Software Architecture - PDF.js (https://github.com/mozilla/pdf.js)

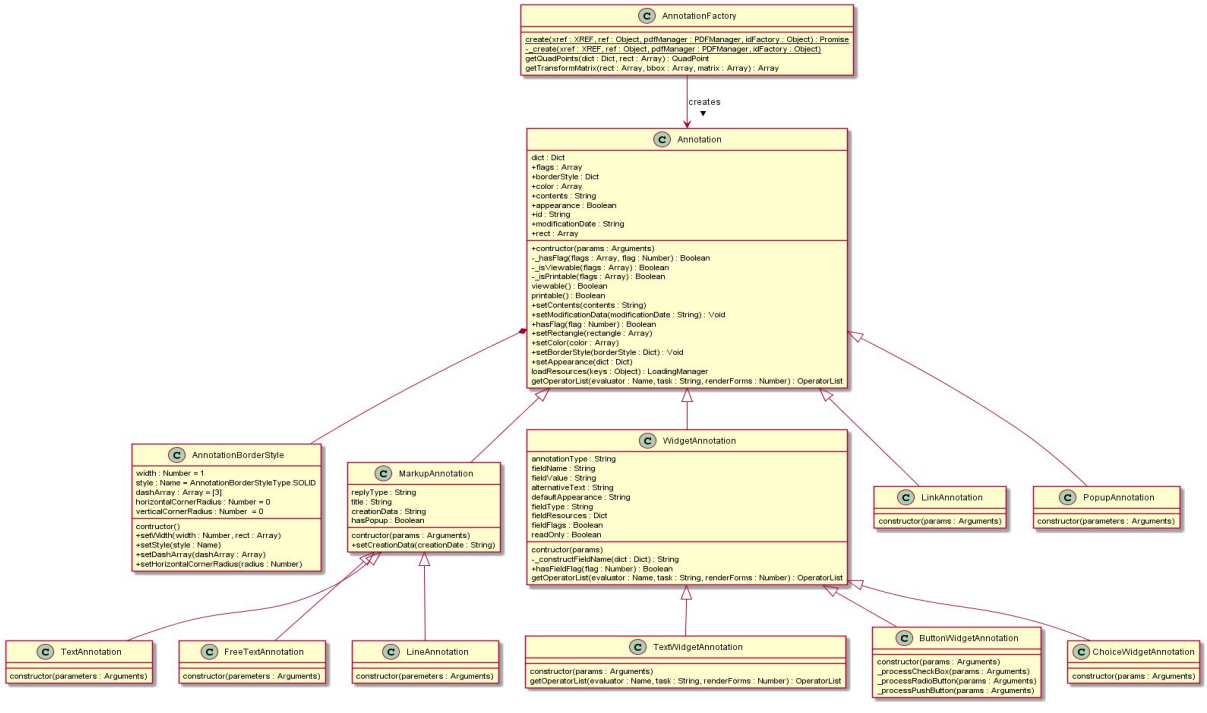
PDF.js is a Portable Document Format(PDF) viewer that is built with HTML5, CSS and javascript. It is a community driven project with the main goal of being a general-purpose, web standards-based platform for parsing and rendering pdf files. It is directly built into Firefox version 19+ and it can be downloaded as an extension on browsers such as Google Chrome.

PDF.js's overall architecture is made up of different layers and understanding each layer is key to understanding the overall project. PDF.js is split into three main layers, namely the Core, Display, and Viewer layer.

