UCLouvain

epl

École polytechnique de Louvain

Vers des ebooks plus interactifs

Author: **Jérémy Kraus**
Supervisor: **Olivier Bonaventure**
Readers: **Olivier Bonaventure, François Michel, Anthony Gégo**
Academic year 2019–2020
Master [120] in Computer Science and Engineering

Acknowledgements

I would like to pay my special regards to my supervisor, Professor Olivier Bonaventure for helping me to find ideas to explore and for giving me useful feedback.

I wish to express my sincere appreciation to François Michel for helping me to set the tools necessary to work on this project, for his useful feedback and for helping me to find solutions when I was stuck.

I wish to express my deepest gratitude to Thomas Purser for helping me proofreading this paper.

I wish to thank Damien Gonze for suggesting me to use Progressive Web Apps during one of the meetings, as it turned out to be a really good solution for the problem.

I wish to show my gratitude to my friends and family for providing me with moral support during the moment I needed it.

Lastly I wish to thank all the teacher and assistants at EPL for being available when I had some questions.

Contents

1 Introduction **4**

2 State of the Art **6**
2.1 Comparison . . . 6
2.2 Paper's Comparison . . . 8
2.3 Comparison's Conclusion . . . 9

3 Project's Architecture **10**
3.1 Design Pattern . . . 10
3.1.1 Layered Pattern . . . 10
3.1.2 Broker Pattern . . . 11
3.1.3 MVC Pattern . . . 12
3.2 Architecture . . . 13
3.2.1 Inginious . . . 13
3.2.2 Syllabus . . . 13
3.2.3 Application . . . 14

4 Progressive Web Applications **16**
4.1 Service Workers . . . 16
4.2 Service Workers life cycle . . . 17
4.3 Caching and Fetching Strategies . . . 18
4.4 The Web App Manifest . . . 21
4.5 Service Worker Cache Versioning . . . 22

5 Sphinx Build **23**
5.1 From HTML To PWA . . . 23
5.1.1 Adding The Web App Manifest . . . 24
5.1.2 Adding The Javascript Files . . . 24
5.1.3 Adding The Service Worker . . . 24
5.2 Config Generation . . . 25
5.2.1 App Config . . . 25

5.2.2 Exercise Config 26
5.2.3 Chapter Config 26
5.3 Config Example 26
5.3.1 App Config Example 27
5.3.2 Exercise Config Example 27
5.3.3 Chapter Config Example 27

6 Offline Behaviour **29**
6.1 Service Worker Plugin 29
6.1.1 Caching and Fetching strategy 29
6.2 Pre-caching INGInious Tasks 30
6.3 Background Sync 31
6.3.1 Theoretical Implementation 32

7 Interactive Features **33**
7.1 Resume Feature 33
7.2 Interactive Learning Path 34
7.2.1 Hide Chapters Plugin 35
7.2.2 Chapters Access Outside The Plugin 38
7.3 Restrict The Access 39
7.4 Unrealised features 40
7.4.1 Dark mode 40
7.4.2 Score board 40
7.4.3 Integrating The Moodle Forum 40

8 PWA's Architecture **41**
8.1 Base 41
8.2 CourseManager 41
8.3 Inginious 41
8.4 LocationManager 42
8.5 Syllabus 42
8.6 Utils 43

9 Conclusion **44**

A Manual **45**
A.1 Service Worker Plugin 45
A.2 Modifying conf.py 45
A.3 Creating The Manifest 46
A.4 Running The Sphinx Build 46
A.5 Hide Chapters Plugin 47

Chapter 1

Introduction

Nowadays, nearly everyone owns a smartphone, which is especially true for university students. The motivation of this thesis is to bring the university courses to a platform the students are very familiar with and have access to all day long.

At EPL, some professors utilizes ebooks as a written support for the students. Those ebooks are generated under the format of web pages with the help of sphinx, which is a tool written in python that allows the user to generate clean documentation. Those ebooks integrate exercises that are hosted on inginious, which is a web platform developed by the INGI team at EPL for the purpose of automatizing the correction of students code.

The first objective of this thesis is to extend the sphinx builder for it to generate ebooks under the format of applications that are usable both online and offline instead of simple web pages.

The second objective of this thesis is to add interactivity to the ebooks in order to make the learning experience more enjoyable and efficient.

Now that we have laid out the objectives, we will give an overview of the different chapters contained within the paper.

