CSCD01 Deliverable 1

Team 18: Sin Chi Chiu, Jason Fong, Jeremy Lai, Mark Padilla, Gavin Zhang

Table of Contents

Design Architecture

Overview

(As of commit e2b30e9e9c16d45477718e9c0608a189f5839b81 (Feb 18) on master branch)

The entirety of the PDF.js project follows the principles of the **Model View Controller (MVC)** design, which is a relatively common design for web based applications. The MVC design calls for an explicit separation between the three primary components of the system, namely the Model, the View, and the Controller.

The **Model** is the portion of the application under the **core** folder. This layer handles the data structures and the logic and functions being applied to the data by the application. In our case this refers to how the application processes any given PDF file.

The **View** is the portion of the application under the **Web and Viewer** folders. This provides the actual display of the application, which in our case is how the application displays the given PDF document back to the user.

The **Controller** is the go-between for the Model and the View and exists in the **display** folder. It receives instructions or data from either the Model or View and sends them to the other in a way the other can then process. It is sort of like an API.

Components

Configuration Files

The files in the root directory mostly relate to the project configuration itself. There are the typical git files like .gitignore.

There are also a great deal of non-critical files relating to certain quality of life related.
For example, files with **gitpod** in them relate to gitpod, which is a coding environment set up for developers via the web (it is not free, so we will not be using this IDE). There are also files relating to a tool called “Eslint”. These are not critical to the project but are instead files for a tool for improving the quality and readability of Javascript code. There is also **.travis.yml** which is a file for a continuous integration tool.

gulpfile.js is a file specifying the build using gulp.js, which is a tool to help speed up builds. This allows the entire project to be built with the singular command “gulp generic”.

package.json and package-lock.json are dependency files used by the project. package.json is a list of every node package used by PDF.js as well as their versions. package-lock.json ensures that all dependencies installed by NPM are of the correct version. This helps contributors keep their versions of the packages in sync.

It is worth noting the LICENSE file which lists the legal rights of those who would use this software. There are also some other files that specify the authors of pdf.js and a code of conduct for contributors.

External Dependencies

External Dependencies are packages of features that PDF.js uses that are not inherent to the core functionality. There are two different external dependencies that PDF.js employs: ones indicated in the package.json that are NPM modules, and ones in the external folder that do not have an installable package from npm.

External

bcmaps is a folder containing binary character maps (font mapping files) that map character codes to character glyphs in CID (Character Identifier) fonts. These are used in PostScript and most Adobe products like reading pdfs, and mostly serve to help in reading documents with East Asian writing systems. This is a legacy system in order to deal with older pdfs created that use this technology.

builder is a folder of files related to building and preprocessing of script/text files that have no built-in preprocessor like JavaScript files. It supports a subset of commands of Mozilla's preprocessor, and supports commands in HTML comments.

cmapscompress like the name suggests, has scripts to compress and package information from the the **cmaps** or **bcmaps** folder, to optimize size and to parse back information, in big-endian order.

dist is a pre-built version of the source code

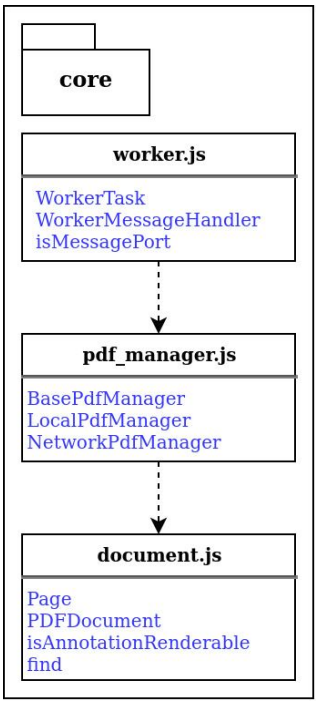
importL10n contains a script that imports language files based on what locale you have

systemjs contains a script from Mozilla Foundation, which enables a connection to the babelcache database, and transpiles that data into something useable

webL10n contains a script from Mozilla Foundation, to prepare the client side l10n and i8n features

webpack contains a from Mozilla Foundation, to load the scripts and content preprocessed by **preprocessor2.js** in the **builder** folder