Core Layer

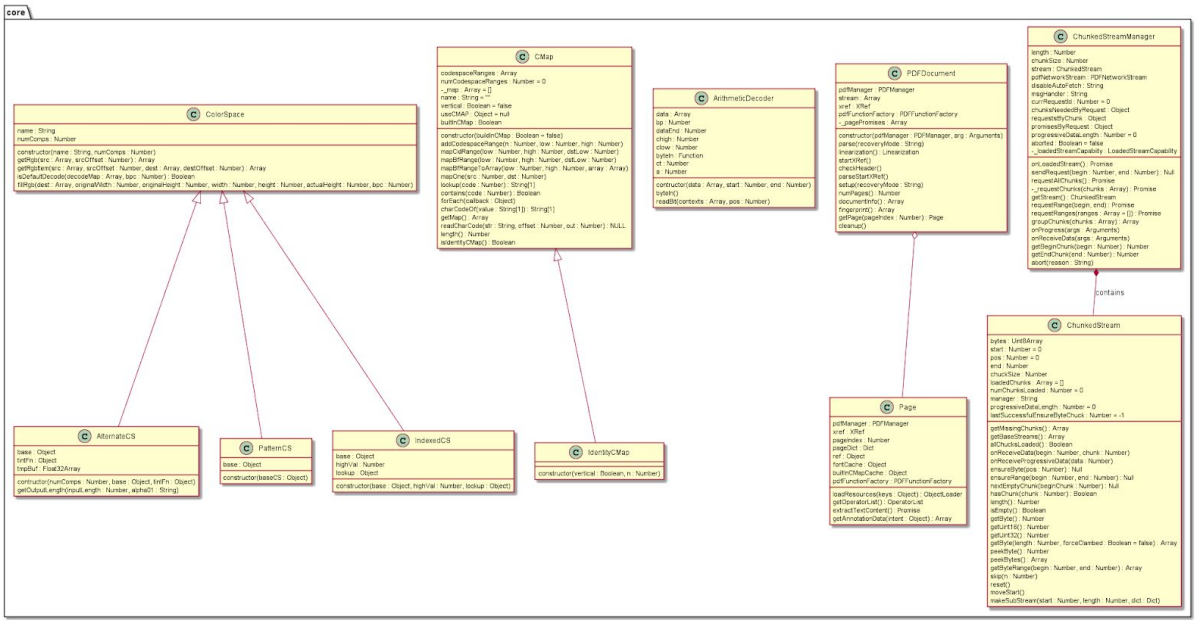
The core layer is the layer where a PDF in binary is parsed and interpreted. It is the foundation for all subsequent layers. It is located in the src/core folder. It has different files for interpreting and parsing PDF files.

The core layer also has several classes for handling multiple types of annotations. Annotations are additional objects that can be added to a pdf document. They help to provide clarification to its content or to expand upon what is already present. Annotations are essential in PDFs as the document format is not easily editable to make the text clearer. This can be seen in the UML diagram below. We found it interesting that they used the factory method design pattern for the creation of annotations, which we like because it makes it easier to create these annotations.



~created using PlantUML

The core layer has different classes for dealing with color. In the UML diagram below we can see a parent class called ColorSpace for dealing with color spaces which are just abstract mathematical models which simply describe the range of colors as tuples of numbers, typically as 3 or 4 values or color components (e.g. RGB). It is inherited by different child ColorSpace classes. For example, the Alternate color space class handles both Separation and DeviceN color spaces. The layer also has files that deal with decoding streams of data, such as ArithmeticDecoder and ChuckedStream. One of the main files in the core layer is the file called document.js. This file contains two classes, namely PDFDocument and Page. The Page class is basically a representation of a single page of a PDF document, while the PDFDocument class holds all the (worker-thread) data of the PDF file. In the UML below, you can see how composition is used to show that page is a part of the PDFDocument.