- **State Of The Art** compares different popular applications with a learning purpose
- **Project's Architecture** gives an overview of the whole project's architecture
- **Progressive Web App** describes the technology we used to build the app
- **Sphinx Build** describes how we extended the standard sphinx html build to generate PWAs

- **Offline Behaviour** describes the behaviour of the app when the user has no access to a connection
- **Interactive Features** describes what interactive features were added to the ebooks and how we proceeded to make these changes
- **PWA's Architecture** gives an overview of the different javascript files contained within the PWA
- **Conclusion** gives an overview of the project
- **Manual** is an abstract chapter that explains how to set up and use the project we developed

Chapter 2

State of the Art

In this chapter, we will describe the state of the start in the domain of applications made with the purpose of teaching information. To write this chapter we tested some popular applications available on the markets.

- **edx** : it is one of the most popular platform for courses of university level. It is used in the context of MOOC (Massive Open Online Course).
- **Duolingo** : it is one of the most popular platform for language courses.

2.1 Comparison

In this section we will compare the different apps analyzed between themselves and our solution which will employ the progressive web app technology which is discussed in its own chapter later in the paper. The comparison is made as an array including the following criteria (which are what we deemed to be important during our analysis) :

- **Format** : the format in which the course is given. Can be either text, video, visual (showing images) or sound.
- **Platform** : the platform on which the app runs. Can be either web-app, iOS or android.
- **Learning path** : the way the course is allowed to be visited. Can be either free (meaning there are no restriction), linear (chapter 1's completion unlocks chapter 2, chapter 2's completion unlocks chapter 3 etc) or multiple (multiple way to visit the course and unlock the content).
- **Exercises** : whether or not the course contains exercises. Can be either yes or no.

- **Offline functional** : whether the course is accessible and functional when the user has no access to connection. Can be either yes, no or pre-downloadable (meaning the user has to download each part of the course individually in prevision of an offline utilization later on)
- **Resume feature** : whether it is possible to resume the course's state from the last time it was used. Can be either yes or no.
- **Gaming experience** : whether the course has elements to make the user feel like he is playing a game. Can be either yes or no.
- **Discussion** : whether the app offers a platform where the users can discuss the course's content in something akin to a forum. Can be either yes or no.

	edx	Duolingo
Format	text + video	text + visual + sound
Platform	web-app + android + iOS	web-app + android + iOS
Learning path	no	linear
Exercises	yes	yes
Offline functional	pre-downloadable	no[1]
Resume feature	yes	no
Gaming experience	no	yes
Discussion	yes	yes

	Our solution
Format	text
Platform	web-app
Learning path	halfway between linear and multiple
Exercises	yes
Offline functional	yes
Resume feature	yes
Gaming experience	no
Discussion	no

As we can see, our solution holds pretty well to the state of the art according of what we deemed important during our analysis.

[1]some content is pre-downloadable but only for paying users

2.2 Paper's Comparison

In this section, we will make a comparison between our solution and the market according to a paper[22] with a related subject to our thesis. Unfortunately it was hard to come by papers that described the specifications of their app, as most papers tended to focus on the positives that applications can have for the students rather than the applications themselves. As a result of the sample of paper being this low, the comparison criteria will probably be quite subjective.

Here is the market comparison according to the paper :

Application Name	**Description**	**Platform**	**Benefits**	**Limitations**
Prodigy[13]	Web-based Math game that follows Ontario standards for mathematics. Uses game first model, education is done via *ingame* combat.	Web-based	• Entertaining, engaging. • Provides feedback for teachers. • Customization of game avatars. • Uses Ontario math curriculum.	• Lack of content customization. • Limited contents. • Need to pay for additional features.
Duolingo[6]	Language learning application.	Web/iOS/ Android	• Utilizes speaking, listening and spelling to teach language. • Earns and uses *Lingots* to purchase in game items. • Can challenge friends/family. • Can see progress of friends/family.	• Lack of content customization. • Need to pay for additional features.
Toca Lab[11]	An application for learning basics of chemistry.	iOS/ Android	• Uses graphics to explain concepts of chemistry and the periodic table. • User engagement using graphics and game-play.	• Lack of content customization. • Need to pay for additional features. • Lack of feedback or option to save.
Motion Math[12]	An application for learning basics of mathematical. Mainly targeted for kids of age 3 to 4 years.	iOS/ Android	• Uses graphics and animation to explain simple addition, subtraction, etc. • Adaptive user engagement using graphics and game-play that gets harder as player progresses.	• Lack of content customization. • No ability to change animation or avatars. • Lack of feedback or option to save.

Figure 2.1: Market Comparison from the paper

As we can see, this study is very biased toward the **gaming experience** feature as it's aimed toward kid/teen rather than university/college students.

According to these criteria, our solution is pretty lacking as we answered no to the **gaming experience** criterion.

2.3 Comparison's Conclusion

What we can draw from the 2 above comparisons is that our solution holds pretty well to the state of the art on a technical standpoint, however on a creative / engaging standpoint it is now really up to pare.

