific plugin's details
POST	/users	email, password	Registers a new user
POST	/session	email, password	Log in as existing user

4.8 Cross-Frame Communication PoC

Shown throughout the appendix are screenshots of basic proof-of-concept (PoC) code showing how plugins would insert elements into the parent site. A test website, shown in Figure A.4, has two sections highlighted in red and yellow respectively. Injected on the website are two instances of an iframe plugin which each add a button to the red section on the site, outside the sandbox. This demonstrates the button creation process described in the methodology section, while also showing plugins do not affect each other as clicking the button corresponding to one plugin does not affect the other plugin's iframe.

In the concept shown in Figure A.1, postMessage called by the plugin iframe resulted in a button being created on the parent site. In the proposed design, instead of a plugin developer executing postMessage, there would be an addButton function. The developer would not need to know that it uses postMessage internally. The plugin system in Figure A.2 is what the third party site would import. It handles the postMessage sent by the client, and has its own addButton function which creates the button on the site itself and forwards the button click callbacks back to the plugin.

For illustrative purposes the iframes are visible in this PoC to indicate that the button callback is executing in the plugin's iframe rather than the web page, whereas in the final design the iframes would be hidden, existing only for their sandboxing capabilities.

4.9 Golang Per-Subdomain Plugin Hosting PoC

Golang is used to host each plugin on its own unique subdomain. This is done so as not to share localStorage or cookies between plugins. Plugins on separate domains cannot steal sensitive information from each other. Figure A.5 is Golang code which handles displaying different files depending on which subdomain is visited.

4.10 Wireframe

Shown below is a wireframe depicting the plugin list core plugin. The core plugin tells the plugin system library to inject a button on the third-party site. When pressed, this button opens a modal which lists site-compatible plugins which can be enabled or disabled. An option to upload new plugins is shown in the modal dialog.

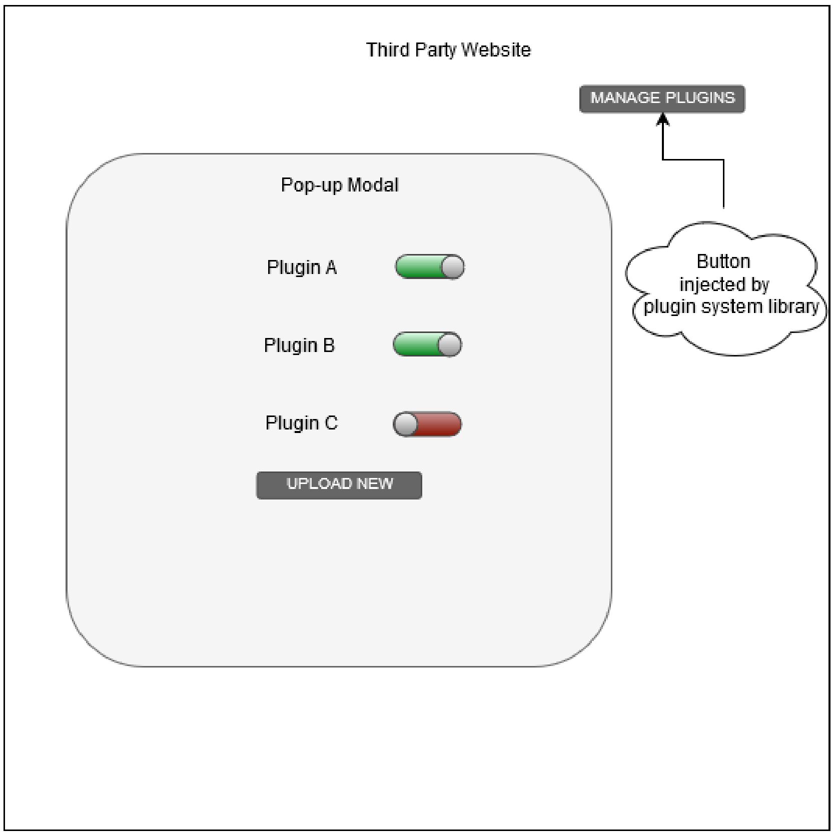


FIGURE 4.2: Wireframe showing the plugin list in a popup modal dialog

4.11 Prototype

The 'Manage Plugins' button described in the wireframe section above might look like the below when displayed on Teamwork Projects. The plugin-created button is highlighted in red for clarity.

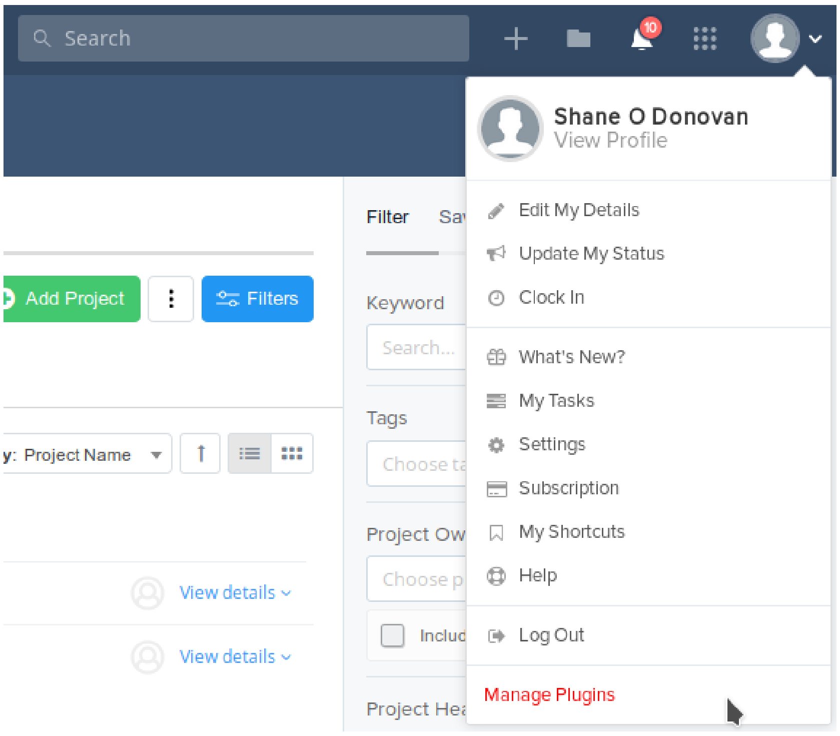


FIGURE 4.3: Prototype showing how the 'Manage Plugins' plugin button might look if shown on Teamwork Projects

4.12 Implementation Plan

TABLE 4.2: Rough workload plan

Weeks	**Work Plan**
1	Create the plugin system's REST API with fully functional standard login, support for uploading and listing plugins
2-4	Create a barebones plugin library and plugin system library.
5	Implement the plugin system in a third party website, e.g. Trello.
6	Implement the plugin system in another website, e.g. Teamwork Projects.
7-8	Create a core plugin which shows a popup of plugins one can add, enable and disable. Retrieves plugins list from REST API previously made and records enabled plugins in localStorage.

This project would be considered successful if the tasks mentioned in the above work plan table have been completed by the end of the semester along with API documentation and all requirements listed in Chapter 3 fulfilled.

Week 1: Only one week is allocated for the REST API as I have written similar REST APIs previously so it should prove reasonably easy to implement.

Weeks 2-4: In this time, a plugin library and plugin system library which communicate with each other, allow plugins make API requests to the third party site. Plugins can create buttons and display popup modal dialogs.

Weeks 5-6: The plugin system needs to be implemented in at least two websites to demonstrate how it can easily be implemented on multiple websites. If possible, the idea would be to make a plugin compatible with both Trello and Teamwork.

Weeks 7-8: On each website implementing the plugin system, there needs to be a list of plugins which can be enabled, an option to enable or disable plugins and upload new plugins. This list will be implemented as a core popup plugin which cannot be disabled by users.

Weeks 9-12: This is allowing time for things to go wrong in the 8 weeks prior. Time would be spent documenting the system, and if time permits, additional time could be spent expanding the feature set of the plugin system or creating more advanced plugins.

Chapter 5

Implementation

5.1 Difficulties Encountered

There were a number of difficulties encountered while developing the plugin system solution. These difficulties are described in tables using fields outlined in Table 5.1.

Difficulties encountered during the implementation phase are outlined in Table 5.2, Table 5.3 and Table 5.4.

Table 5.1: Descriptions of the fields used when outlining difficulties

Difficulty	One-line description of the problem being faced
Classification	How challenging was the issue to overcome? 'Easy', 'Medium' or 'Hard', where hard means it couldn't be handled
Description	In-depth explanation of the problem
Impact	How did the problem or its solution affect the implementation stage?
Solution	How was the problem addressed, or how could it have been addressed?

TABLE 5.2: Difficulty writing complex GORM queries

Difficulty	**Learning how to write more complex queries in GORM**
Classification	Easy
Description	GORM is an ORM (Object-Relational Mapping) for Go. This means it allows Golang applications to communicate with databases using classes and functions instead of writing SQL code. Golang is used by the plugin system's REST API, which keeps track of uploaded and enabled plugins. This author had some knowledge of GORM, and it sped up the development process for simpler queries in the Plugin API. However, when it came to more complex queries, such as cascade deleting plugins or sub-queries, GORM made development significantly more difficult than it would have been if one was to simply write raw SQL queries. Several hours were spent trying to find GORM solutions when the SQL solution was already known.
Impact	This pushed the implementation schedule back a few hours as it was unexpected and time was not budgeted for it.
Solution	Eventually, this author found GORM solutions for complex queries encountered. If one was creating this API in future however, it would be a good idea to consider opting for raw SQL queries instead of using an ORM as there is far more documentation available on SQL than there is on GORM.

TABLE 5.3: Difficulty maintaining compatibility with older browsers

Difficulty	**Maintaining compatibility with older browsers (IE11)**
Classification	Medium
Description	Internet Explorer 11 is supported by both Teamwork Projects [35] and, until July 2019, Trello [36]. However, IE11 doesn't support ES6, a JavaScript standard used in this project, meaning Pluggy simply wouldn't function. Likewise, there is CSS used in this project which doesn't function on older browsers, so elements weren't being styled correctly.
Impact	This affected the architecture of the solution, making it necessary to use additional tools in the building process and overall increasing the size of the library. This easily cost the project several days. The PostCSS solution mentioned below took several hours alone to find and implement.
Solution	Babel, as mentioned in the previous chapter, needed to be used so as to convert the JS ES6 code into ES5 code which IE11 understands. In addition, PostCSS in combination with a CSS Auto-prefixer was used to convert some newer CSS code into older CSS suported by IE11. Even with both of the above solutions, there still remained instances where JS or CSS code would inexplicably not work on IE11. The result was that JS or CSS code needed to be rewritten on a few occasions or ugly IE11-specific hacks needed to be implemented.

TABLE 5.4: Difficulty having plugin libraries load quickly

Difficulty	**Having the plugin libraries load quickly**
Classification	Medium
Description	One of the goals in this project was to have the file size of the libraries be as small as possible to minimise load times. Each plugin imports the plugin library, so with 5 plugins on a page, one would be loading the library 5 times. Hence the library needed to be as small in size as possible to accommodate for this. However, to maintain IE11 compatibility, as described previously, Babel was used. Babel transpiles JavaScript, essentially converting it from the newer ES6 standard to an older standard used in outdated web browsers. This comes at the cost of an increased file size whenever newer features are used. This increase in file size would have an impact on load times.
Impact	This affected the architecture of the solution, as one constantly needed to monitor the impact minor changes had on file size and make code changes accordingly. It slowed down development for the entire duration of the project as one always needed to be conscious of the issue.
Solution	Whenever any significant code changes were made, the JavaScript code was transpiled by Babel and its old file size compared with the new file size. At one point a for .. of loop was used instead of a traditional for loop for iterating over array items. Choosing the former for loop resulted in Babel increasing the file size by roughly 0.3KB. One can imagine given the codebase would include dozens of loops, this could cause a considerable increase in file size. A conscious effort was made to use traditional for loops exclusively and try to avoid using as many features which would require polyfills as possible. As a reminder, polyfills are essentially pieces of JavaScript code written to emulate newer JavaScript features. Using the 'URL' interface to perform relative-to-absolute URL conversion would have instantly increased the file size by roughly 10KB for instance. Only 3 polyfills were used, and they were only used as they were absolutely necessary and there was no simpler alternative. Additionally, JS and CSS minification needed to be performed to further reduce file sizes. A check was made to ensure that all servers serving this content were set up and using GZIP compression. GZIP ensures even if a file is large, it is compressed to reduce its size so it loads faster.

Table 5.5: Difficulty having plugins support translations

Difficulty	**Adding translation support for plugins**
Classification	Hard
Description	Originally it was desired for plugins to be able to display translations, so if a website was to change its language its language then plugins would update button text to show in the appropriate language. However, time constraints mean that this was not possible to implement.
Impact	This meant that the project did not achieve all of its non-functional requirements.
Solution	There is no solution for this as such. In reality this should never have been a requirement, and should instead have been considered a nice-to-have feature. It seems almost inaccurate to consider this 'Hard', as in reality this author expects it could have been implemented in a few hours. The issue is there simply wasn't enough time to implement it. Implementation would consist of adding a new tl() function to the plugin library. This function would check what the language the website was using. In the event of the website changing its language, the system library would re-render all sections, causing the translations to update.

Table 5.6: Difficulty showing pop-up dialogs with clickable list items

Difficulty	**Showing pop-up dialogs with clickable list items**
Classification	Hard
Description	One of the desired features for plugins was for them to be able to display pop-up lists. These lists would allow one to click on an item and have the plugin perform an action. It would have been a list of buttons essentially. However, time did not allow for this feature to be implemented as it would have taken several days to implement.
Impact	This meant that the project did not achieve all of its functional requirements.
Solution	As with the previous difficulty, there is no solution for this either. There simply was not enough time to implement this feature. Implementation would have been complex, and involved making a new kind of Dialog class across both the plugin library and plugin system library. Additionally, there would need to have been a ListItem class made for use with the new dialog. This class would also have needed to exist in both libraries. It would have taken several days to implement the feature.

5.2 Architecture

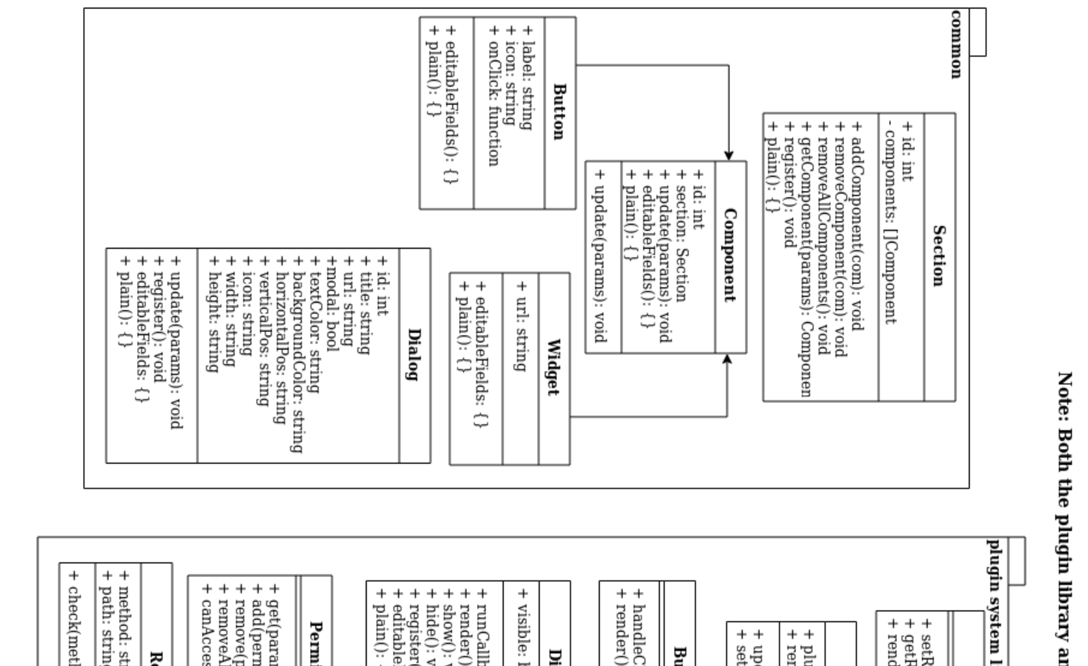
The implementation approach described in Chapter 4 is largely the same as the approach which was ultimately taken. To summarise Chapter 4, the main components in the system are described in Table 5.7. For a more visual representation of the component structure, see Figure 4.1. The structure did not change in the implementation stage.