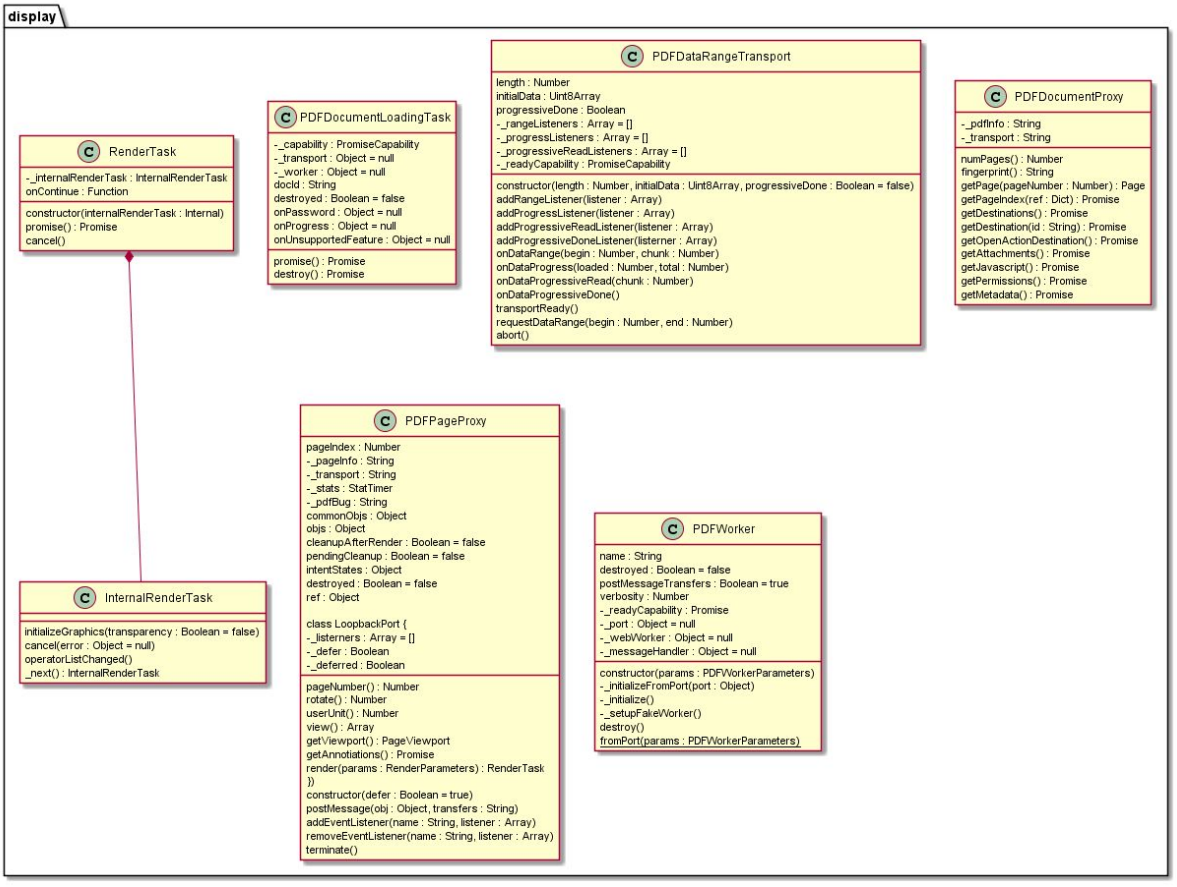


~created using PlantUML

Overall, the structure of the core layer was pretty good. A lot of the files seemed like that didn't really interact with each other, but that was to be expected because the core layer is the foundation for subsequent layers, so many of these files will be used to support files in other layers of the project.

Display Layer

The display layer takes the core layer and exposes an easier to use API to render PDFs and get other information out of a document. The display layer can be found in the src/display folder.



~created using PlantUML

The main file in the display layer is the file called api.js. In this file there are classes that deal with rendering and proxy of a PDF document. These classes can be seen from the UML class diagram above.

.

Class	**Description**
PDFDocumentLoadingTask	Controls the operations required to load a PDF document (such as network requests) and provides a way to listen for completion, after which individual pages can be rendered
PDFDataRangeTransport	Supports range requests
PDFDocumentProxy	A proxy to a PDFDocument (see core layer PDFDocument UML on previous page)
PDFPageProxy	A proxy to a PDFPage in the worker thread

PDFWorker	PDF.js web worker abstraction, which controls the instantiation of PDF documents. Message handlers are used to pass information from the main thread to the worker thread and vice versa. If the creation of a web worker is not possible, a "fake" worker will be used instead.
RenderTask	Allows for control of the rendering tasks
InternalRenderRask	Internally used by RenderTask class