Chapter 3

Project's Architecture

In this chapter we will give an overview of the whole project's architecture by explaining what are its different components and what is their purpose. We will begin by describing some useful design patterns and then highlight how those patterns are present within the project when describing the different components.

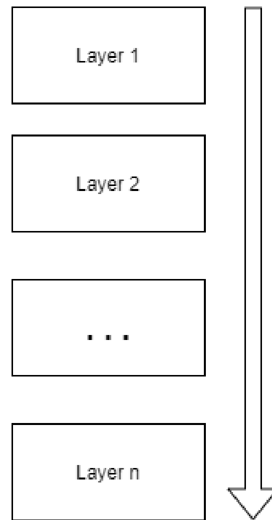
3.1 Design Pattern

One of the most common obstacle in software engineering is the variety of solutions to a same problem. At first glance, this might come over as a good thing, but in some cases it can lead to a disaster as each solution often comes with its own way of handling information and requires a specific structure to work. This creates a disparity in the way developers and project architects manage their application. In order to fix this problem, software engineers have started implementing design patterns[5] under the form of repeatable solutions to commonly occurring problems, therefore standardizing solutions and avoiding unnecessary confusion.

3.1.1 Layered Pattern

The layered pattern[9] is incredibly useful, it challenges the developers to adopt a critical way of structuring their project.

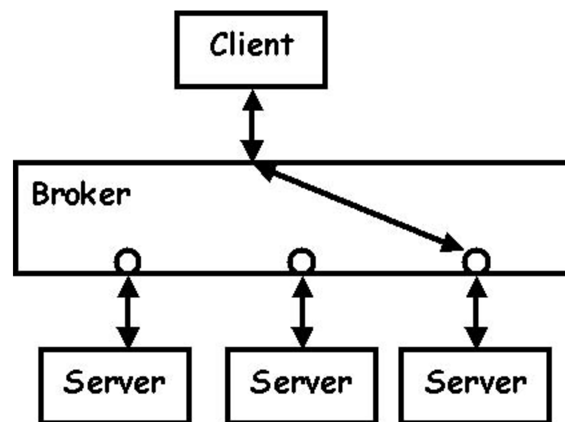
The idea behind this pattern is to divide the code into different "layers", with each of the layers having its own specific role. It also ensures a degree of isolation between, reducing the amount of problems at each level, as it's difficult for them to propagate from a level to another. Each process also relies on the process underneath, which as a result enforces security as a concept of the architecture.



A good way to represent this pattern visually to compare it to a factory. Each room in that factory is assigned to a specific task. The process of completing that task and the errors that might occur are handled in that room and as soon as a result is available, it is forwarded to the next room for the next stage of the development.

3.1.2 Broker Pattern

In an environment where multiple actors have to interact and share information together, the need arises to centralize communication between the said actors. The Broker[2][3] is a pattern designed for this purpose, in other words to standardize the way incoming and outgoing messages are handled.



From http://www.dossier-andreas.net/software_architecture/broker.html

Working with this kind of pattern greatly improves the readability and accessibility

of information as well as the overall management of exceptions. That is because all errors can be dealt with locally, therefore providing a transparent layer to work upon.

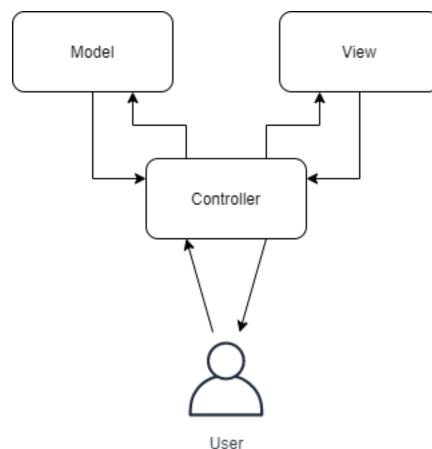
On the other hand, such a pattern requires to know exactly how the communication with other parties is handled, and can only work efficiently if all actors agree on a common way of exchanging the information. Therefore it is important to decide a common protocol for communication between the different parties using the broker, restricting the amount of development required to keep the broker up-to-date.

3.1.3 MVC Pattern

One of the most popular patterns in web applications is MVC[13][12], also known as Model-View-Controller pattern. As all patterns, it focuses on making the code less complex, and improves the overall readability for developers. On a basic level, MVC focuses on splitting the code into a part handling the logic and another one handling the display, and connects them using a controller.

The logic part, also called **model**, focuses mainly on handling the data related to the application. It will for instance handle database database calls by retrieving or updating user specific information. On the other hand, the display part called **view** can focus solely on the outcome, accepting structured data as an input and generating outcomes through various templating systems. For instance, while generating the front-page of a generic web-site the view might be passed a list of links to display in the navigation bar, which it will then proceed to display using adequate templates.