5.2.1 Plugin Libraries

In order to have more of an understanding as to how the plugin library and system library work, it is worth looking at their code structures, represented in Figure 5.1.

TABLE 5.7: Description of main components in Pluggy

Component	**Purpose**
Plugins	These are hidden iframes which are injected onto third party sites running Pluggy. These plugins load the plugin library. Plugins tell the plugin library the various properties of the elements (e.g. buttons) they want created. Plugins also handle any callbacks on the created elements, such as click callbacks. As these plugins are hidden iframes with no DOM access, there is no XSS.
Plugin library	All plugins must include this library. It provides plugins with functions and classes which let plugins communicate with the system library. The plugin library for example has functions which handle the sending of element properties to the system library for elements to be created. It also handles any messages sent by the system library and sends them to the plugins. These messages include click callbacks.
system library	This library is included on any third party site implementing Pluggy. It is not contained in an iframe, it is on the site itself. The plugin library communicates with this library from a plugin iframe. This library is responsible for the creation of the HTML elements using the properties specified by the plugin library. If an element is clicked or a callback otherwise occurs, this callback is sent by the system library using the postMessage API to the relevant hidden plugin iframe so the callback can be handled by the plugin library and in turn, handled by the plugin which ordered the element's creation.
Core plugin	The core plugin is a plugin which is always enabled no matter what, with no option of disabling it. It adds a 'Manage Plugins' button onto the site which when clicked, shows a plugin management modal dialog. This plugin-created dialog allows users to enable, disable and upload new plugins. In other words, this plugin is responsible for the loading of all user-created plugins. When the page loads, and in turn this plugin loads, the plugin checks the plugin system REST API to see if any plugins are enabled and if so, it tells the plugin library to enable the plugins, and in turn, the library tells the system library which injects the plugin iframes onto the site.
Plugin system REST API	This REST API allows users to log in, upload and enable plugins. It is used by the core plugin for allowing plugin management functionality and for checking if plugins should be loaded on page load.



Each outer block in Figure 5.1 represents a different package. There are three packages: common, system library and plugin library. Classes in both the system library and plugin library extend the classes with the same name in the common package. This design approach helps reduce the amount of duplicate code when developing both libraries. Less duplicate code means the code is easier to make adapt the code in future, as said changes wouldn't need to be applied in several places.

We see many of the classes in the plugin library have the same names as the classes in the system library. The difference both libraries' classes is:

- In the plugin library, most of these classes exist solely for communication with the system library. Any property change or function invoked on each class instance notifies the system library of this action where the system library then repeats this action to retrieve data from the third-party site or modify its DOM.
- The classes in the system library have rendering functions. Class instances are created in the system library based on the properties sent by the plugin library. In other words, the system library mirrors any class instances from the plugin library. The system library uses these instances to render HTML elements on the third party website using the render() function.

A way of understanding this is, the plugin library mostly acts as a proxy, forwarding any requests to the system library where they are handled. The reason for this proxy approach is, as mentioned in Chapter 4, plugins are hidden iframes with no access to the DOM. Therefore the plugin library, which plugins import, also has no DOM access. The system library differs in that it is imported directly on the third party site and therefore has DOM access. Hence, using this proxy approach, the system library is giving the plugin library a controlled, secure means of inserting new UI elements into the DOM.

With that clarified, we can now focus on outlining the purpose of each unique class in the plugin library and system library. These are explained below.

5.2.1.1 Section

Sections are the areas on a site where the website operator has permitted plugins to create UI elements. For example, a website operator may permit buttons to show on the sidebar, but they don't want them appearing elsewhere on the site. The website operator registers a section via new Section(id: 'sidebar') and all plugins must also create a Section of the same name and associate it with any components they want to show in the sidebar. In the system library, sections keep track of section-specific render overrides, used when a site operator wants a button displayed differently for a specific section such as the sidebar. Multiple areas on a site can have the same section, such as items in a list. Several list items could exist in the section with ID 'list-item' and this would be fine. A section's render() function causes all sections the given ID of to re-render their components' HTML.

5.2.1.2 Component

A component is any plugin-created UI element associated with a section. It is a superclass to Button and Widget later described. This class is responsible for associating elements with unique sequential IDs and attaching them to sections.

5.2.1.3 Button

As its name implies, this is a plugin-created button. Buttons can have labels and onclick callbacks. When a button is clicked on the third-party website, the system library catches the click callback. It then sends the callback to the plugin library, and the onClick property in the button is executed in the plugin by the plugin library. In the system library, this class is responsible for rendering a button's HTML.

5.2.1.4 Widget

This is a plugin-created iframe one can associate with a section, in the same way buttons are associated with sections. This only existed for testing. It otherwise serves little purpose. When separating Button and Component logic, to ensure that the Component logic worked, this Widget class was created which extended it. In the system library, this class handles rendering a widget's HTML. This took virtually no time to implement.

5.2.1.5 Dialog

Dialogs are basically pop-ups which show an iframe of a specified URL on the site. They may be modal, in which case an overlay will be displayed around the pop-up, preventing outside interaction. Alternatively, they can be non-modal, in which case outside content can be interacted with. Dialogs do not extend Component as they are not linked to a section. The idea is buttons can display a dialog when clicked. The 'Manage Plugins' button in the directory modal mentioned in Chapter 4 displays the plugin management dialog which is created using this class. In the system library, dialogs HTML elements are rendered via this class.

5.2.1.6 PluginAPI

This API allows enabling and disabling plugins. When a request is made to enable a plugin, an iframe is injected directly onto the website, with the iframe's URL being the plugin URL. The system library and plugin library support this API, meaning the website administrators, plugins or plugin-created dialogs may make use of it.

5.2.1.7 Plugin

This is used internally by PluginAPI to keep track of enabled plugins and their metadata. It remembers information such as a plugin name, description, or if it is privileged. Only privileged plugins may access the PluginAPI and enable or disable other plugins.

5.2.1.8 PermissionAPI

This may be used via either the plugin library or plugin system library. When used by the plugin system library, this API is simply for registering permissions and associating them with routes. For example, an administrator could add a permission on the site via new Permission(id: 'addUsers'). When used by the plugin library, this API allows requesting permissions using the 'request' function. Plugins would request a registered permission ID and then have access to any API URLs associated with said permission.

5.2.1.9 Permission

Permissions may be added by site operators. This allows operators to offer plugins limited access to their internal API. Permissions have a unique ID such as 'addUsers' and they may be associated with multiple Routes.

5.2.1.10 Route

Routes are classes which hold a (Method, Route) structure, allowing site operators to link a specific HTTP method and URL. Permissions can have routes associated with them. For instance, in the case of an 'addUsers' permission, one may allow (POST, /users) as the method and route. It allows the operator to control what internal API URLs plugins may access. In the above case, GET /users would remain inaccessible, and only POST /users would be permitted.

5.2.1.11 Store

This is used solely by plugins and plugin-created dialogs. In simple terms, it grants plugins and their dialogs the ability to store shared data and communicate with each another. It has functions 'set', 'get', and '(un)subscribe'. 'Set' stores a key,value pair on the plugin's site. Even if a dialog is hosted on a separate domain, whenever it calls 'set', the data will be stored on whatever domain the plugin is on, not the dialog. 'Get' likewise retrieves the value at a given key on the plugin's site. Plugins and dialogs may subscribe to a key and detect when its value changes via the 'subscribe' function. This API is used in one of our later examples in order for a settings dialog to close when a 'Save' button within it is pressed. Dialogs cannot close themselves but their plugin may close them. In the settings dialog example, the plugin subscribes to the 'settingsSaved' key and when the 'Save' button is pressed within the dialog, the dialog changes the value of 'settingsSaved', triggering the subscription on the plugin, letting the plugin know to close the dialog.

5.2.1.12 Deferred

Deferred is a custom class used by the system to make plugin library ¡-¿ system library communication easier to accomplish. It is basically a JavaScript Promise with resolve() and reject() functions exposed and a unique ID associated with it. A Promise, for reference, is similar to a callback function. It is an object whose resolve() function is executed after an asyncronous operation completes. Whenever one library needs to send a message to the other library, it sends said message via the postMessage API. However, postMessage is designed to send one-way messages. It is not designed to handle waiting for replies to messages it sends. Using Deferred allows an ID to be sent with postMessage. This means when the other library receives the message, it retrieves whatever data was requested and replies to the library with the same Deferred ID. The original library can then use this ID to find the original Deferred and resolve it with the data returned from the second library. This class is used heavily thoughout the project to handle message replies on both libraries. Trello applies a similar approach using Promises in its own system [37].

5.2.2 'Manage Plugins' Plugin

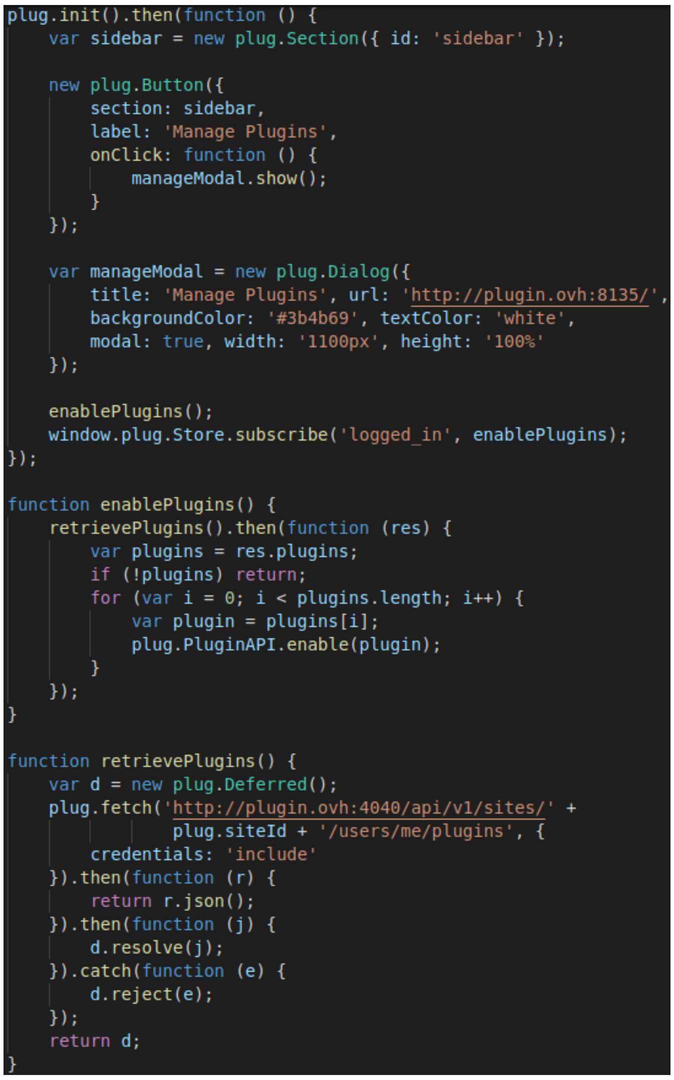
As mentioned previously, this plugin is responsible for automatically loading any user-enabled plugins and giving users the option to enable or disable plugins. Given Pluggy's library architecture was described, now is a good opportunity to demonstrate its usage.

This plugin is broken down into 2 chunks:

- The plugin itself, which loads enabled plugins on page load via the PluginAPI.
- The 'Manage Plugins' iframe, shown within the dialog created by the plugin. This iframe allows enabling and disabling plugins.

5.2.2.1 Plugin (auto-loads enabled plugins)

The code in Figure 5.2 executes the following operations:



```
plug.init().then(function () {
    var sidebar = new plug.Section({ id: 'sidebar' });

    new plug.Button({
        section: sidebar,
        label: 'Manage Plugins',
        onClick: function () {
            manageModal.show();
        }
    });

    var manageModal = new plug.Dialog({
        title: 'Manage Plugins', url: 'http://plugin.ovh:8135/',
        backgroundColor: '#3b4b69', textColor: 'white',
        modal: true, width: '1100px', height: '100%'
    });

    enablePlugins();
    window.plug.Store.subscribe('logged_in', enablePlugins);
});

function enablePlugins() {
    retrievePlugins().then(function (res) {
        var plugins = res.plugins;
        if (!plugins) return;
        for (var i = 0; i < plugins.length; i++) {
            var plugin = plugins[i];
            plug.PluginAPI.enable(plugin);
        }
    });
}

function retrievePlugins() {
    var d = new plug.Deferred();
    plug.fetch('http://plugin.ovh:4040/api/v1/sites/' +
                plug.siteId + '/users/me/plugins', {
        credentials: 'include'
    }).then(function (r) {
        return r.json();
    }).then(function (j) {
        d.resolve(j);
    }).catch(function (e) {
        d.reject(e);
    });
    return d;
}
```

FIGURE 5.2: Plugin code which auto-loads enabled plugins

- Registers a section with ID 'sidebar'. It is assumed the website has a section with ID 'sidebar' also, so the two are linked, meaning anything added to the section within the plugin gets added to the section on the website.

- Creates a button labeled 'Manage Plugins', which when clicked will show the 'Manage Plugins' dialog. Links the button with the 'sidebar' section. This means the button will show in the sidebar on the third-party site.

- Registers a dialog with various properties. The dialog being created is modal with a title 'Manage Plugins'. The dialog's contents are hosted on 'plugin.ovh:8135' in an iframe.

- Executes enablePlugins(). This basically retrieves the list of plugins from the plugin system REST API and enables them one by one via PluginAPI.enable().

- The plugin subscribes to a logged-in event, so if within the management dialog a user logs in, their plugins will be enabled in real time by the plugin.

In short, the code adds a ’Manage Plugins’ button which when clicked shows a ’Manage Plugins’ dialog. The code also auto-loads enabled plugins.

5.2.2.2 Plugin Dialog (allows managing plugins)

This is a VueJS website, containing substantially more code involved than was in the plugin. It offers the following features:

- Users can register a new email and password, or log in if they already have an account. User accounts are stored in the external plugin system REST API.
- Users can upload plugins and delete them.
- Users can view uploaded plugins.
- Users can enable and disable uploaded plugins.
- Users can click on an enabled plugin and, if it has a settings menu, open it.

Logging in / Registering Logging in and registering is done via a modal dialog. When the form is submitted, an AJAX request is sent to the plugin system’s REST API. The requested endpoint depends on the action. If one is logging in, a POST request is sent to /session. Otherwise, if registering, a POST request is sent to /users.

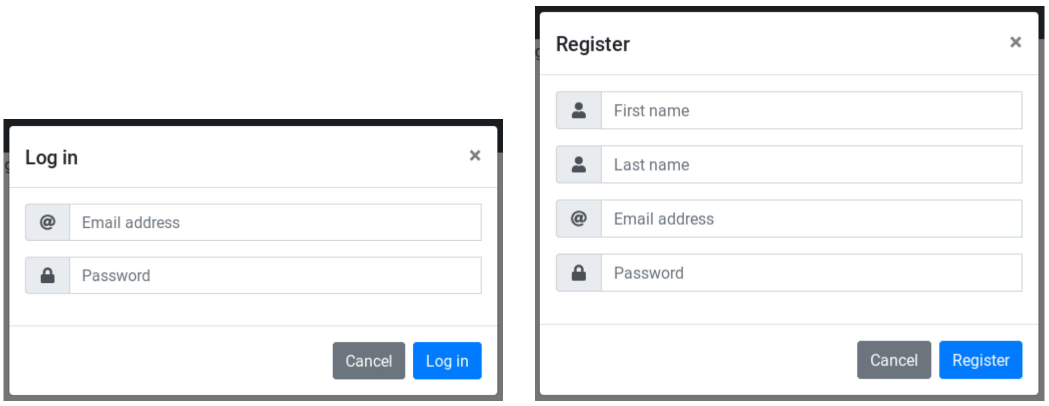


Figure 5.3: Left: Logging in. Right: Registering a new user.

Uploading Plugins One uploads plugins via a modal dialog. Plugins are in the form of an archive containing index.html and any JS files included by index.html. A ZIP file is recommended, although the extraction library used supports other formats. When the form is submitted, an AJAX request is sent, POST /plugins, which creates an entry for the plugin via the API.