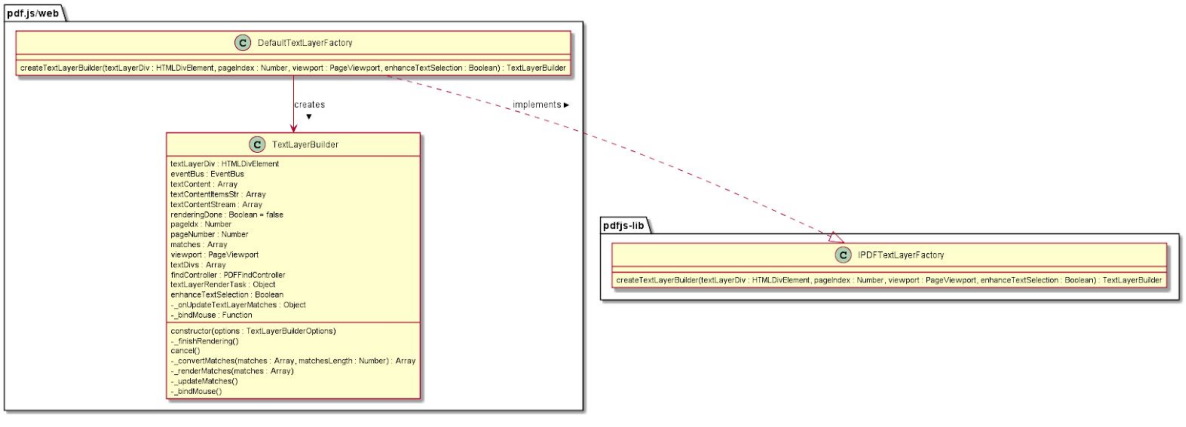
Viewer Layer

The viewer is built on the display layer and is the User Interface (UI) for PDF viewer in Firefox and the other browser extensions within the project. The viewer layer can be found in the web folder.

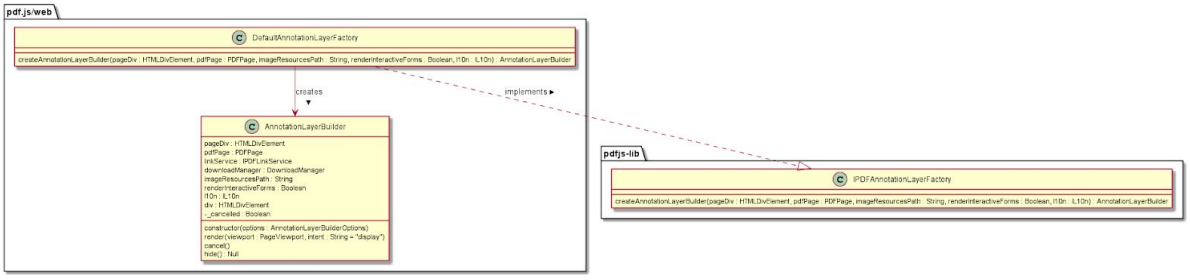
Basically, PDF.js parses raw arrays of bytes into streams of PDF “bytecode”, compiles the bytecode into JavaScript programs then executes the programs. All of these are done in the core and display layers. These are then used to draw on an HTML5 canvas. The drawing is done in the viewer layer. The commands in the bytecode include simple things like “draw a curve”, “draw this text here”, and more complicated things like filling areas with “shading patterns” specified by custom function. Additionally, the stream of commands itself, and other data embedded within like fonts and images, might be compressed and/or encrypted in the raw byte array. pdf.js has basic support to decompress some of these streams (core layer).

The viewer is just a regular HTML page. It is a rendered page that contains:

- A non-scaled canvas
- A text layer, which is an invisible to store text for selection (UML class diagram can be seen below)
- An annotation layer than contains hyperlinks and notes (UML class diagram can be seen below)



~Class diagram for text layer created using PlantUML



~Class diagram for annotation layer created using PlantUML

The implementation of text selection, hyperlinks, zooming in, and so on are done in the viewer layer. The components of the viewer layer (text and annotation layer) are packaged nicely in the pdf_viewer.component.js file.

Conclusions

The overall system is designed pretty good, but there is a lot of code that is not well documented, so reading through some of the code and trying to understand what was going on was pretty difficult. We like how they used a shared folder for code that is used by both the core and display layer.

We liked how the project is split into layers, each layer having a specific role in the overall structure of the system, rather than just putting everything in one big folder.

We found it interesting that there were several ways to run the project. In the viewer layer there is a file called pdf.js that packages the project and creates a prebuilt version of the project that includes a generic build of PDF.js and the viewer, rather than just directly running it from the source code or downloading it as an extension for browsers like Google Chrome.

PDF.js is designed and structured quite good, but there is still room for improvements when it comes to performance and features. For example the viewer takes some time to load some larger pdf documents. I think this is because they draw on an HTML5 canvas, which isn't really good for rendering large images, but better for rendering many smaller images, when compared to something like svg (scalable vector graphics), which is a vector based graphics and is better for rendering large images. I think the next step for the project would be to implement a way for the viewer to distinguish whether to draw on an HTML5 canvas or a svg, based on the rendering speed for a specific pdf document that is required to be viewed.