Finally, the **controller** works as a of middle man, requesting information from the model and passing it to the view in order to create response adapted to the request.



3.2 Architecture

Now that the basic designs and patterns have been explained, let's consider the global architecture of the project. First, we will begin by explaining the different parts building up the application, then we will finish by explaining how these changes are implemented to fit in the big picture.

There are three major actors, working together in a secure fashion to provide the different services to the end user :

- Inginous
- Syllabus
- Application

3.2.1 Inginious

As explained in the introduction, is a web platform that automatizes the correction of students code . It uses dockers containers to isolate the server from the code that is executed to avoid problems from happening. In the context of our project, its most interesting feature is that it allows the integration of the exercises on other domains via a secure LTI session.

The platform itself is very extensible as it offers a comprehensive plugin system for developers to exploit as shown in the figure bellow.

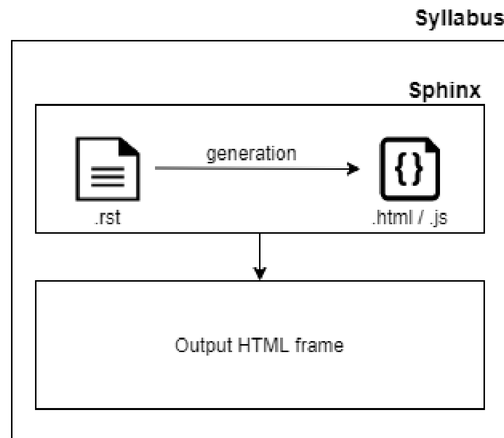
Internally, inginious presents a layered pattern, with the core systems being made available via managers to the potential plugins. This ensures a clean way of extending the application without disturbing the base components, and forces the plugins to interact with all components in a controlled manner.

3.2.2 Syllabus

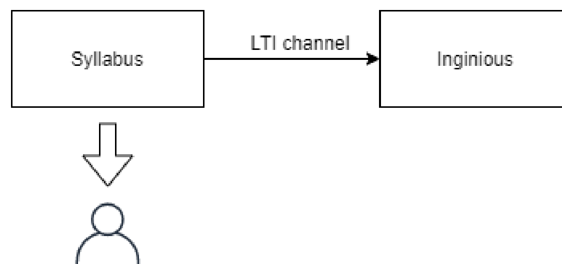
Syllabus is an online platform built for educational purposes, offering access to courses and exercises for the student to use. It relies on several technologies to automatically generate and update information across the platform, with minimal intervention from administrators.

It incorporates a tool called Sphinx to automatically generate a web-based output based on documentation uploaded by the content manager. The generated output is then displayed on the syllabus' website. The syllabus seems to heavily rely on an MVC structure. It further improves on the Jinja templating system used by

sphinx its view component, and injects information based on the request made by the user.



In addition to the text-documents, it also incorporates some exercises for the user by creating a secure channel with inginious, by using LTI. Syllabus therefore integrates something akin to a broker, which is designed to handle communication with inginious, and is able to create and use LTI sessions. LTI however is a discussion of its own and will not be covered in this document.

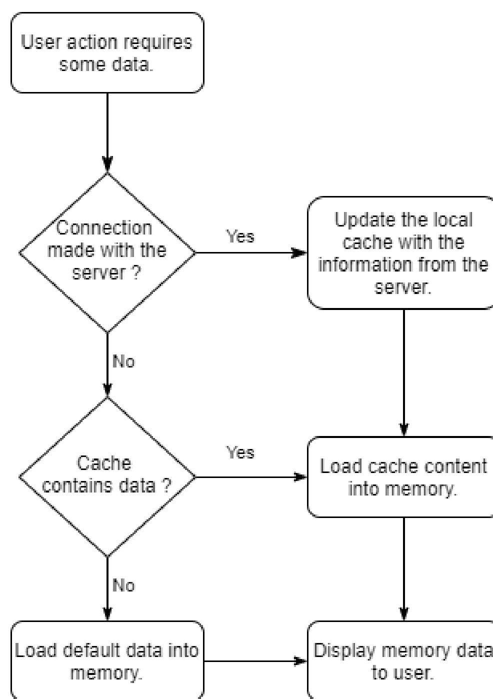


3.2.3 Application

The third and final actor of this project is the app itself. Previously, the user had to be online to access the course's content. However the end goal of this project as explained during the introductory part is to obtain a product that is able to be accessible at any time. The technology used to make this possible will be explained in the next chapter, for now we will focus on describing the overall picture of the application.