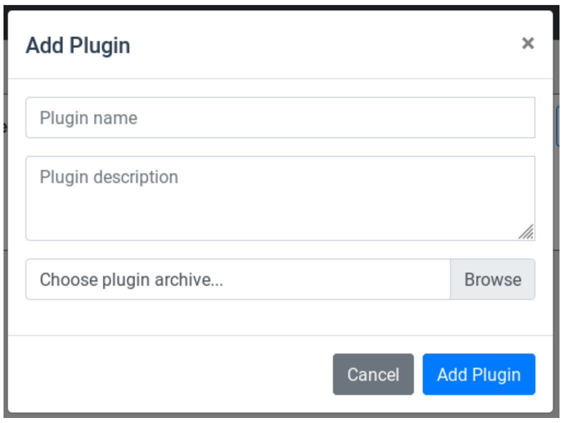


FIGURE 5.4: UI allowing for uploading plugins

Enabling and Disabling Plugins If a plugin is not enabled, in the plugins list one will see an 'Add' button next to it. Clicking this will sent a POST request to /users/me/plugins and also call PluginAPI.enable(plugin) where plugin is whichever plugin was clicked. The AJAX request is needed so as to keep track of enabled plugins. The PluginAPI call is needed to instantly enable the plugin on the parent website without requiring a page refresh.

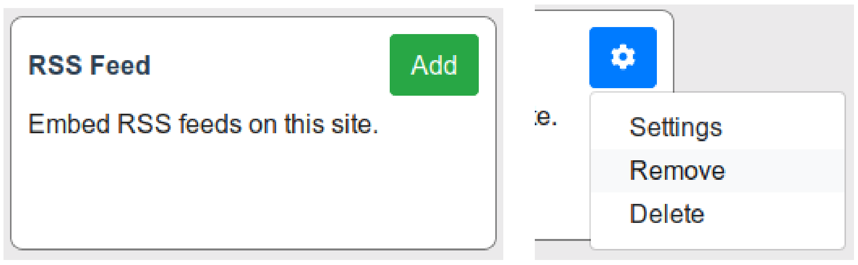


FIGURE 5.5: Left: Option to enable plugin. Right: Option to disable plugin.

Viewing Plugin Settings If a plugin has a settings menu, it will appear in the context menu along with the option to disable a plugin. Plugins create settings dialogs by creating a Dialog with ID 'settings'. Once that is done, the 'Settings' button will appear in this menu.

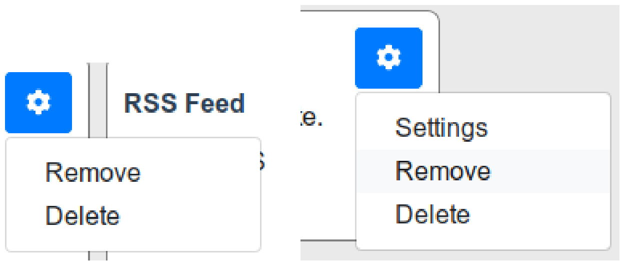


Figure 5.6: Left: Plugin without settings. Right: Plugin with settings.

5.2.3 Plugin System REST API

The REST API structure is almost identical to what was discussed in Chapter 4. Please refer to Figure 4.1 for a recap on the endpoints and their purpose. There was only one change made, which was adding in support for tracking enabled plugins.

The plan originally was to keep track of enabled and disabled plugins by logging them in localStorage. However, this was only suggested as an option as it seemed the fastest means of making the system work. It was known at the time that a REST API solution was more optimal, as a REST API would mean users' enabled plugins would be remembered across devices.

It was stated in section 4.6.1 in Chapter 4 that, if time permitted, additional time would be spent developing API endpoints which keep track of enabled plugins. These endpoints were implemented and are as follows:

Table 5.8: REST API endpoints which keep track of enabled plugins

Method	**Route**	**Params**	**Purpose**
GET	/users/me/plugins	*N/a*	Get a list of enabled plugins
POST	/users/me/plugins	id	Enable the plugin with the specified ID
DELETE	/users/me/plugins/:id	*N/a*	Disable the plugin with the specified ID

Enabled plugins are tracked in the MariaDB database described in Chapter 4.

Implementing the API additions resulted in lost time. However, having enabled plugins be remembered helped speed up testing the system's compatibility with different browser environments. It was worth the time investment.

5.3 Use Cases / Plugins

Uses cases were not discussed in Chapter 4. As this project is a plugin system, the closest equivalent to use cases are plugins. These test that the system works and serve as examples of how it can be used. In Chapter 2, various plugin complexities were described. These were as follows:

5.3.1 Original Use Cases

- **Simple** - Static plugins. The plan for these was for them to be logic-less widgets with no back-end element. An example given was displaying a read-only calendar.
- **Medium** - Plugins which can communicate with an external API. An example was retrieving calendar data from an external API and displaying events on the calendar.
- **Hard** - Plugins which can communicate with an internal API. The idea here was to have it so that plugins could access the website's internal API, but only if they first asked permission to do so. There was no plugin example given here.

5.3.2 Modified Use Cases

The original examples were not used, but an attempt was made to develop plugins which make use of the mentioned functionality (simple, logic-less widgets, ones which communicate with external API, ones which communicate with internal API). The modified use cases are below:

- **Simple** - This plugin is mostly static. It adds a button onto the website. When this button is pressed, an RSS feed is displayed in a dialog. If one goes into settings, the category of the RSS feed being shown can be changed. For instance, one can change it from showing news on UK Politics to Sports instead. The addition of the 'settings' concept makes this plugin more advanced than the read-only calendar would have been.
- **Medium** - This is a plugin which communicates with an external REST API. The core 'Manage Plugins' plugin fulfills this use case. Said plugin communicates with

the plugin system REST API in order to retrieve enabled plugins. The plugin system REST API is an external API.

- **Hard** - These are plugins which communicate with the website's internal REST API. A plugin was made which allows exporting Teamwork Projects or Trello data to CSV. To do so, it communicates with the internal API of the site it is on, whether it is Projects or Trello. The initial goal was to have a permission request prompt show, letting users allow or deny permissions requested by plugins. Unfortunately, a lack of time meant that this was not possible. Instead, website administrators can register permissions and associate them with allowed API URLs. Plugins can request these permissions. However, no dialog is shown. Plugins are instead granted the permission instantly due to time constraints.

5.4 Implementation Schedule

The schedule described in Table 4.2 was altered significantly in the implementation phase. The original schedule had realistic timings for what was described in the plan. However, had one only completed the tasks outlined in said plan, Pluggy would only have been a bare-bones proof of concept with minimal functionality. It would not have had the Store API, meaning plugins and their dialogs would not have been able to communicate with each other. Plugin-created dialogs would not have had access to any plugin system functionality at all for that matter. It would have been missing translation functionality and support for accessing the website's internal API.

In the end, translation functionality was not implemented. Functionality for allowing internal API access via a permission system was partially implemented.

Time was allowed in the original plan in weeks 9-12 for implementing additional functionality. 4 weeks was an insufficient amount of time however, given both missing functionality and documentation needed to be completed in this time. In hindsight, the scope of the project should have been narrowed more during the research phase to allow for all requirements to be fulfilled.

Demonstrated in Table 5.9 is roughly how the workload ended up being handled.

Table 5.9: How workload ended up being handled

Weeks	**Work Completed**
1	Create the plugin system's REST API with fully functional standard login, support for uploading and listing plugins.
2-3	Create a barebones plugin library and system library. This means button functionality and semi-functioning dialogs.
4	Implement the plugin system in Teamwork Projects. Continue expanding on the feature set of the plugin libraries. Refactor the code and its directory structure.
5	Implement the plugin system in Trello. Continue expanding on the feature set of plugin libraries.
6-7	Implement core plugin, also known as the plugin management plugin. Make additions to the plugin system's REST API such that it can keep track of enabled plugins.
8-12	Continue expanding on feature set in plugin system. Fix lots of bugs and race conditions. Fix compatibility issues in some browsers. Make basic plugins to prove some functionality works. Permission system to communicate with website API. Allowing plugin-created elements (i.e. buttons) access contextual information about the section they exist under.
1-12	For weekly supervisor meetings, find time to describe progress being made. Document any changes made. Write up the implementation phase report.

Chapter 6

Testing and Evaluation

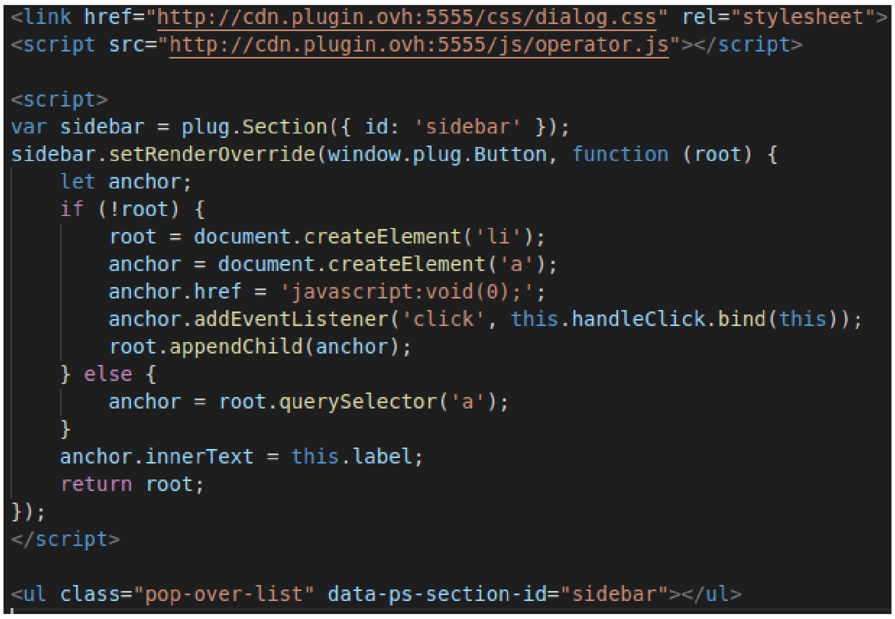
The goal for this project was to make a plugin system. As such, the only way of really testing this system is to develop plugins for it. That a given plugin works is proof that the system itself works. To this end, screenshots of several plugins and their corresponding code will be explored below. Additionally, the aim was for the system to work on multiple websites. Hence, we will explore how to implement the system on both Trello and Teamwork Projects.

6.1 How to Set Up the System

6.1.1 Trello

The first two lines of code in Figure 6.1 is where the dialog's CSS and the system library are imported. The dialog's CSS is intentionally separated from the system library so that the website operator can change dialog styling if they so desire.

After these lines is a script tag. In this, the section with ID 'sidebar' is declared. This is necessary because in single-page applications, elements such as the sidebar may not exist on page-load. It may take some interaction for the element to exist, such as a button press. However, plugins still need to know if the section is a valid section on the website. Hence, its ID is registered here, even if its corresponding element does not yet exist.



```
<link href="http://cdn.plugin.ovh:5555/css/dialog.css" rel="stylesheet">
<script src="http://cdn.plugin.ovh:5555/js/operator.js"></script>

<script>
var sidebar = plug.Section({ id: 'sidebar' });
sidebar.setRenderOverride(window.plug.Button, function (root) {
    let anchor;
    if (!root) {
        root = document.createElement('li');
        anchor = document.createElement('a');
        anchor.href = 'javascript:void(0);';
        anchor.addEventListener('click', this.handleClick.bind(this));
        root.appendChild(anchor);
    } else {
        anchor = root.querySelector('a');
    }
    anchor.innerText = this.label;
    return root;
});
</script>

<ul class="pop-over-list" data-ps-section-id="sidebar"></ul>
```

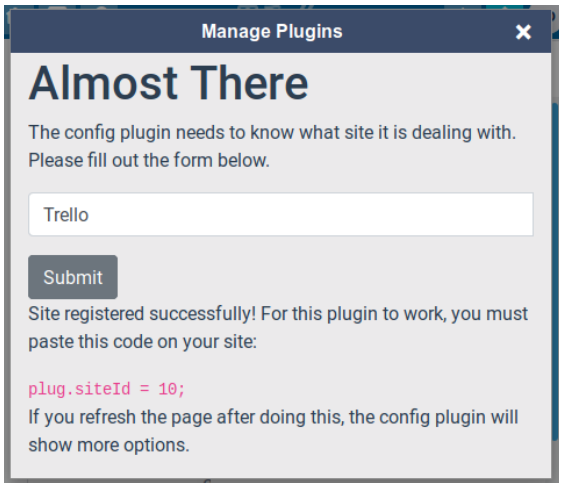
FIGURE 6.1: Code ran on Trello to make Pluggy work

Once the section is registered, a 'setRenderOverride' function is called. This allows overriding how buttons are rendered in said section. It is more useful than simply allowing CSS styling, as one can completely change how button HTML markup is rendered, allowing replicating the styling of the section on the site they are rendered on. This allows matching the styling of other items in the sidebar.

Shown underneath the script tag in Figure 6.1 is an unordered-list HTML element. This is the sidebar's corresponding HTML element, or the parent element of the buttons, if you will. The system library will automatically populate this element with any plugin-created buttons. data-ps-section-id is 'sidebar', matching the already-registered section from earlier. The site operator needs to create this element themselves so Pluggy knows where to render buttons.

The code in Figure 6.1 is enough to make the 'Manage Plugins' button visible in the sidebar and clickable, as shown in Figure **??**. However, if one clicks one this button, the dialog shown does not allow uploading plugins yet. There is one further step which the site operator must complete.

A form is shown in Figure 6.2. In it, one must enter the name of the site that they are setting up, and press the 'Submit' button. Once this is done, a site ID is generated. Below the submitted form, a line of JS code will appear. In this example, 'plug.siiteId = 10;'. The site operator must add this onto their site and plugins will then function as



Manage Plugins

Almost There

The config plugin needs to know what site it is dealing with. Please fill out the form below.

Trello

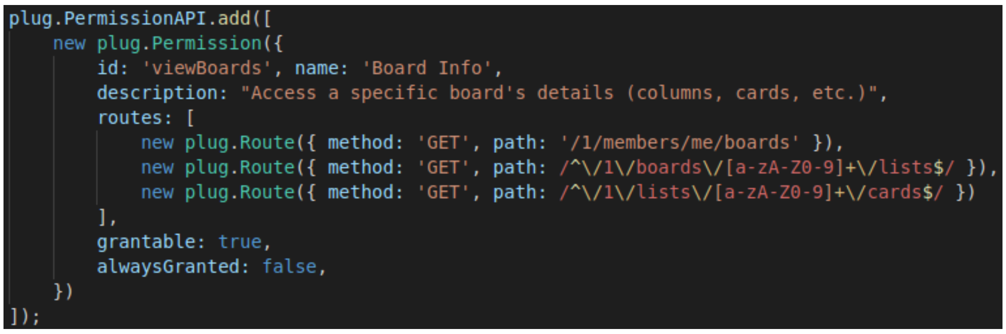
Submit

Site registered successfully! For this plugin to work, you must paste this code on your site:

```
plug.siteId = 10;
```

If you refresh the page after doing this, the config plugin will show more options.