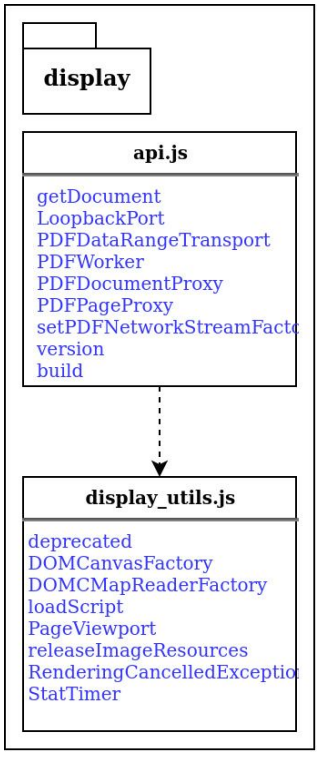
Core



The core layer is where the raw binary of a PDF file is parsed and interpreted. It acts as the model in the MVC design pattern, processing data for the display layer to eventually be used by the viewer. There are three files in core which are most important to this process, in the **core** folder: worker.js, pdf_manager.js, and document.js.

worker.js sets up the pdfManager object (defined in /core/pdf_manager/js) which as part of its implementation, creates and uses the PDFDocument class (defined in /core/document.js) which directly retrieves PDF data and metadata from the PDF itself (itself using numerous dependencies in /core). With the pdfManager initialized, worker.js uses it to initialize the MessageHandler (defined in **/shared**/message_handler.js), which can stream chunks of the PDF to the display layer.

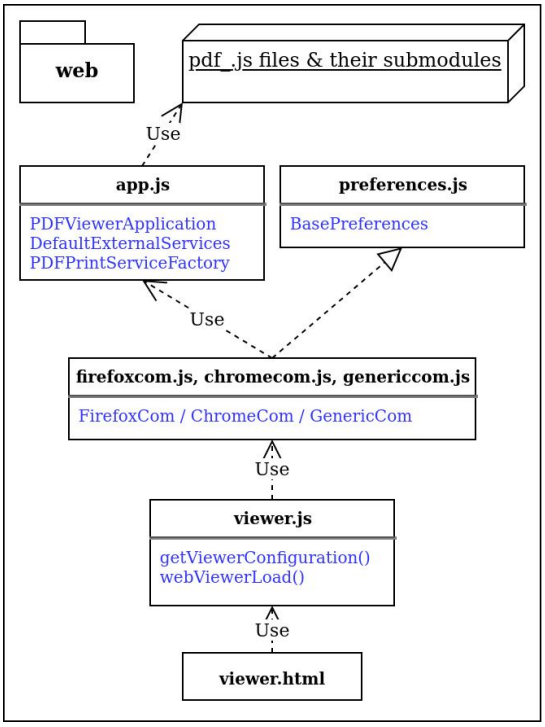
Display



The display layer is the API that exposes access to the core layer, and acts as the controller in the MVC design pattern where PDF data is passed through between the core and viewer. It is responsible for taking in the PDF data and processing it through the core, and then returning the rendered data and other information to the viewer.

The key files in the display layer are api.js and display_utils.js. **api.js** is where most of the other components in the display layer are integrated, and it handles the **rendering** of the PDF via **Canvas** and **WebGL**.
The **display_utils.js** is where the **formatting** happens. It converts processed PDF binary from core to a format that can be used in Javascript or for rendering.

Viewer



The Viewer layer serves as the 'View' component in pdf.js' MVC structure. It is responsible for taking processed PDF data from the Display layer and displaying it on a web page, alongside a UI that has controls for PDF viewing. As such, this is the layer that third-parties interact with when embedding pdf.js directly into their web pages.

app.js serves as the control centre for this layer. It is responsible for initializing other key components in this layer and gathering them into higher-order objects that are exposed for usage other modules. One of these higher-order objects is **PDFViewerApplication**, which holds sub-objects for core components such as the rendering queue and the UI's toolbar.

preferences.js is another important file. It stores and retrieves persistent settings for the viewer widget using the browser's local storage. Since the exact implementation of local storage operations changes depending on the browser, this is one of the few files in the Viewer layer that makes use of an abstract base class, which in this case is called **BasePreferences**.

This abstract base class is extended inside of **firefoxcom.js** and **chromecom.js**. These two files serve as the communication points between the View layer and Firefox and Chrome respectively, as they handle operations specific to each browser. firefoxcom.js and chromecom.js both make extensive use of PDFViewerApplication and BasePreferences, and as such they are tightly coupled to the components defined in app.js and preferences.js.

A generic version of the 'com' files also exist, named **genericcom.js**, which is used for browsers other than Firefox and Chrome.

viewer.js is the file that decides which 'com' to use and then directly interacts with the HTML document. It defines a function called **getViewerConfiguration** that returns a JSON object with references to all of the viewer UI's DOM elements.