The app works closely with the syllabus component to access and update the information on the client device through the syllabus' API.

The content of the app itself is cached and is static. However it is not the same for the content outside the app, such at the exercises hosted on inginious or the information retrieved through the syllabus API(will be developed in a later chapter). Thus, in order to access information from outside, the app will test whether or not the connection to those servers is possible.
If the server is available, it will retrieve the most up-to-date data and cache it, otherwise it will simply retrieve the data from the cache. The figure below explains the basic refresh-cycle of the cache for outside data.



Chapter 4

Progressive Web Applications

A PWA[14] is a type of application that is installable on any device be it a smartphone, a tablet or even a computer via the web browser. This makes the usage of distribution platform such as the app store unnecessary as they are installable directly from the website they emulate.

They are built with regular web languages such as javascript, HTML and CSS. Furthermore, as opposed to standard web applications, they have the ability to work offline much like native applications, this feature is made possible by the inclusion of service workers.

Some additional characteristics of PWAs are :

- theoretically secure against tampering as it is required that the servers offering installable PWA use HTTPS.
- update themselves automaticaly when the author releases an update.
- their layout and user experience are similar to a regular native app

4.1 Service Workers

A service worker[16] is a javascript program running in the background on a separate thread than the the other javascript files. They listen to different events such as fetching content or the service worker installation and offer the possibility to react to them. They serve as a proxy between the app and the internet.

Service workers are very useful in the context of the application because they allow us to cache files needed by the app for offline utilization. Another utility of the service workers is the background sync, it allows the app to defer actions such as forwarding packets until a stable connection is resumed. In our case it would be

very useful as it allows students to solve exercises on inginious while offline, and then handle the forwarding of the answers later on when the connection is stable again. We will extend more about background sync in a later chapter related to the app's offline behaviour.

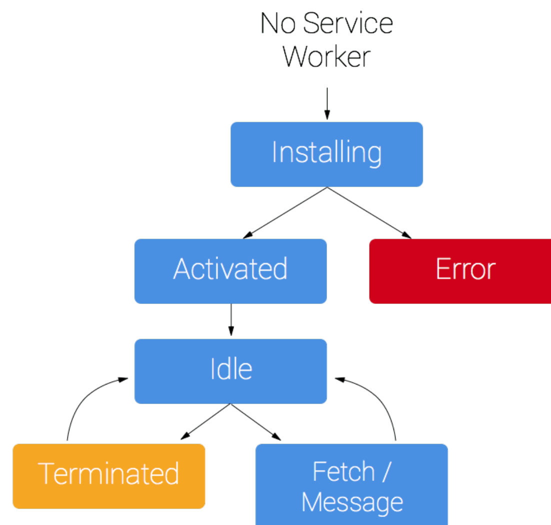
As of the writing of this paper, service workers are supported by the most popular browsers such as[10] :

- google chrome
- mozilla firefox
- opera
- samsung internet
- safari
- microsoft edge

But not every features of the service workers are supported on each of these browser as of yet.

4.2 Service Workers life cycle

To understand how service workers work, we need to dive a bit into their life cycle. Refer to the figure bellow to have a visual support to our explanation.



From https://developers.google.com/web/fundamentals/primers/service-workers

It begins with the app's javascript program attempting to register the service worker by specifying the path where the service worker is located within the app. It is very important to place the service worker in the root directory of the app. That's because the service worker can only control files within its scope. Hence if it is not well placed, the service worker won't be able to control every files contained in the app.

Afterward, the registration triggers an install event on the service worker's side. This event can only happen the first time the browser opens the app for each specific version of the service worker. Meaning that if the service worker is already installed and has been left unchanged then the install event won't be triggered.

Once the installation is completed, if there is no previous version of this service worker already installed, then an activate will be triggered, which will result into the service worker to become active. However, if there is already an older version of the service worker that is active, the new version won't be activated directly after its installation is completed, the activate event will have to wait until every instance of the app running on the machine are closed to be triggered. Until then, the app will use the previous version of the service worker.

Once the service worker is activate, every time the app tries to get some resource, it has to go through the service worker by triggering a fetch event. When listening to a fetch event we can decide which caching/fetching strategy to adopt depending on the type of resource requested.

4.3 Caching and Fetching Strategies

The strategies discussed within this section concern only the resources contained within the app (the ones that are output by the sphinx build).
We will only mention the strategies we decided to use for the app's service worker, but if you are interested you can find more about this topic here[4].

The caching strategy employed by the service worker is called **On install**. It's purpose is to cache the desired assets during the install event.

The inconvenience of this strategy is that the user has to download a lot of files (around 15 MB with the ebook we used when testing) when installing the service worker.