Software Development Process

For our project, we will be using an agile system development process based on the Kanban Methodology.

One of the most important parts of the Kanban Methodology is the Kanban board which is used to track tasks. We will be using Trello to facilitate our Kanban board. We will not be including Kanban swimlanes on our board, which are separate sections of the board signifying different types of actions. They are not really needed for this project since each deliverable is separated by activity so we should never have multiple active swimlanes, such as a swimlane for bugs and a swimlane for features. The following are the main Kanban board components we will be using:

Kanban cards will be created for every task that needs to be completed. Each card will contain a basic title so it is easy to track, as well as a more detailed description with specifications we agreed on as a team in our meetings. Every card in our "To-Do" section onwards will have a due date to help in prioritizing which tasks to work on. Once each task is picked up by someone they will be assigned to that card so we can keep track of who is working on each task. Cards with a red label are marked as "blocked" meaning there is something preventing the continuation of work on this card, and the card will have a description of why it is blocked.

Kanban columns are the different stages that a task goes through. The stages on our board in order are Backlog, To-Do, Doing, Code Review, Testing, and Done. The "Backlog" stage includes any items which are not of high priority and may not have a due date yet, these tasks are not expected to be started prior to the next meeting. The "To-Do" column contains fully specified cards with due dates which are not currently being worked on but are of high priority. The "Doing" column contains any card which someone is currently working on. "Code Review" is for tasks that are complete but need to be reviewed by at least one other person to ensure there are no major problems with the code and that the specifications of the task are all met. For "Testing", the tasks here just need to be tested. Finally, the "Done" column is for fully tested and accepted tasks that can be added to the main branch on GitHub.

Work-in-Progress (WIP) Limits will be applied to the "Doing" and "Code Review" columns to accurately track the workflow and ensure that we prioritize getting tasks to "Done". For the "Doing" column, there is a limit of four tasks so we can only work on one task each. If we go over this limit there needs to be a good reason to be working on more than one task at a time. If someone comes across a problem in their task that halts progress, it will be marked as "blocked" and moved back to "To-Do" to avoid going over the WIP limit. The limit for "Code Review" will be three cards, so that if you are the final team member to place a card in the column you

must review at least one other task before starting a new task to avoid going over the limit. This is to prevent bottlenecks at code review so that items can get tested and deployed.

Justification for Development Process Choice

Waterfall (not selected)

We decided not to use waterfall because we are aiming to divide our work up into small tasks in order to accommodate our different schedules. Since we will be doing work at different times, we won't be able to all keep to the same schedule.

A benefit to the waterfall process is that each phase of the development process is distinct and this makes it easy to track progress. Our team will try to follow this idea in our development with our kanban board by using different columns to track progress. The kanban board will give us a visual representation of our work and we will be able to track progress by individual task.

A drawback of using waterfall is that it is harder to accommodate for change. In this project, we know that we will be facing a lot of change, from the requirements themselves, but also within our team. If a group member is very busy one week, we will be able to accommodate better by using incremental development. This is why waterfall is mostly unsuitable for our project.

Reuse-oriented (not selected)

This process is not really applicable to our project as we will be writing mostly new code. We will probably re-use existing components in our development to make the code cleaner, but re-use will not be the core task in our development process.

Incremental (selected)

This is the process we chose because it works well with our schedules. We will be meeting weekly to plan each increment, assess progress, and implement changes to our process. Using incremental development will make it easier for us to keep on track.

Incremental development is good for incorporating feedback. We don't have a client to interact with and get feedback from, but we will definitely have feedback of our own and we can improve on our past work in new increments.

A drawback of incremental development is that there are no “visible” releases, which is bad for team managers, but since we are managing ourselves, this will not be a problem. Another drawback is that the system structure can degrade with time, and this is something we need to be mindful of. We will do our best to plan out our work carefully during our weekly meetings and address any issues as they come up.

Plan-driven vs Agile

We chose to use agile development because our deliverables will be defining our increments, and it seems to make sense with that structure. We also want to be able to deal with changing requirements and group circumstances as best as possible. We will still use planning during our weekly meetings, because it is very useful to have a plan, but it will be limited to that week’s tasks only.