These DOM elements are defined inside of **viewer.html**, which loads viewer.js as a script. It also loads other HTML files in this layer, such as viewer-snippet-firefox-extension.html, depending on the browser.

Most other files in this layer, such as the ones prefixed with "pdf_", are dependencies of app.js and define specific components of the viewer's UI, such as the search bar. As such,

these files have a tight coupling with app.js. Based on this, one improvment would be to organize these files into their own subdirectory named “app_components” or something similar, as the “pdf_” prefix does not make this association clear to first-time viewers of the repo.

Test

The **test** directory contains all files used for testing pdf.js code.
pdf.js uses three types of tests that are meant to be used in tandem.

The first two are load tests and reference tests.
Load tests attempt to load PDF files and check if the application crashes or not.
Reference tests run the full application, take screenshots, and compare them to existing screenshots to see if there are any discrepancies. Reference tests are further broken down into three layers based on what is being screenshotted: eq (full application), text overlay, and annotations overlay.

In the test directory, **test.js** manages load tests and reference tests. It makes use of other files in the test directory in order to execute each test.

The third type of tests are **unit tests**. pdf.js implements them with the Jasmine testing framework. In the test directory, Jasmine tests are kept together in the **unit** subdirectory.

Other Files

Pdf.js and **pdf.worker.js** specify metadata about the entire application such as build version. These files also **export** important dependencies so that the rest of the application can make use of them by specifying a path to the pieces it wants.

Worker_loader.js specifically works around a bug in Webkit and Chrome 48-.

Pdf.image_decoders.js specifies the specific image decoders to be used by the application in order to keep consistency across different environments.

Interfaces.js is an ‘interface’ that represents the transfer of PDF data when it is being processed by the application.

License_header.js is not an important technical file but it does have the license for terms of use for the application.

Architecture Commentary and Room for Improvement

One potential problem with MVC is that it can obfuscate the degree of coupling between components in different layers. For example, it may be more logical in some scenarios to keep the controller and view in the same logical area, if the view is less independent and more of an extension of the controller. MVC would end up hiding this tight coupling without any added benefit.

However, pdf.js avoids this problem because each of the core, display, and viewer layers can be used in isolation via pre-built production bundles. For example, other PDF display engines can hook into core to receive nicely-formatted PDF data, while custom-made frontends can import pre-built core and display layers for handling business logic. As such, there is a very strong justification for keeping these three layers as separate entities.

The main area of improvement for pdf.js would be in structuring and formatting to make the project more easily navigable and comprehensible for newcomers. As mentioned in the web section, one way to do this would be to use naming conventions and subdirectories to more accurately communicate how certain groups of files are sub-components of another file. Additionally, more comprehensive comments would be a great benefit to anyone learning the code base. Even a header at the top of each file explaining the file's purpose would go a long way toward accomplishing this.

Development Philosophy

Kanban (The Selected Methodology)

We have decided to use Kanban development for this project. This is mostly because Kanban makes it very easy for a team member to allocate an appropriate amount of work to themselves according to their availability in a given week. This makes it easier to deal with the fact that the amount of work able to be done per member will vary due to our differing course loads.

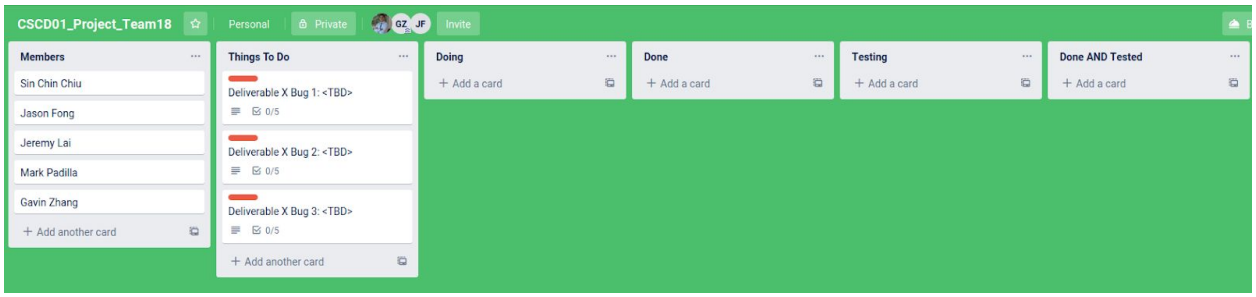
Since we will mostly be working on bug fixes that do not require significant overhead, the lightweight nature of Kanban will be extremely beneficial. Bug fixes for the most part have a few phases: Recreate the bug -> Find the part causing the bug -> fix the bug -> test the fix. This process can be very easily represented in a Trello Board. (A screenshot of our Trello board setup can be seen in the next section.)