FIGURE 6.2: Form in which one generates a site ID for Trello



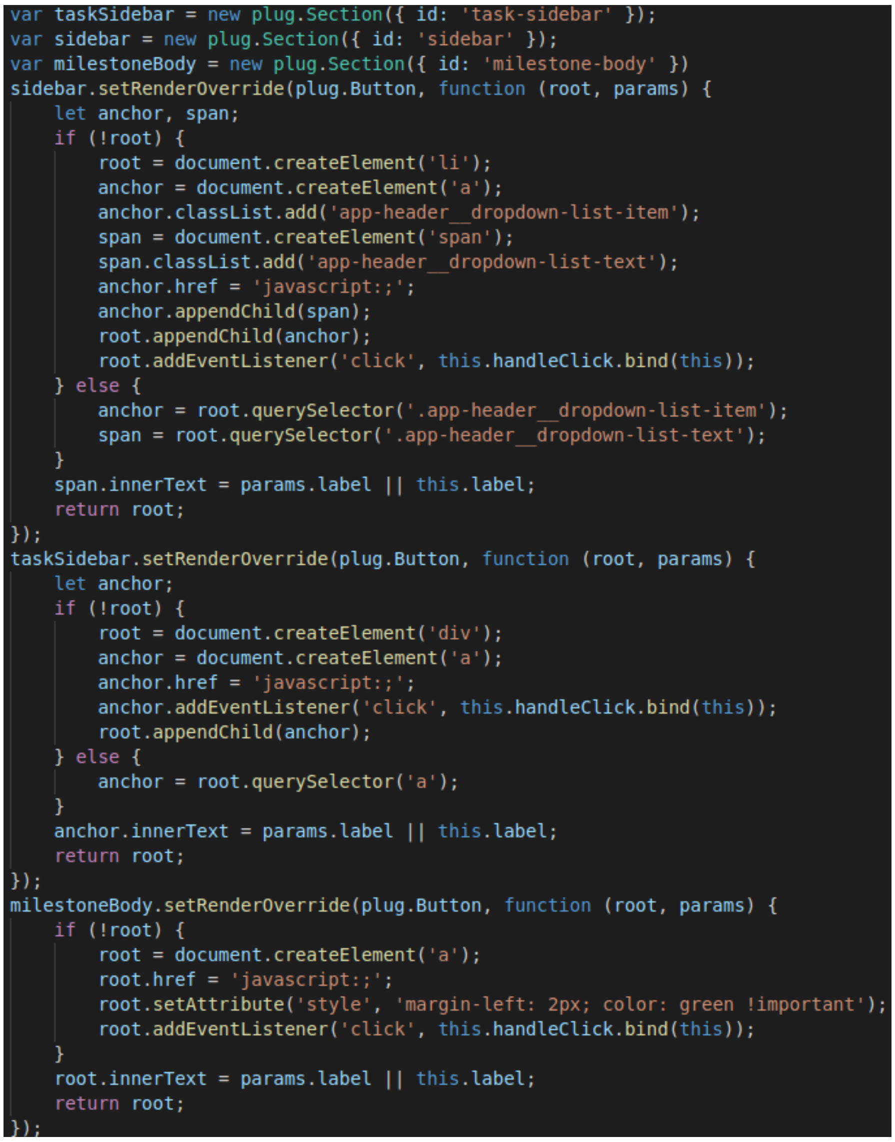
```
plug.PermissionAPI.add([
    new plug.Permission({
        id: 'viewBoards', name: 'Board Info',
        description: "Access a specific board's details (columns, cards, etc.)",
        routes: [
            new plug.Route({ method: 'GET', path: '/1/members/me/boards' }),
            new plug.Route({ method: 'GET', path: /^\/1\/boards\/[a-zA-Z0-9]+\/lists$/ }),
            new plug.Route({ method: 'GET', path: /^\/1\/lists\/[a-zA-Z0-9]+\/cards$/ })
        ],
        grantable: true,
        alwaysGranted: false,
    })
]);
```

FIGURE 6.3: Trello code, registering 'viewBoards' permission

intended. The reasoning behind having this extra strep is that plugins can then check this siteId as a means of detecting what site they are running on. Checking the domain wouldn't be enough as in some cases websites could have multiple siteIds per domain. For example Teamwork Projects and Teamwork Desk exist on the same domain, but one may not want to show the same plugins on both sites, so they would require separate siteIds here.

In addition to the above, there is one more change needed. In order for plugins to access Trello's internal REST API, Trello needs to set up permissions which they can request to use said REST API. These permissions are shown in Figure 6.3. This allows website operators more granular control over what API URLs plugins can access, reducing the risk of allowing plugins to use their internal API.

In reality, the code was not as simple as is outlined here, as this author did not have direct access to Trello's source code and so a hackier approach involving Tampermonkey



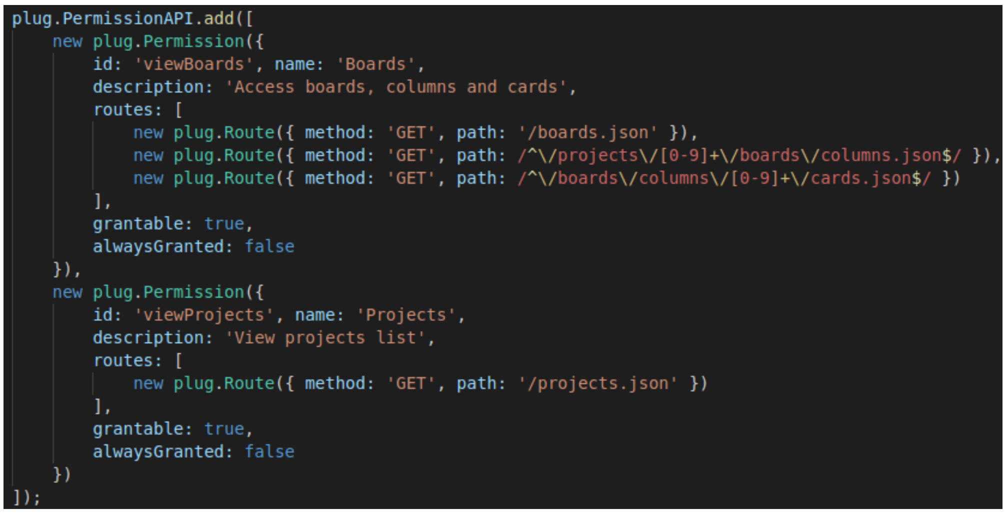
```
var taskSidebar = new plug.Section({ id: 'task-sidebar' });
var sidebar = new plug.Section({ id: 'sidebar' });
var milestoneBody = new plug.Section({ id: 'milestone-body' })
sidebar.setRenderOverride(plug.Button, function (root, params) {
    let anchor, span;
    if (!root) {
        root = document.createElement('li');
        anchor = document.createElement('a');
        anchor.classList.add('app-header__dropdown-list-item');
        span = document.createElement('span');
        span.classList.add('app-header__dropdown-list-text');
        anchor.href = 'javascript:;';
        anchor.appendChild(span);
        root.appendChild(anchor);
        root.addEventListener('click', this.handleClick.bind(this));
    } else {
        anchor = root.querySelector('.app-header__dropdown-list-item');
        span = root.querySelector('.app-header__dropdown-list-text');
    }
    span.innerText = params.label || this.label;
    return root;
});
taskSidebar.setRenderOverride(plug.Button, function (root, params) {
    let anchor;
    if (!root) {
        root = document.createElement('div');
        anchor = document.createElement('a');
        anchor.href = 'javascript:;';
        anchor.addEventListener('click', this.handleClick.bind(this));
        root.appendChild(anchor);
    } else {
        anchor = root.querySelector('a');
    }
    anchor.innerText = params.label || this.label;
    return root;
});
milestoneBody.setRenderOverride(plug.Button, function (root, params) {
    if (!root) {
        root = document.createElement('a');
        root.href = 'javascript:;';
        root.setAttribute('style', 'margin-left: 2px; color: green !important');
        root.addEventListener('click', this.handleClick.bind(this));
    }
    root.innerText = params.label || this.label;
    return root;
});
```

FIGURE 6.4: Code ran on Teamwork Projects to make Pluggy work

was necessary, as shown in Figure A.9 and Figure A.10. Had direct access to the source code been available, it would have been as simple as mentioned above.

6.1.2 Teamwork Projects

The process for getting the system to work on Teamwork Projects is basically the same as was described for Trello, so it will not be explained to the same extent. As shown in Figure 6.4, the site operator registers three sections with IDs 'task-sidebar', 'sidebar' and 'milestone-body' respectively. Each section is told to render its buttons differently to match each section's styling correctly.



```
plug.PermissionAPI.add([
    new plug.Permission({
        id: 'viewBoards', name: 'Boards',
        description: 'Access boards, columns and cards',
        routes: [
            new plug.Route({ method: 'GET', path: '/boards.json' }),
            new plug.Route({ method: 'GET', path: /^\/projects\/[0-9]+\/boards\/columns.json$/ }),
            new plug.Route({ method: 'GET', path: /^\/boards\/columns\/[0-9]+\/cards.json$/ })
        ],
        grantable: true,
        alwaysGranted: false
    }),
    new plug.Permission({
        id: 'viewProjects', name: 'Projects',
        description: 'View projects list',
        routes: [
            new plug.Route({ method: 'GET', path: '/projects.json' })
        ],
        grantable: true,
        alwaysGranted: false
    })
]);
```

FIGURE 6.5: Teamwork Projects code, registering 'viewBoards' and 'viewProjects' permissions

Asides form that, there is only one other major difference which is the permissions being registered. In Teamwork Projects' case, there are two permissions being registered, 'viewBoards' and 'viewProjects'. See Figure 6.5 for the code creating said permissions. Plugins can request these permissions and then access the internal API URLs associated with them. For example, if a plugin attempted to access /boards.json without 'viewBoards' permission, the request would fail. If said plugin was to ask for the same URL having been granted 'viewBoads' permission, it would go through successfully.

6.2 Use Cases / Plugins

6.2.1 Simple - RSS Feed

6.2.1.1 Requirement(s) Being Addressed

- "Plugins should have the ability to add buttons in [specific] areas, and said buttons should take the styling of the website on which they're being displayed."
- "Plugins should be capable of displaying pop-ups containing HTML elements styled however they desire."
- "... it should be possible to customise a dialog's contents even after it is rendered, and changes should show instantly"

- ”A plugin written for one website should require minimal configuration to make it compatible with another website.”

6.2.1.2 What the Plugin Does (Screenshots)

Before showing code, it seems like it would be a good idea to illustrate the plugin’s functionality. The plugin adds a ’Show RSS Feed’ button onto the sidebar of the website it is on.

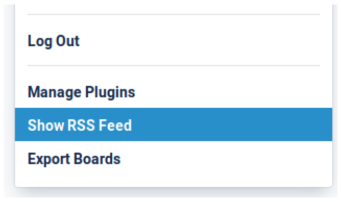
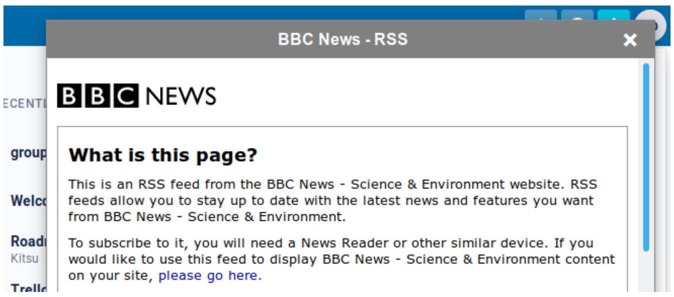


FIGURE 6.6: RSS feed plugin sidebar button

When the ’Show RSS Feed’ button is pressed, an RSS feed dialog is shown. Note how it is displaying displaying the news category ’Science & Environment’.



What is this page?

This is an RSS feed from the BBC News - Science & Environment website. RSS feeds allow you to stay up to date with the latest news and features you want from BBC News - Science & Environment.

To subscribe to it, you will need a News Reader or other similar device. If you would like to use this feed to display BBC News - Science & Environment content on your site, please go here.

FIGURE 6.7: RSS feed plugin dialog

If one closes the RSS feed dialog and instead clicks on ’Manage Plugins’ in the sidebar, they will see a context menu button on the RSS feed plugin in the plugins list. Upon clicking this, one will notice a ’Settings’ button.

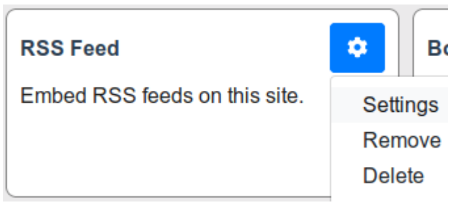


FIGURE 6.8: RSS feed plugin settings button

If one clicks the 'Settings' button previously mentioned, a settings dialog is shown in which one can change the news category displayed in the RSS feed plugin dialog. 'Settings' was not one of the functional requirements, but it should have been. Trello offers the same functionality [38]. It is reasonable to expect that one would be able to edit a plugin's settings.

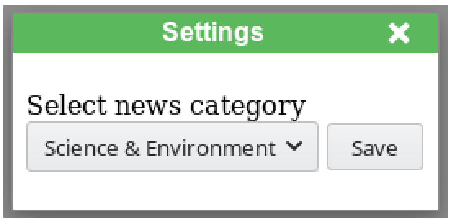


FIGURE 6.9: RSS feed plugin settings dialog

If one was to change the news category to UK and press 'Save', the settings dialog would close, having saved any changes made.

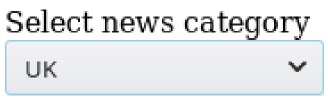
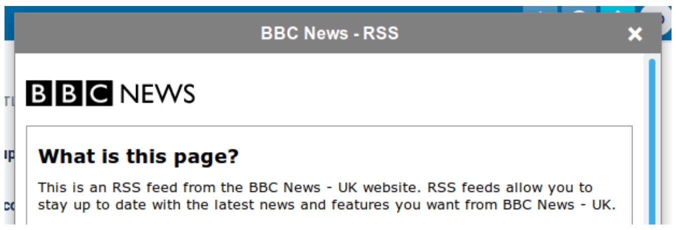


FIGURE 6.10: Changing news category to UK

The RSS feed dialog displays the newly changed news category immediately. Note how in this screenshot it says 'BBC News - UK', indicating that the plugin detected the change in category. For this to work, it required communication between the plugin's settings dialog and the plugin itself so it could update the RSS feed dialog's URL.



What is this page?

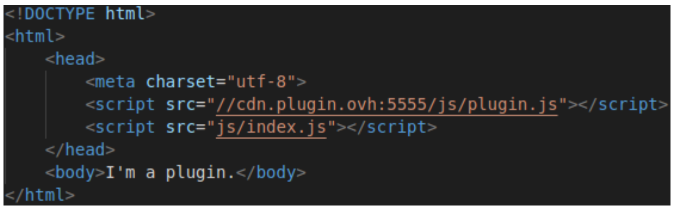
This is an RSS feed from the BBC News - UK website. RSS feeds allow you to stay up to date with the latest news and features you want from BBC News - UK.

FIGURE 6.11: RSS plugin dialog, with news category set to UK

Lastly, all of the above screenshots are of the plugin working on Trello. To demonstrate that the plugin also works on Teamwork Projects, a screenshot is included below. This proves the system and its plugins work on multiple websites.