The advantage of this strategy is that the user has directly cached every resources right after the installation without having to visit the whole syllabus beforehand. This makes it possible for the syllabus to be usable while offline directly after the installation is finished.

However, all this caching is meaningless if the app isn't using it, that's why we also have to decide on fetching strategies.

The fetching strategies are the following :

- **Cache falling back to network** : this strategy is pretty self explanatory, it first tries to find the requested resource within the cache but if it can't, it will resign itself to use the network.

  The inconvenience of this strategy is that the data fetched with this strategy will never be updated in the cache. However it's not a problem in our case as we only employ this strategy for static data. If the authors of the syllabus wish to modify static data, they only have to update the service worker's version which will trigger the new service worker to be installed, thus triggering a new on-install event which will update everything contained within the cache, including the modified static content.

  The advantage of this strategy is that the loading speed for data fetched with this strategy will be fast.

  We use this strategy for every static resources contained the sphinx output. (Meaning every non-HTML files).

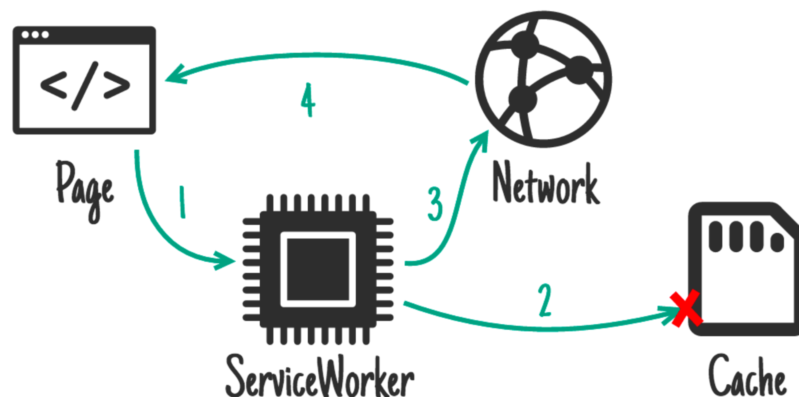


Figure 4.1: Cache falling back to network

From https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/

- **Network falling back to cache** : again this strategy is pretty self explanatory, it first tries to get data by using network, but if not possible it will look for it in the cache.

The strength and weakness of this strategy are pretty much the opposite of cache falling back to network : it will always have up-to-date data, but at the cost of speed when accessing them.

We use this strategy for every non-static resources from the sphinx output. (Meaning HTML files). The HTML files aren't static because their content vary depending on whether the user is logged in to the syllabus or not.

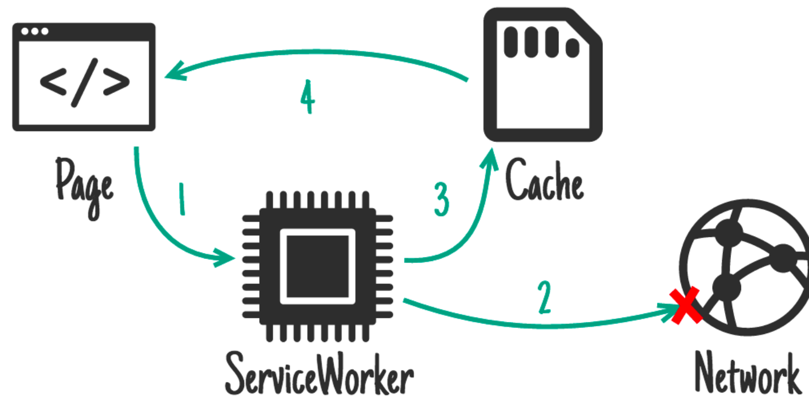


Figure 4.2: Network falling back to cache

From https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/

As explained above, the HTML pages aren't static and change according to the logging status of the user. To avoid the user caching every HTML pages in their offline state during the **on install** event, we decided to restrict the app so that it can only fire the **on install** if the user is logged in to the syllabus.

As mentioned at the beginning of this subsection, the strategy discussed above only concern data contained within the sphinx output, but it is also possible to cache resources coming from outside the app with an **on fetch strategy** which consist to cache resources as they are fetched (for the first time or every time). However after experimenting with this, we noticed that the space taken by those resources is way bigger than the content of the syllabus itself which led us to not keeping this feature in the final product. However this feature could be implemented if we decided to analyze what kind content is cached this way over a long utilization on multiple syllabus and to set filters on what should not be cached this way to keep the cache size reasonable.

Another topic is the exercises contained in the app. As the syllabus serves the inginious exercises within iframes, it is impossible to listen to those fetch events

with the app's service worker. The method to cache them will be discussed in the **Offline Behaviour** chapter.