Moreover, bug fixes do not, in general, require a great deal of documentation in comparison to developing new features or new systems. This lends itself nicely to Kanban or other incremental development processes where documentation may not be possible due to the consistent and short release cycles.

Kanban is also a great fit for pdf.js specifically because the MVC structure allows developers to work on different issues in different layers without a great deal of coordination involved. As such, it makes sense to use Kanban's highly-independent ticketing system and thus give each of us members freedom to complete tickets and start on new ones at their own pace.

Our Kanban Implementation

To help visualize our Kanban process, we will use the Trello web service. A screenshot of our Trello board setup can be found below:



The columns on this board follow the standard process order for software development iteration: a backlog, work in progress, task completion, testing, and validation.

One area of interest is the ‘Done’ and ‘Testing’ columns. Tickets that have been completed, but not yet validated, are placed in ‘Done’. Then, any other team member is free to reassign that ticket to themselves and place it in ‘Testing’ to show that they are validating it. This procedure still follows the ‘pull’ principle of Kanban, so team members are allowed to take testing tickets at their own pace and only when the WIP limit allows it.

In this way, all testing procedures are visible on the board at all times, and we ensure that the completed code works on multiple systems.

For Blocked items, we will use a custom Trello tag that we named ‘Blocked’.

One part of Kanban that we plan to change is the WIP Limit. Specifically, we will have a ‘dynamic’ WIP Limit that changes each week according to each member’s schedules and other responsibilities. This is because our school courses will limit the amount of time we can spend on the project on some weeks, while other weeks will afford more time.

Another part that we will change to accommodate our team is the frequency of standup meetings. We will change them from once a day to once a week during our regular Thursday meetings. This is because our class schedules differ and because not all of us will be available to meet each day in person.

One thing we plan to add to this process is Test Driven Development. We believe this will be a good compliment to Kanban because both TDD and Kanban are iterative processes. Additionally, since we are doing bug fixes, TDD will help us verify our bug fixes in a more consistent and rigorous manner.

Other Processes We Considered

Waterfall

The very nature of open source and web applications goes against Waterfall's development philosophy. Web development is incremental by design and many components can change on the fly. Additionally, because open source projects are worked on by a wide variety of different developers that are not formally united by an organization, it would be impossible to organize a concrete planning phase. Additionally, new information via bug reports can arrive on short notice and Waterfall would require that development revert to the requirements phase every time to account for new issues.

For example, were we to develop a new feature for PDF.js, it could be in the form of a web page. On this web page, we could have multiple components. Under a Waterfall-based development cycle, each feature would either need its own Waterfall cycle to make it easier to fix individual components, or we would group everything into a single Waterfall cycle. Either of these Waterfall-based solutions would require a large amount of overhead and significant repetition because Waterfall does not allow you to return to a specific phase, only all the way back to the requirements phase.

Extreme Programming

Extreme Programming would theoretically be even more efficient than Kanban and get the bug fixes and new features done in the least amount of time. This would be perfect in a vacuum, but unfortunately, CSCD01 is not the only course that our members are taking. Due to workload and time commitments required by other courses and other parts of our lives, it is impossible to keep up with the extremely strict time windows required by Extreme Programming.

Additionally, the "listening" phase of Extreme Programming would be beneficial if we were designing a new system for a customer or some targeted user base, but for our purposes it is not particularly useful. For the most part, we will be providing bug fixes for bugs reported by specific users. In this case, outside of the initial bug report and some additional information such as how to reproduce the bug, there would be little need to communicate directly with the bug reporter.

The planning game would be extremely useful in refining requirements, but it also requires a significant time investment in comparison to putting tasks on a Kanban board with rough estimates of time cost. Together with other members' other responsibilities, this was another reason to decline Extreme Programming as our development process.

Rational Unified Process

The Rational Unified Process is a highly formal process. It consists of a great deal of phases and requires significant time spent on risk analysis and planning up-front. Although it is an iterative process like Kanban, RUP possesses a much higher overhead with little advantage in exchange for our specific situation. Since we will for the most part be doing bug fixes, the other phases provided by RUP would be largely unnecessary.

Additionally, RUP is a process that focuses on software safety and stability. PDF.js is open source software for viewing documents, which means that it is held to a lower standard for safety and stability compared to professionally-made licensed software for mission-critical systems. In that sense, RUP would be excessive and unnecessary for our purposes.