6.2.1.3 How the Plugin Works (Code)

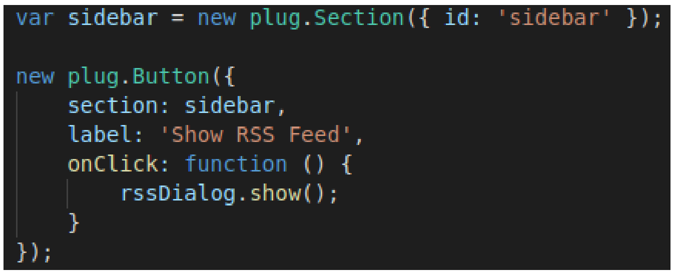
All plugins start off with an index.html file. This plugin is no exception. The purpose of index.html is to import the plugin library (plugin.js) and the main JS file used to initialize the plugin (index.js in this example).



```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <script src="//cdn.plugin.ovh:5555/js/plugin.js"></script>
        <script src="js/index.js"></script>
    </head>
    <body>I'm a plugin.</body>
</html>
```

FIGURE 6.12: RSS Feed plugin code, index.html

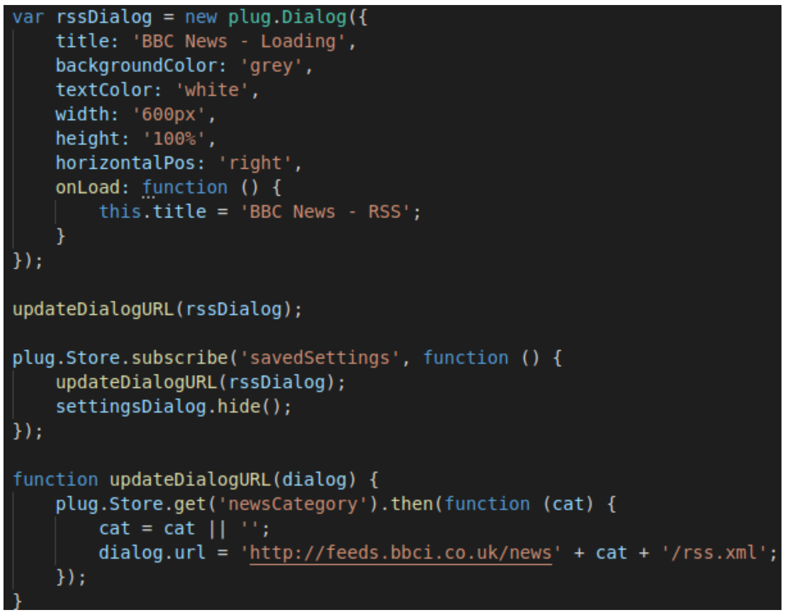
In order to create the button in the sidebar, the plugin first registers the section with ID 'sidebar'. It can then create the button and link it to the sidebar. The section must already have been registered on the site itself, as mentioned in Section 6.1 earlier. The code in Figure 6.13 simply creates a button labeled 'Show RSS Feed' in the sidebar. Once this button is cicked, the RSS feed dialog (whose creation is described later) is shown.



```javascript
var sidebar = new plug.Section({ id: 'sidebar' });

new plug.Button({
    section: sidebar,
    label: 'Show RSS Feed',
    onClick: function () {
        rssDialog.show();
    }
});
```

FIGURE 6.13: RSS Feed plugin code, creating a button

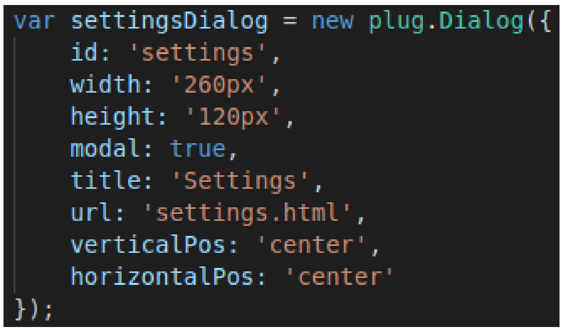
Shown in Figure 6.14 is the code necessary to show the RSS feed dialog. What happens is, it creates the dialog with text and various other properties. Prior to the dialog loading, its title shows 'Loading', but once the iframe within the dialog has loaded for the first time, the dialog changes to show 'RSS'. The URL of the iframe is set to the current value of the news category via the updateDialogURL() function. The plugin can detect when its settings dialog is saved via the Store.subscribe() function. In the event of settings changing, the settings dialog is hidden and the dialog URL is updated with the new category value.



```
var rssDialog = new plug.Dialog({
    title: 'BBC News - Loading',
    backgroundColor: 'grey',
    textColor: 'white',
    width: '600px',
    height: '100%',
    horizontalPos: 'right',
    onLoad: function () {
        this.title = 'BBC News - RSS';
    }
});

updateDialogURL(rssDialog);

plug.Store.subscribe('savedSettings', function () {
    updateDialogURL(rssDialog);
    settingsDialog.hide();
});

function updateDialogURL(dialog) {
    plug.Store.get('newsCategory').then(function (cat) {
        cat = cat || '';
        dialog.url = 'http://feeds.bbci.co.uk/news' + cat + '/rss.xml';
    });
}
```

FIGURE 6.14: RSS Feed plugin code, creating the RSS dialog

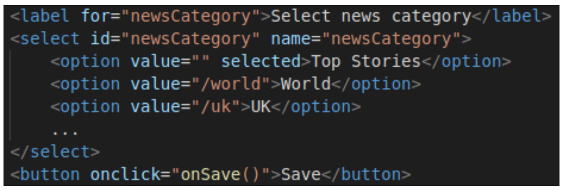
Registering the settings dialog is easy. It is basically just a case of creating a normal dialog and setting its ID to 'settings'. The 'Manage Plugins' plugin will find this dialog and display it when the plugin's 'Settings' menu option is pressed. Worth highlighting is that the 'settings.html' URL is being displayed. This is the next file we will describe.



```
var settingsDialog = new plug.Dialog({
    id: 'settings',
    width: '260px',
    height: '120px',
    modal: true,
    title: 'Settings',
    url: 'settings.html',
    verticalPos: 'center',
    horizontalPos: 'center'
});
```

FIGURE 6.15: RSS Feed plugin code, creating the RSS settings dialog

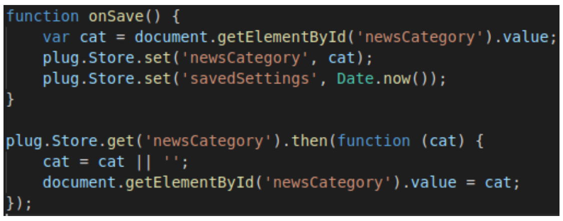
Settings.html imports plugin.js, the plugin library, in the same way that index.html does. This is so it can have access to the Store API. Below is a select dropdown where one can choose a news category. Also shown is the 'Save' button which performs onSave() when pressed.



```
<label for="newsCategory">Select news category</label>
<select id="newsCategory" name="newsCategory">
    <option value="" selected>Top Stories</option>
    <option value="/world">World</option>
    <option value="/uk">UK</option>
    ...
</select>
<button onclick="onSave()">Save</button>
```

FIGURE 6.16: RSS Feed plugin code, settings.html (settings dialog)

When the settings dialog is opened, the selected news category is updated to match the category previously saved. This is done via Store.get(). When the 'Save' button is pressed, plug.Store.set() is called, saving the news category and the 'savedSettings' value. As mentioned before, the RSS feed plugin itself is subscribed to the 'savedSettings' key, so when its value changes, the plugin is notified and knows that it should close the dialog as the dialog cannot close itself. The URL of the category in the RSS dialog updates after saving also, as previously mentioned.



```
function onSave() {
    var cat = document.getElementById('newsCategory').value;
    plug.Store.set('newsCategory', cat);
    plug.Store.set('savedSettings', Date.now());
}

plug.Store.get('newsCategory').then(function (cat) {
    cat = cat || '';
    document.getElementById('newsCategory').value = cat;
});
```

FIGURE 6.17: RSS Feed plugin code, settings.js (settings dialog JS)

6.2.2 Medium - 'Manage Plugins' Plugin

This plugin was already thoroughly explained in Section 5.2.2 due to it being a core part of the plugin system's architecture. This plugin serves as an example of a plugin that can interact with an external API. There would be little value in outlining the plugin a second time, so instead only the requirements it satisfies are listed below.

6.2.2.1 Requirement(s) Being Addressed

- "Plugins should have the ability to add buttons in [specific] areas, and said buttons should take the styling of the website on which they're being displayed."
- "Plugins should be capable of displaying pop-ups containing HTML elements styled however they desire."
- "Plugins should have the ability to request external API endpoints."

- ”Plugins should be able to programmatically determine the site on which they are hosted.”
- ”A plugin written for one website should require minimal configuration to make it compatible with another website.”

6.2.3 Hard - Export to CSV

6.2.3.1 Requirement(s) Being Addressed

- ”Plugins should have the ability to add buttons in [specific] areas, and said buttons should take the styling of the website on which they’re being displayed.”
- ”Plugins should be capable of displaying pop-ups containing HTML elements styled however they desire.”
- ”Plugins should be capable of modifying/retrieving data from the API of the website on which they are being displayed.”
- ”Plugins should be able to programmatically determine the site on which they are hosted.”
- ”A plugin written for one website should require minimal configuration to make it compatible with another website.”
- ”Plugins should not be able to make unauthorized API requests.”

6.2.3.2 What the Plugin Does (Screenshots)

This plugin exports board data to CSV. ’Boards’ is a feature that Trello and Teamwork Projects have in common. The concept of this plugin is that it compiles board data from either site into CSV format.

We see below that the plugin adds an ’Export Boards’ button to the sidebar.

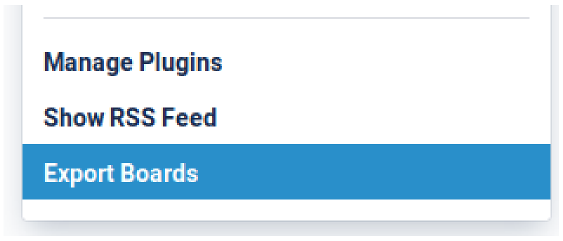


FIGURE 6.18: Export Boards plugin sidebar button

Once the 'Export Boards' button is clicked, a dialog is shown. The dialog's content differs depending on which site the plugin is running on. On Trello, there is a list of boards. Teamwork Projects differs in that there is a list of projects, each with a single board. Therefore, one must specify a board to export in Trello's dialog, whereas one must specify a project in Projects' dialog.

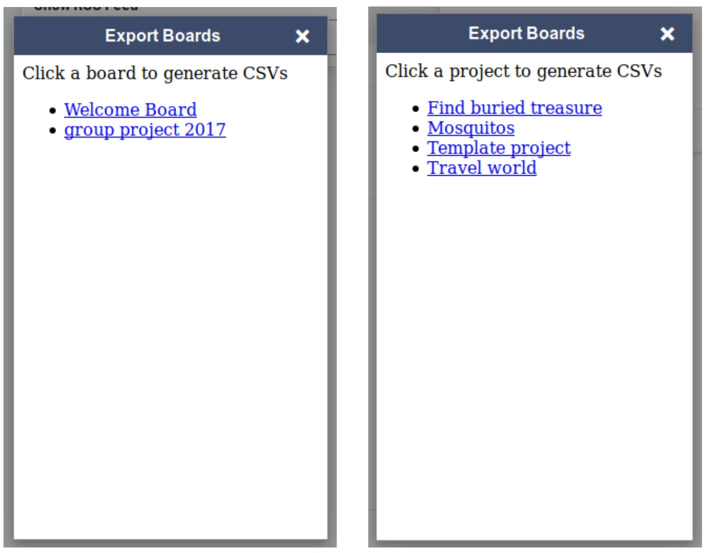


FIGURE 6.19: Left: Choosing a Trello board. Right: Choosing a Projects project.

After specifying either the board or project depending on which site it is, the user is then given the option of either exporting all column metadata for said board or exporting card data for a given column. The name of the board or project whose data is being exported is shown at the top of the modal. In Trello's case, it is a board named "group project 2017". If one clicks on the 'Columns' URL then all columns in said board/project are exported as CSV. If one clicks on an individual column URL than all cards in that particular column are exported.

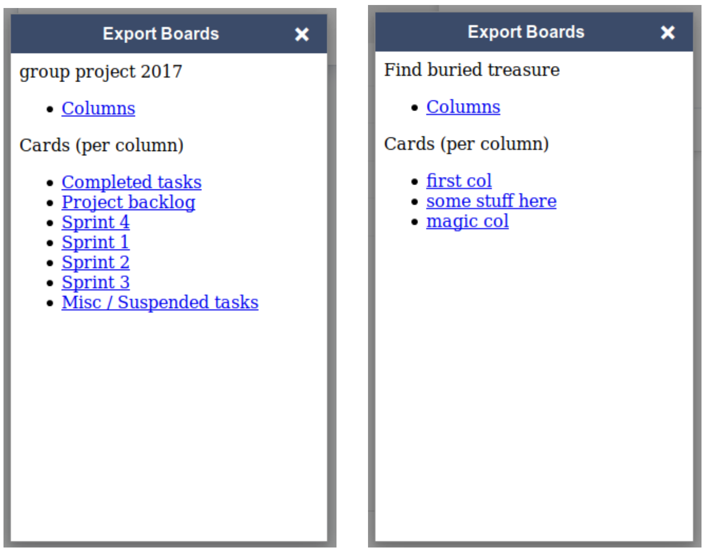
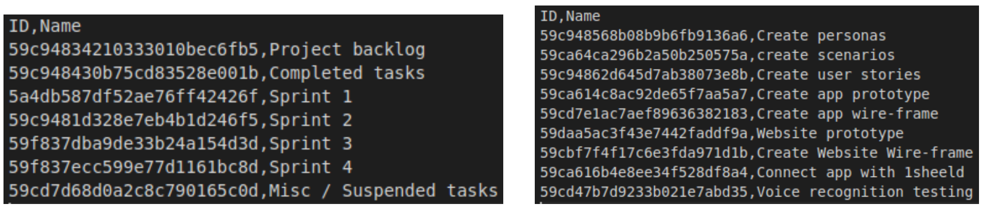


FIGURE 6.20: Left: Exporting a Trello board. Right: Export a Projects project.

Shown on the left in Figure 6.21 is the result of choosing to export all columns as CSV. On the right is the result of exporting all cards of a given column to CSV.



```
ID,Name
59c94834210333010bec6fb5,Project backlog
59c948430b75cd83528e001b,Completed tasks
5a4db587df52ae76ff42426f,Sprint 1
59c9481d328e7eb4b1d246f5,Sprint 2
59f837dba9de33b24a154d3d,Sprint 3
59f837ecc599e77d1161bc8d,Sprint 4
59cd7d68d0a2c8c790165c0d,Misc / Suspended tasks
```

```
ID,Name
59c948568b08b9b6fb9136a6,Create personas
59ca64ca296b2a50b250575a,create scenarios
59c94862d645d7ab38073e8b,Create user stories
59ca614c8ac92de65f7aa5a7,Create app prototype
59cd7e1ac7aef89636382183,Create app wire-frame
59daa5ac3f43e7442faddf9a,Website prototype
59cbf7f4f17c6e3fda971d1b,Create Website Wire-frame
59ca616b4e8ee34f528df8a4,Connect app with 1sheeld
59cd47b7d9233b021e7abd35,Voice recognition testing
```

FIGURE 6.21: Left: Exporting columns. Right: Export cards.

As a bonus, Pluggy supports handling contextual information in Teamwork Projects. This means an 'Export This Board' button can be added to a board's context menu, so a project need not necessarily be specified. When this button is clicked, the plugin is passed the button's contextual information. This means the plugin retrieves data associated with the parent of the button. In this case, it is passed information about the board's project. The export dialog appears and displays the cards and columns for said project. The dialog is the same as shown in Figure 6.20.

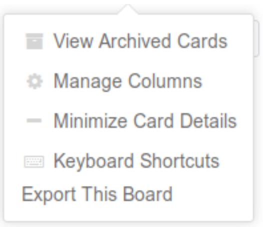
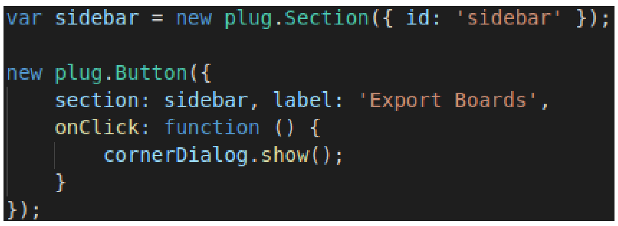


FIGURE 6.22: 'Export This Board' button, which relies on contextual information

6.2.4 How the Plugin Works (Code)

This plugin, the RSS feed plugin and all other plugins, starts off with an index.html file. See Figure 6.12 for an example of this.