4.4 The Web App Manifest

The web app manifest[20][21] is a JSON file which contains metadata. Those metadata are mainly used by the browser in order to give a visual experience that resembles native app. It is required that the PWA possess a web app manifest to be installable. Here is a listing of the web app manifest properties used within the thesis and what their role is :

- **name** : the name of the web app that is normally displayed to users
- **short_name** : the name of the web app that is displayed to users when there is not enough space to display **name**
- **start_url** : the url that is first loaded when the app is launched
- **display** : the display style of the app, it can be either of the 4 following values :
  - **browser** : the app will be opened in the browser, like any web page
  - **fullscreen** : the app will use every space available and removes every aspect that looks like a browser.
  - **standalone** : the app will be opened like a native app outside the browser. The app won't keep the browser's navigation interface.
  - **minimal-ui** :the app will be opened like a native app outside the browser. The app will keep the browser's navigation interface.
- **background_color** : the background color of the app
- **theme_color** : the theme color of the app, it's not used by every OS
- **orientation** : the orientation style used to display the app, there are too many possible values to list here
- **icons** : a set of image used as icons to distinguish by the app in different contexts. It is a list of items containing the size, type and source of the images.

4.5 Service Worker Cache Versioning

After learning about our service worker using a cache falling back to network strategy, one might wonder how the app manages to update the content of the cache when a new version of the app is released.

This is possible by modifying the service worker which will result in a new install event to be triggered, resulting a re-caching of all of our static assets as we use an on install caching strategy.

A good way to modify a service worker without changing the way it operates is to update the cache version. An example of how to proceed would be to update the value of staticCache in the figure below to 'app-static-v2' upon modifying the app's content.

```
const staticCache = 'app-static-v1';
```

However, this solution leads to the creation of a new cache, resulting in two new problems.The first one being that the cache uses now 2 time the space it used previously and the second, with the second one being that the app is now lost between what cache to use when retrieving the data.

Those problems are both resolved by deleting the old cache versions upon receiving an activate event.

Chapter 5

Sphinx Build

Sphinx[17] is a tool used to generate documentation in many different formats out of a reStructuredText source. As it is currently used by some professors at EPL to build their teaching resources, we decided to write an extension that directly builds a PWA instead of the format they currently use.

5.1 From HTML To PWA

The fact that sphinx already possesses a builder that outputs documentation in the HTML format, and that PWAs are mainly constituted of HTML files is a very good match. As a result of this, we will work on extending the HTML builder to make it generate installable PWAs. We will now describe the necessary modifications to transform the HTML output into a PWA with the regular HTML builder . They are the following :

- adding the web app manifest
- adding the main java script files and referencing them in every HTML files
- adding the service worker and registering it

Sphinx posses an extension API [18] which allows us to branch our code at different steps of the build phase, access meta data such as the source and destination directories and reference the added files to the different HTML files. This being enough for our need, it is then preferable to write an extension doing what we want instead of directly adding a PWA builder to the source code of Sphinx.

In order to modify and add files with our extension, we branch at the **build-finished** event and do the changes explained in the following subsections.

5.1.1 Adding The Web App Manifest

In order to generate the web app manifest, we read a YAML config to get the necessary information and use it to write the manifest to the output.

As for referencing the manifest in every HTML pages present in the ebook, it is done by using the property of sphinx themes[19] which allow to cleanly add HTML content to the generated pages.

5.1.2 Adding The Javascript Files

To add the javascript files composing the PWA to the output, we simply have to include them in the source folder and sphinx will write them over by itself.

As for getting all the different HTML pages from the app to have a reference to the newly added scripts, we do it by using the a part of the sphinx extension API.

The name of the added javascript files are the following :

- base.js
- CourseManager.js
- Inginious.js
- LocalManager.js
- Syllabus.js
- Utils.js

Their content will be discussed in a later chapter when describing the app's architecture.

5.1.3 Adding The Service Worker

Once every files are written over to the output directory by sphinx, we scan the destination folder and generate an exhaustive list of the files contained in said folder. Then we add this list of files to the assets to be pre-cached by the service worker. This is done by reading the template service worker already placed in the source folder and writing the modified service worker to the root of the output folder, as to have a full access on the content of the app.

5.2 Config Generation

After executing all of the above steps, we now have an installable PWA. But that's not the entirety of our objective which is to have an **interactive** syllabus. In order to reach this goal, we need some information that can, for the most part, only be obtained by looking at the RST or HTML files' content.

To make this information easier to access we have decided to generate some JSON and YAML configuration files containing the required data.

The following subsection focuses only on the content of those config files, their utility will be explained in another chapter.