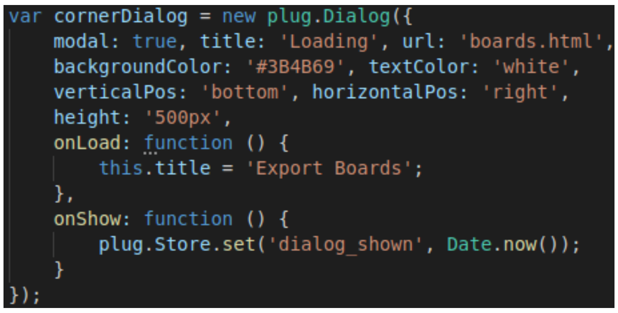
The 'Export Boards' button is created in a similar manner to how 'Show RSS Feed' was created earlier. The plugin registers the sidebar section. A button is then created and its section is set to the sidebar. As with in the RSS feed plugin, when this button is clicked, it displays a dialog, in this case a dialog named cornerDialog.



```
var sidebar = new plug.Section({ id: 'sidebar' });

new plug.Button({
    section: sidebar, label: 'Export Boards',
    onClick: function () {
        cornerDialog.show();
    }
});
```

FIGURE 6.23: Export Boards plugin code, 'Export Boards' sidebar button

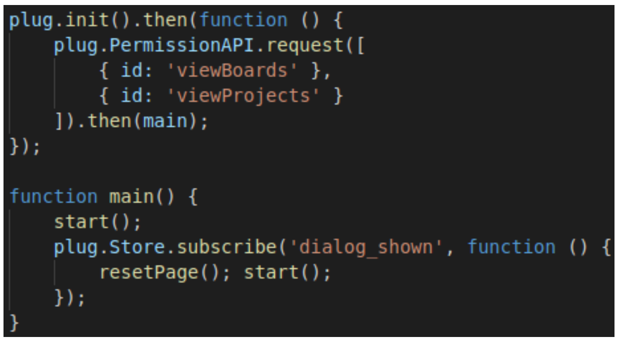
The code for the dialog itself is shown in Figure 6.24. A modal dialog is created. Until its iframe with URL boards.html loads, its title is 'Loading'. When the page loads, the dialog's title is changed to 'Export Boards'. When the dialog is shown, the vaue at 'dialog_shown' is updated. This is necessary as the dialog's iframe only loads the first time it is opened. If one closes the dialog and reopens it, it continues to show the old contents. Adding this Store.set() call allows the dialog to detect when it has been reopened so it knows to update or reset its contents.



```
var cornerDialog = new plug.Dialog({
    modal: true, title: 'Loading', url: 'boards.html',
    backgroundColor: '#3B4B69', textColor: 'white',
    verticalPos: 'bottom', horizontalPos: 'right',
    height: '500px',
    onLoad: function () {
        this.title = 'Export Boards';
    },
    onShow: function () {
        plug.Store.set('dialog_shown', Date.now());
    }
});
```

Figure 6.24: Export Boards plugin code, 'Export Boards' dialog

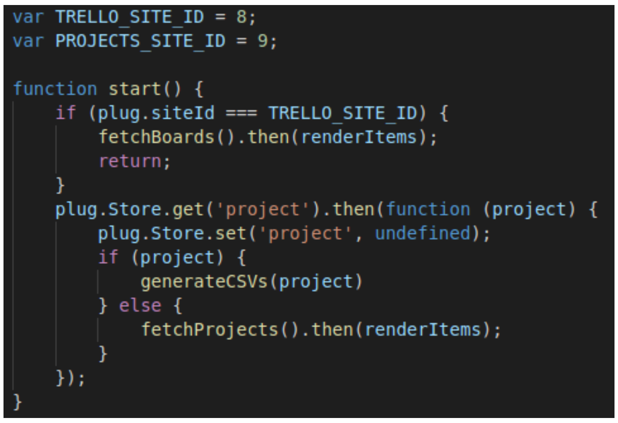
In order for the plugin to retrieve data for boards, projects, cards and columns, it needs to access the internal REST API of the sites it is enabled on. Shown in Figure 6.25 is the plugin requesting 'viewBoards' and 'viewProjects' permission. Once the permission has been granted, the code inside main() is executed. What the code does is, if this is the first time opening the dialog, it executes start(). If not, dialog_shown must have triggered, in which case resetPage() is called to reset the page, and start() is then called to populate it again.



```
plug.init().then(function () {
    plug.PermissionAPI.request([
        { id: 'viewBoards' },
        { id: 'viewProjects' }
    ]).then(main);
});

function main() {
    start();
    plug.Store.subscribe('dialog_shown', function () {
        resetPage(); start();
    });
}
```

Figure 6.25: Export Boards plugin code, requesting necessary API permissions

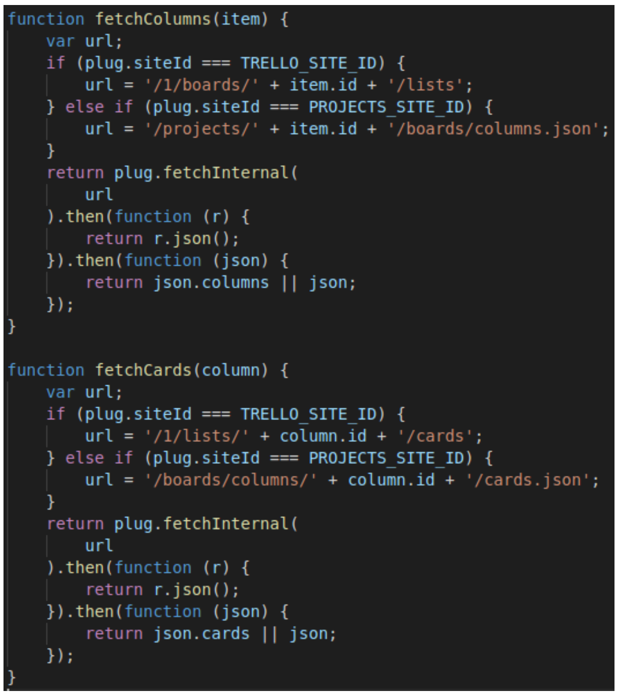
Once the dialog has been granted the requested permissions, it then checks what website it is on. If it is on Trello, it calls a fetchBoards() function which retrieves the list of boards from the API. It then calls renderItems(boards), which iterates over each board and displays it in a list on the page. If it is on Projects, however, the rules get more complex. If the dialog is opened via the button in the sidebar, then Store.get('project') is undefined. The project is only ever defined if the button in the board context menu is pressed. Hence, if the button in the sidebar is pressed, fetchProjects() is called to get the list of projects so the user can choose one. Otherwise, if the context menu button is pressed, the project is known due to the project being set in the Store earlier when the dialog was opened, in which case CSVs are generated for that specific project and the user is not shown a list of projects.



```
var TRELLO_SITE_ID = 8;
var PROJECTS_SITE_ID = 9;

function start() {
    if (plug.siteId === TRELLO_SITE_ID) {
        fetchBoards().then(renderItems);
        return;
    }
    plug.Store.get('project').then(function (project) {
        plug.Store.set('project', undefined);
        if (project) {
            generateCSVs(project)
        } else {
            fetchProjects().then(renderItems);
        }
    });
}
```

FIGURE 6.26: Export Boards plugin code, calling different functions depending on site

The remainder of the plugin code is not hugely different. What is worth showing however is the internal API requests being made. The fetchInternal() function gets called by the plugin library, which is acting as a proxy for the system library's fetchInternal() implementation. This approach allows the function to look at the requested URLs and see if they are in the URLs granted by permissions, and if not, the internal requests are rejected. The use of fetchInternal() is shown in Figure 6.27.



```
function fetchColumns(item) {
    var url;
    if (plug.siteId === TRELLO_SITE_ID) {
        url = '/1/boards/' + item.id + '/lists';
    } else if (plug.siteId === PROJECTS_SITE_ID) {
        url = '/projects/' + item.id + '/boards/columns.json';
    }
    return plug.fetchInternal(
        url
    ).then(function (r) {
        return r.json();
    }).then(function (json) {
        return json.columns || json;
    });
}

function fetchCards(column) {
    var url;
    if (plug.siteId === TRELLO_SITE_ID) {
        url = '/1/lists/' + column.id + '/cards';
    } else if (plug.siteId === PROJECTS_SITE_ID) {
        url = '/boards/columns/' + column.id + '/cards.json';
    }
    return plug.fetchInternal(
        url
    ).then(function (r) {
        return r.json();
    }).then(function (json) {
        return json.cards || json;
    });
}
```

FIGURE 6.27: Export Boards plugin code, accessing different API URLs depending on site

6.3 Results

Shown in Table 6.1 are the requirements which were or were not fulfilled after the implementation of this Project. Much of these requirements were shown to be fulfilled throughout the setup explanation in this chapter and throughout the plugin demonstrations.

Table 6.1: Which requirements were and were not fulfilled

Requirement	**Type**	**Completed Successfully**
Website can define areas in which they want plugins to display UI elements.	Functional	Yes. Shown in Chapter 6.
Plugins can able to add buttons. Buttons take the styling of the areas they're on.	Functional	Yes. Shown in Chapter 6.
Plugins can display pop-up lists, where clicking on a list item performs action.	Functional	**No**
Plugins can display pop-up dialogs contaning HTML styled however they desire.	Functional	Yes. Shown in Chapter 6.
Plugins can make requests to external API endpoints.	Functional	Yes. Shown in Chapter 5.
Plugins can make requests to API endpoints of the site they're enabled on.	Functional	Yes. Shown in Chapter 6.
Plugins can determine what site they're on.	Functional	Yes. Shown in Chapter 6.
Website operators can restrict what endpoints plugin can access. Plugins can request to access an endpoint via a prompt.	Functional	**Partially**. Shown in Chapter 6. Operators can make permissions which must be granted to access endpoints. However, if a plugin requests a permission, it is automatically granted. There is no prompt.
Plugins can modify the properties of any previously rendered UI elements.	Functional	Yes. Shown in Chapter 6.
Plugins written for one site can easily work on another site.	Non-Functional	Yes. Shown in Chapter 6.
Website operators can implement the system on their site with minimal changes.	Non-Functional	Yes. Shown in Chapter 6.
Plugin code can only ever execute in a sandboxed environment.	Non-Functional	Yes. Explained in Chapter 4.
Plugins have minimal load times. Load times could be reduced by minimising library file sizes.	Non-Functional	Yes. Shown in Chapter 5, difficulties.
Plugins have an API with which they can support translations.	Non-Functional	**No**

Chapter 7

Discussion and Conclusions

7.1 Solution Review

The original goal for the project was to make a plugin system capable of working on several websites. It was also desired for the system to be secure, such that plugins could not introduce XSS vulnerabilities onto the websites on which they were enabled. Throughout Chapter 6, a number of plugins were shown running on both Trello and Teamwork Projects. For each plugin, it was explained which requirements each plugin needed fulfilled in order to function. These plugins functioning is evidence of the system working as intended and that the majority of requirements were fulfilled. On that basis, this author considers the developed solution a success, as it achieved its overall goal. There were some improvements which could have been made, which will be discussed in the Future Work section, but as a whole it was a success.

7.2 Project Review

Throughout this project's research phase, a considerable amount of time was spent investigating how one would go about developing this system and performing prototyping to ensure ideas were feasible. Originally OAuth was going to be used by the plugin system but it was discovered through attempts at prototyping that almost every site implements OAuth differently meaning making a system able to handle all sites is not feasible given the project's time constraints. Additionally, there was much time spent determining what options there were for sandboxing plugins and deciding which sandboxing approach would best suit the project's design. Investigating the architecture behind and the functionality offered by existing solutions such as Trello Power-Ups and Zendesk Apps proved useful when developing Pluggy. This was as it highlighted necessary functionality and requirements and showed how one might develop a plugin architecture.

However, due to so much time being spent on researching implementation, an insufficient amount of time was spent planning out how the workload would be handled in advance of the implementation phase. The requirements outlined during the research phase were not achievable within the time frame specified, which was originally 8 weeks.

Because there was such a development workload, this resulted in a significant portion of the documentation being held off until the end. The original plan was to allow 4 weeks for this. The code was only finished in week 12. In spite of these issues during the planning and implementation phase however, this author deems the project a success. The goal was to research how one would make a portable, secure plugin system. This system was successfully implemented. Developing sample plugins was not originally planned and time was not allocated for it, but nonetheless some were successfully developed despite time constraints.

In hindsight, if this author was to work on the project again from scratch, more time would have been spent planning out the workload and reducing the number of requirements so as to ensure all of the project's requirements were met. The work was carried out as efficiently as it could have been. The issue was simply that there was physically not enough time to get everything done. Allowing user-created code to run on multiple websites in a secure manner proves difficult to accomplish, but it is possible, as demonstrated by Pluggy functioning by the end of the project.

7.3 Future Work

- Create a dedicated website for the plugin system, on which all aspects of the system would be documented. This would help developers learn how to implement the system.
- Allow plugin developers sell access to their plugins as a one-time fee or on a subscription basis.

- Permit plugin developers lock down plugins or plugin features to higher plans such as Enterprise-only users.
- Allow third-party sites limit the number of plugins which can be inserted on their site on a per-plan basis.
- Attempt to build a bridge which will convert Trello Power-Up code into code which Pluggy (the plugin system) understands. The goal would be to allow one to make use of Trello's existing plugins on other similar sites.
- Come up with a mechanism for updating plugins. WordPress permits hosting plugins on a subversion (SVN) system which allows auto-updating in an easier manner. Perhaps a similar approach could be adopted using Git?
- Add an option to the plugin management dialog, whereby users can post their requested features and other users can vote on said features so developers can prioritise developing the more frequently requested plugin ideas.
- Significantly improve the system's performance on sites running VueJS and KnockoutJS by developing a custom VueJS directive or KnockoutJS binding which one can apply to a section's HTML element. Directives and bindings, in simple terms, allow code to be executed when an element is attached or detached to them, in this case section elements. Elements are attached to directives or bindings when said elements come into existence. Leveraging these technologies would allow detecting when a section has been added to the DOM. Currently Pluggy's code is using MutationObserver, which is not performant at all due to it requiring constantly looking at every modified element and said element's children on a page to see if a section needs to be rendered. Using directives or bindings would allow detecting when a section has been added without needing to traverse the DOM tree. The caveat is that this optimisation would only benefit sites running the VueJS or KnockoutJS frameworks.

Bibliography

[1] Power-ups — trello. [Online]. Available: https://trello.com/power-ups

[2] D. Ribeiro, “Sandboxing javascript,” August 2016. [Online]. Available: https://medium.com/zendesk-engineering/sandboxing-javascript-e4def55e855e

[3] Apps script — google developers. [Online]. Available: https://developers.google.com/apps-script/

[4] Asana for slack — asana. [Online]. Available: https://asana.com/slack

[5] A. Sahney, “Do more from your inbox with gmail add-ons,” October 2017. [Online]. Available: https://www.blog.google/products/g-suite/do-more-your-inbox-gmail-add-ons/

[6] R. Miller, “Salesforce is buying mulesoft at enterprise value of $6.5 billion,” March 2018. [Online]. Available: https://techcrunch.com/2018/03/20/salesforce-is-buying-mulesoft-at-enterprise-value-of-6-5-billion/

[7] Cross-site scripting (xss) — owasp. [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[8] Appexchange is the salesforce store. [Online]. Available: https://appexchange.salesforce.com/

[9] J. Sinai, “The world’s leading business app marketplace turns 10,” January 2016. [Online]. Available: https://www.salesforce.com/blog/2016/01/appexchange-10-birthday.html

[10] B. Cervino, “Announcing power-ups for all!” August 2016. [Online]. Available: https://blog.trello.com/trello-power-ups-for-all

[11] ——, “Introducing the new trello power-ups directory,” January 2018. [Online]. Available: https://blog.trello.com/trello-power-ups-directory

[12] B. Cook, “Building a trello power-up,” May 2017. [Online]. Available: https://tech.trello.com/power-up-tutorial-part-one/

[13] J. Constine, “Slack launches app directory and joins top vcs for $80m fund backing developers,” December 2015. [Online]. Available: https://techcrunch.com/2015/12/15/trophy-emoji/

[14] Building slack apps. [Online]. Available: https://api.slack.com/slack-apps#slack_app_management_tour

[15] Teamwork projects features - teamwork.com. [Online]. Available: https://www.teamwork.com/projects-features

[16] Integrations - teamwork projects support. [Online]. Available: https://support.teamwork.com/projects/addons-and-integrations

[17] Wordpress plugins — wordpress.org. [Online]. Available: https://wordpress.org/plugins/

[18] J. de Valk, “The anatomy of a wordpress theme,” January 2011. [Online]. Available: https://yoast.com/wordpress-theme-anatomy/

[19] N. Tomovic, “More surprising statistics about wordpress usage,” April 2016. [Online]. Available: https://managewp.com/blog/statistics-about-wordpress-usage

[20] (2018, December) Wordpress vulnerability statistics. [Online]. Available: https://wpvulndb.com/statistics

[21] (2018, November) Email client market share - november 2018. [Online]. Available: https://emailclientmarketshare.com/

[22] S. API, “Introducing actions: A simple shortcut attached to every slack message,” May 2018. [Online]. Available: https://medium.com/slack-developer-blog/introducing-actions-a-simple-shortcut-attached-to-every-slack-message-e2404414ece

[23] Booklet by vince. [Online]. Available: https://trello.com/power-ups/598b2de2adedc253537fdfcb/booklet-by-vince

[24] Using the github power-up. [Online]. Available: https://help.trello.com/article/1065-using-the-github-power-up

[25] Same-origin policy — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[26] M. Bentkowski, "Xss-es in google caja," July 2016. [Online]. Available: https://blog.bentkowski.info/2016/07/xss-es-in-google-caja.html

[27] ——, "Yet another google caja bypasses hat-trick," November 2017. [Online]. Available: https://blog.bentkowski.info/2017/11/yet-another-google-caja-bypasses-hat.html

[28] Css: Cascading style sheets — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS

[29] R. Damlencour, "Securing an iframe thanks to the sandbox attribute," July 2015. [Online]. Available: https://blog.dareboost.com/en/2015/07/securing-iframe-sandbox-attribute/

[30] Window.postmessage() — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

[31] Mutationobserver — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver

[32] Localstorage — mdn. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

[33] Getting started with gorm — gorm guide. [Online]. Available: http://doc.gorm.io/

[34] T. Noe, "Applying javascript: User scripts," April 2016. [Online]. Available: https://medium.freecodecamp.org/applying-javascript-user-scripts-2e505643644d

[35] Choosing a browser to use with teamwork projects - teamwork projects support. [Online]. Available: https://support.teamwork.com/projects/using-teamwork-projects/choosing-a-browser-to-use-with-teamwork-projects

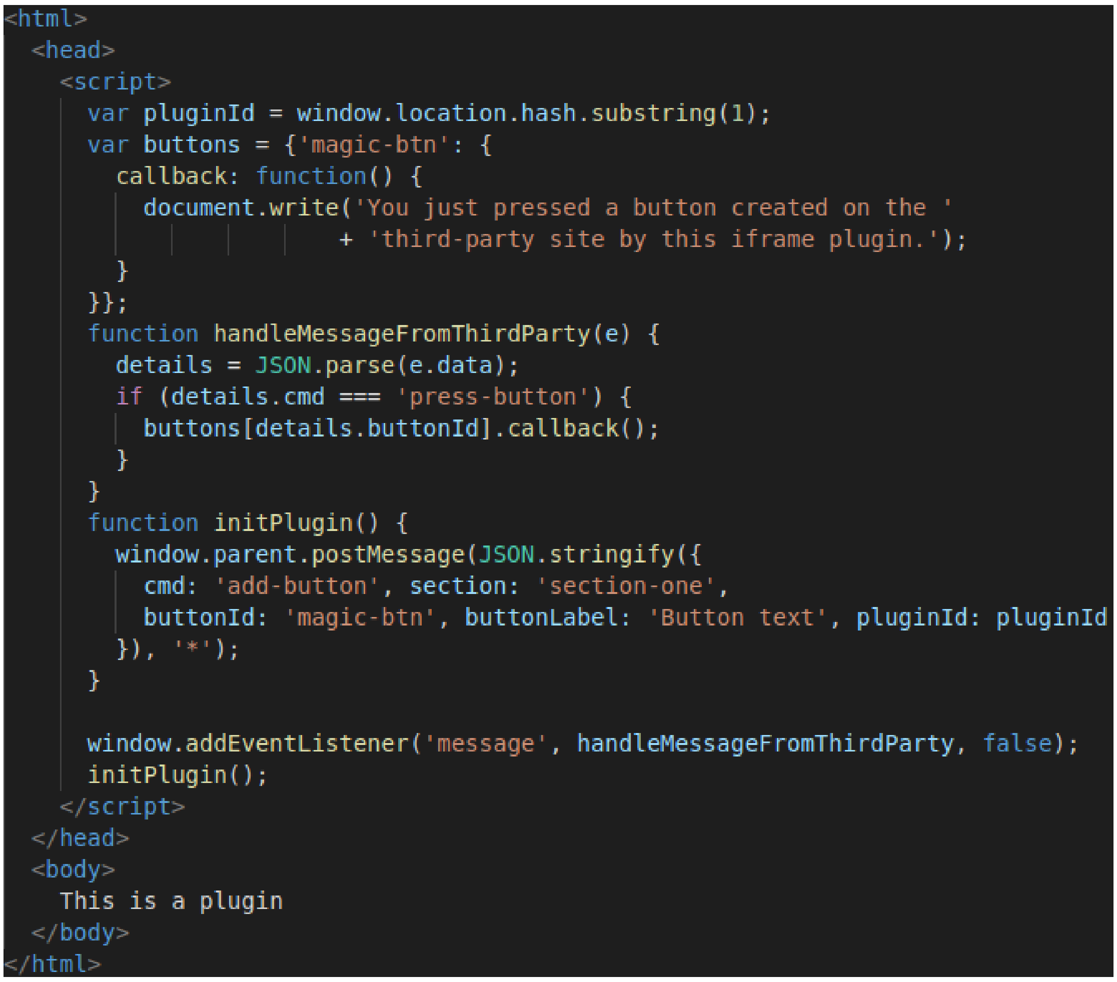
[36] Extended ie11 support. [Online]. Available: https://trello.com/en/support/extended-ie11

[37] Bluebird promises. [Online]. Available: https://developers.trello.com/reference/#bluebird-promises

[38] show-settings. [Online]. Available: https://developers.trello.com/reference/#show-settings

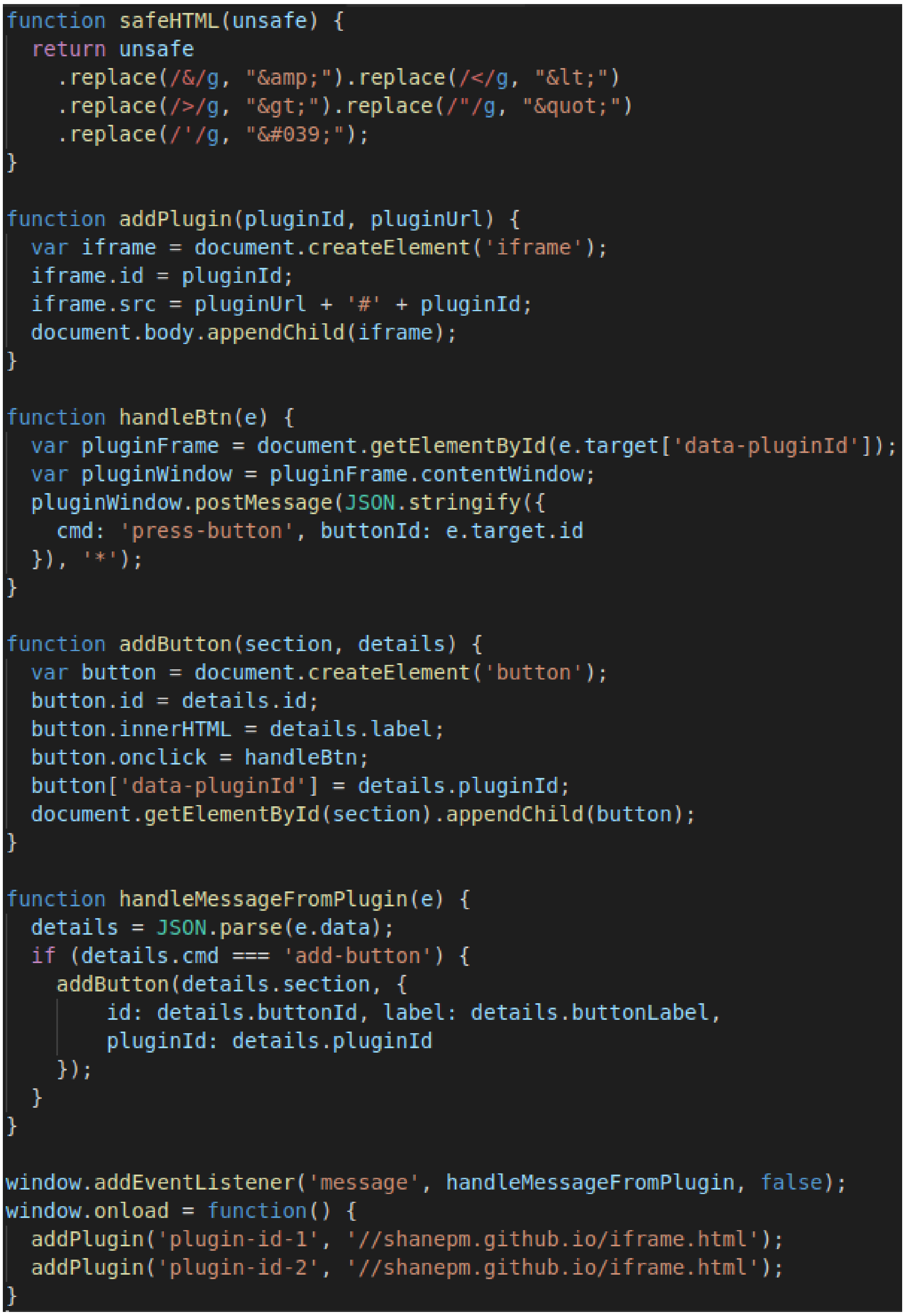
Appendix A

Code Snippets



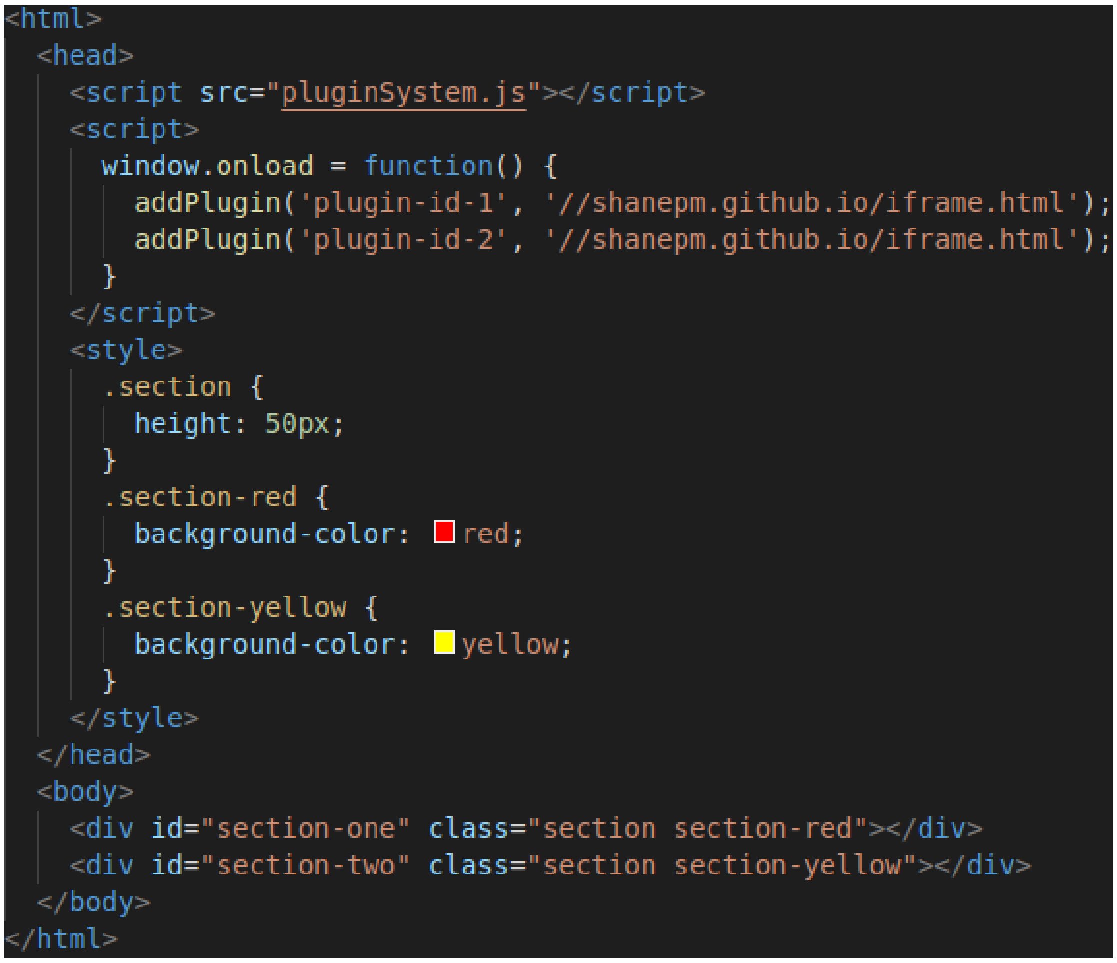
```
<html>
  <head>
    <script>
      var pluginId = window.location.hash.substring(1);
      var buttons = {'magic-btn': {
        callback: function() {
          document.write('You just pressed a button created on the '
                                 + 'third-party site by this iframe plugin.');
        }
      }};
      function handleMessageFromThirdParty(e) {
        details = JSON.parse(e.data);
        if (details.cmd === 'press-button') {
          buttons[details.buttonId].callback();
        }
      }
      function initPlugin() {
        window.parent.postMessage(JSON.stringify({
          cmd: 'add-button', section: 'section-one',
          buttonId: 'magic-btn', buttonLabel: 'Button text', pluginId: pluginId
        }), '*');
      }

      window.addEventListener('message', handleMessageFromThirdParty, false);
      initPlugin();
    </script>
  </head>
  <body>
    This is a plugin
  </body>
</html>
```

Figure A.1: PoC plugin code, contained in an iframe



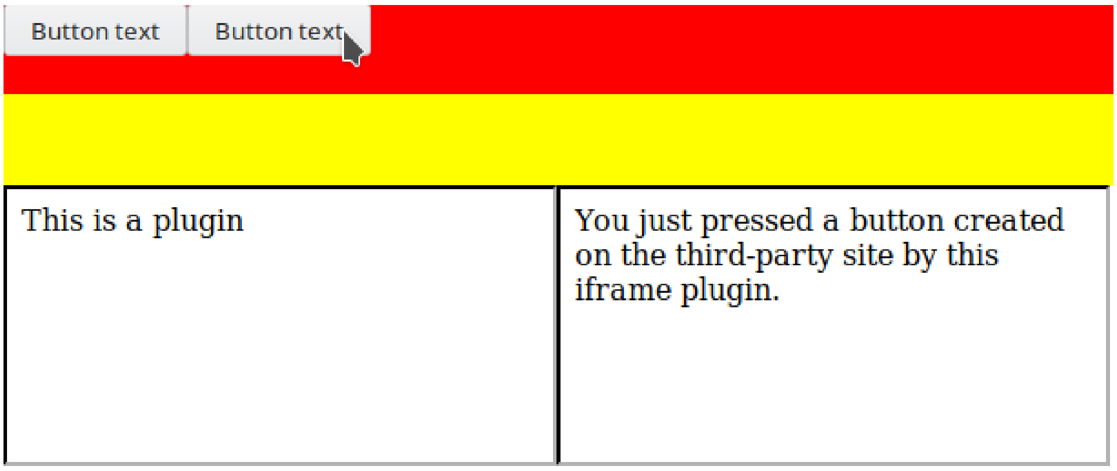
```
function safeHTML(unsafe) {
  return unsafe
    .replace(/&/g, "&amp;").replace(/</g, "&lt;")
    .replace(/>/g, "&gt;").replace(/"/g, "&quot;")
    .replace(/'/g, "&#039;");
}

function addPlugin(pluginId, pluginUrl) {
  var iframe = document.createElement('iframe');
  iframe.id = pluginId;
  iframe.src = pluginUrl + '#' + pluginId;
  document.body.appendChild(iframe);
}

function handleBtn(e) {
  var pluginFrame = document.getElementById(e.target['data-pluginId']);
  var pluginWindow = pluginFrame.contentWindow;
  pluginWindow.postMessage(JSON.stringify({
    cmd: 'press-button', buttonId: e.target.id
  }), '*');
}

function addButton(section, details) {
  var button = document.createElement('button');
  button.id = details.id;
  button.innerHTML = details.label;
  button.onclick = handleBtn;
  button['data-pluginId'] = details.pluginId;
  document.getElementById(section).appendChild(button);
}

function handleMessageFromPlugin(e) {
  details = JSON.parse(e.data);
  if (details.cmd === 'add-button') {
    addButton(details.section, {
        id: details.buttonId, label: details.buttonLabel,
        pluginId: details.pluginId
    });
  }
}

window.addEventListener('message', handleMessageFromPlugin, false);
window.onload = function() {
  addPlugin('plugin-id-1', '//shanepm.github.io/iframe.html');
  addPlugin('plugin-id-2', '//shanepm.github.io/iframe.html');
}
```

FIGURE A.2: PoC plugin system, communicates with plugins



```
<html>
  <head>
    <script src="pluginSystem.js"></script>
    <script>
      window.onload = function() {
        addPlugin('plugin-id-1', '//shanepm.github.io/iframe.html');
        addPlugin('plugin-id-2', '//shanepm.github.io/iframe.html');
      }
    </script>
    <style>
      .section {
        height: 50px;
      }
      .section-red {
        background-color: 🟥 red;
      }
      .section-yellow {
        background-color: 🟨 yellow;
      }
    </style>
  </head>
  <body>
    <div id="section-one" class="section section-red"></div>
    <div id="section-two" class="section section-yellow"></div>
  </body>
</html>
```

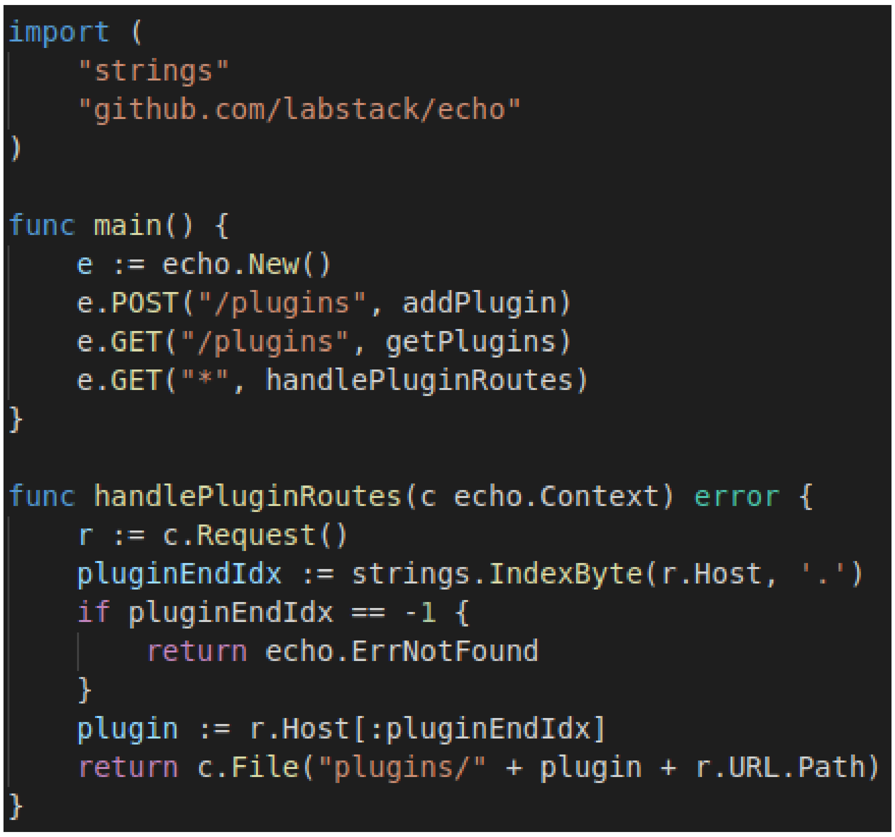
FIGURE A.3: PoC third-party site, uses plugin system



This is a plugin

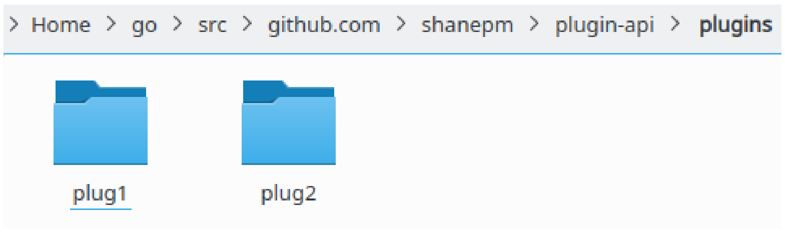
You just pressed a button created on the third-party site by this iframe plugin.

FIGURE A.4: PoC site working screenshot, one button has been clicked



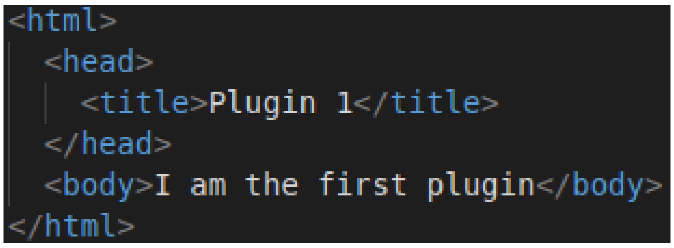
```go
import (
    "strings"
    "github.com/labstack/echo"
)

func main() {
    e := echo.New()
    e.POST("/plugins", addPlugin)
    e.GET("/plugins", getPlugins)
    e.GET("*", handlePluginRoutes)
}

func handlePluginRoutes(c echo.Context) error {
    r := c.Request()
    pluginEndIdx := strings.IndexByte(r.Host, '.')
    if pluginEndIdx == -1 {
        return echo.ErrNotFound
    }
    plugin := r.Host[:pluginEndIdx]
    return c.File("plugins/" + plugin + r.URL.Path)
}
```

FIGURE A.5: Golang code which hosts plugins on separate subdomains



> Home > go > src > github.com > shanepm > plugin-api > plugins
plug1
plug2

FIGURE A.6: Each plugin has a separate directory which corresponds to subdomain



```html
<html>
  <head>
    <title>Plugin 1</title>
  </head>
  <body>I am the first plugin</body>
</html>
```

FIGURE A.7: Example index.html for plugin with subdomain plug1

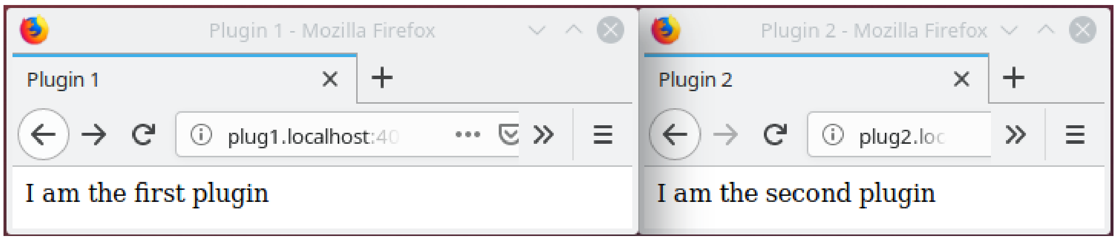
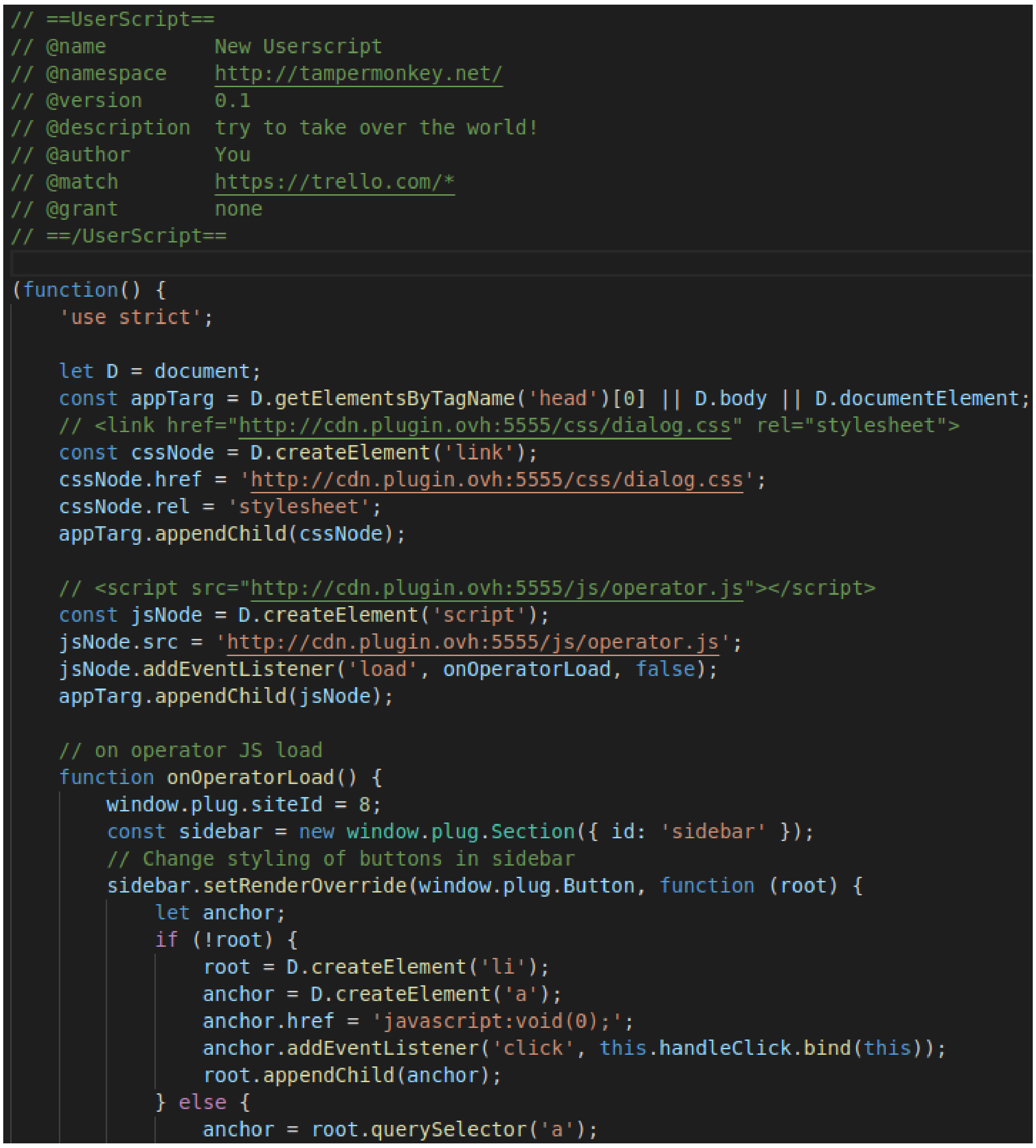
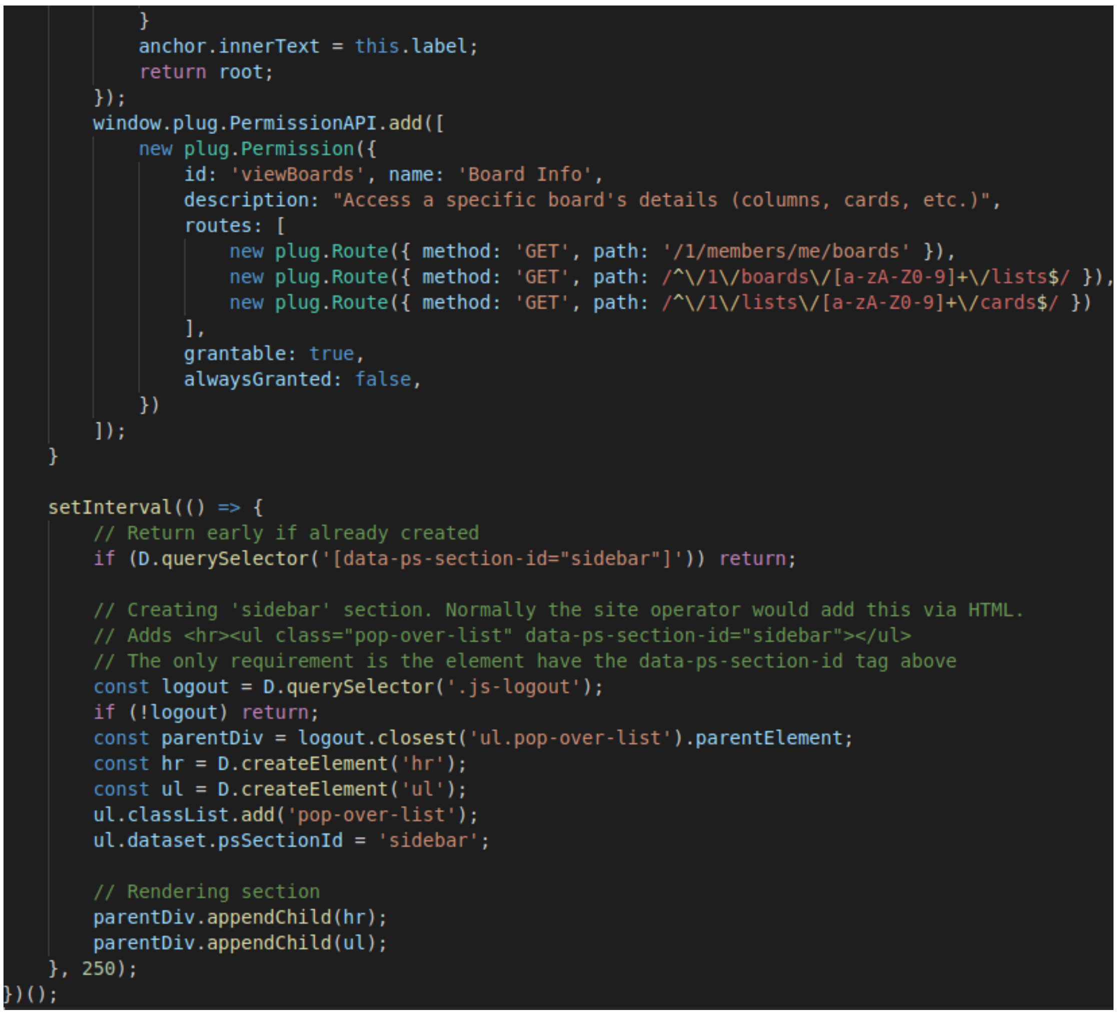


FIGURE A.8: Demonstrating the two plugins running on separate subdomains



```
// ==UserScript==
// @name         New Userscript
// @namespace    http://tampermonkey.net/
// @version      0.1
// @description  try to take over the world!
// @author       You
// @match        https://trello.com/*
// @grant        none
// ==/UserScript==

(function() {
    'use strict';

    let D = document;
    const appTarg = D.getElementsByTagName('head')[0] || D.body || D.documentElement;
    // <link href="http://cdn.plugin.ovh:5555/css/dialog.css" rel="stylesheet">
    const cssNode = D.createElement('link');
    cssNode.href = 'http://cdn.plugin.ovh:5555/css/dialog.css';
    cssNode.rel = 'stylesheet';
    appTarg.appendChild(cssNode);

    // <script src="http://cdn.plugin.ovh:5555/js/operator.js"></script>
    const jsNode = D.createElement('script');
    jsNode.src = 'http://cdn.plugin.ovh:5555/js/operator.js';
    jsNode.addEventListener('load', onOperatorLoad, false);
    appTarg.appendChild(jsNode);

    // on operator JS load
    function onOperatorLoad() {
        window.plug.siteId = 8;
        const sidebar = new window.plug.Section({ id: 'sidebar' });
        // Change styling of buttons in sidebar
        sidebar.setRenderOverride(window.plug.Button, function (root) {
            let anchor;
            if (!root) {
                root = D.createElement('li');
                anchor = D.createElement('a');
                anchor.href = 'javascript:void(0);';
                anchor.addEventListener('click', this.handleClick.bind(this));
                root.appendChild(anchor);
            } else {
                anchor = root.querySelector('a');
```

Figure A.9: First half of the Trello Tampeymonkey Pluggy-enabling code



```
            }
            anchor.innerText = this.label;
            return root;
        });
        window.plug.PermissionAPI.add([
            new plug.Permission({
                id: 'viewBoards', name: 'Board Info',
                description: "Access a specific board's details (columns, cards, etc.)",
                routes: [
                    new plug.Route({ method: 'GET', path: '/1/members/me/boards' }),
                    new plug.Route({ method: 'GET', path: /^\/1\/boards\/[a-zA-Z0-9]+\/lists$/ }),
                    new plug.Route({ method: 'GET', path: /^\/1\/lists\/[a-zA-Z0-9]+\/cards$/ })
                ],
                grantable: true,
                alwaysGranted: false,
            })
        ]);
    }

    setInterval(() => {
        // Return early if already created
        if (D.querySelector('[data-ps-section-id="sidebar"]')) return;

        // Creating 'sidebar' section. Normally the site operator would add this via HTML.
        // Adds <hr><ul class="pop-over-list" data-ps-section-id="sidebar"></ul>
        // The only requirement is the element have the data-ps-section-id tag above
        const logout = D.querySelector('.js-logout');
        if (!logout) return;
        const parentDiv = logout.closest('ul.pop-over-list').parentElement;
        const hr = D.createElement('hr');
        const ul = D.createElement('ul');
        ul.classList.add('pop-over-list');
        ul.dataset.psSectionId = 'sidebar';

        // Rendering section
        parentDiv.appendChild(hr);
        parentDiv.appendChild(ul);
    }, 250);
})();
```

Figure A.10: Second half of the Trello Tampermonkey Pluggy-enabling code