5.2.1 App Config

The app config as its name suggests is the config used by the application, it's a JSON file containing the following fields and values :

- **course** : the inginious course's name related to this syllabus.
- **basicChapterConfig** is a list containing a very restrictive access to the chapters, as it is generated by the sphinx build while supposing the user did no exercises at all.
- **freeChapterConfig** is an empty list, which means no access restriction to the chapters.
- exercises : the name of every inginious task contained in this syllabus.
- **passingThreshold** is a number between 0 and 100 determining the threshold for an exercise to be considered passed or not.

An alternative to using a config file for "exercises" would be to use the get_tasks() method from inginious to retrieve the exercises name when needed. Ultimately, both solution amount to the same thing, which is making a request to retrieve either the config or the exercises themselves from inginious. Moreover, in both strategies the exercises name list can be cached within the local storage for offline usage in case it has an usage in this situation.

However since we already had to generate an exhaustive list of the exercises to generate exerciseConfig (see below), we figured we might as well add this field to the config in case a rare case arises in the future where the following conditions are met :

- inginious is not available

- it is the user first usage of the app ever, meaning he couldn't have retrieved the exercises list from inginious
- we have a usage for the exercises list which isn't reliant on inginious being available (which is impossible on this version of the app)

5.2.2 Exercise Config

The exercise config is made to be used by the inginious plugin that controls the chapter accessibility (see the chapter about interactive features). It contains for each chapters of the syllabus :

- the name of this chapter
- the exercises contained within this chapter

5.2.3 Chapter Config

The chapter config is also made to be used by the inginious plugin that controls the chapter accessibility. It contains for each chapters of the syllabus :

- the name of this chapter
- the associated id of this chapter, they are set incrementally during the sphinx build
- the ids of chapter required to be passed in order to access this chapter

If chapterConfig is modified, then the basicChapterConfig inside appConfig should be modified adequately as well.

It is important to note that what we mean by "the name of this chapter" in exercise config and chapter config, is the chapter name including the path starting from the directory containing the master document (in our case index.html). This is for the purpose of avoiding name conflict. A good example would be the cnp3 course that has chapters named "dns" in both the "exercises" and "protocols" directory.

5.3 Config Example

For the sake of this example, we will work with a small syllabus containing only 5 chapters named from A to E and 6 exercises.

5.3.1 App Config Example

```
{
"course" : "cnp3",
"basicChapterConfig" : ["chapterB", "chapterC","chapterE
   "],
"freeChapterConfig" : [],
"exercises" : ["exercise1","exercise2","exercise2","
   exercise4","exercise5","exercise6"],
"passingThreshold" : 50
}
```

What we can draw from this config aside of the obvious is that if the syllabus is configured to be restrictive (using basicChapterConfig as default), logged off user will only be able to access chapterA and chapterD as they are the chapters not present in the list contained in basicChapterConfig. However in case it's configured to be non-restrictive (using freeChapterConfig) then logged off user will be able to access every chapters.

5.3.2 Exercise Config Example

```
chapterA:
- exercise1
- exercise2
chapterB:
- exercise3
chapterC:
- exercise4
chapterD:
- exercise5
- exercise6
chapterE: []
```

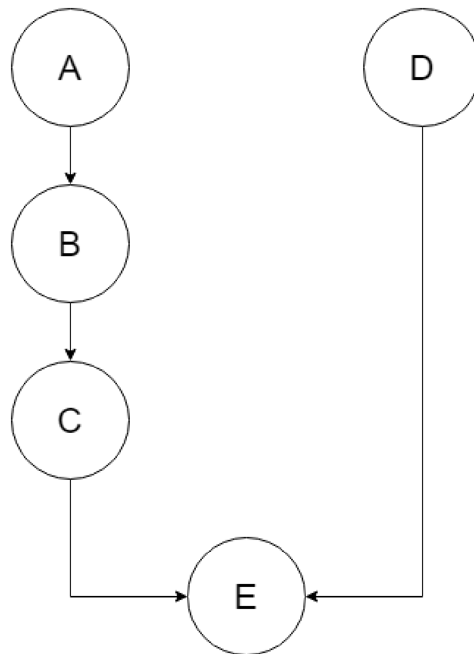
Thanks to this config, we can see the exercises contained within each chapters.

5.3.3 Chapter Config Example

```
chapterA:
    id: 0
chapterB:
    condition:
    -0
```

```
    id: 1
chapterC:
    condition:
    -1
    id:2
chapterD:
    id:3
chapterE:
    condition:
    -2
    -3
    id:4
```

This configuration can also be expressed as a dependency graph such as in the figure below :



By looking at the graph, it's fairly easy to notice that :

- chapterA and chapterD are accessible without any condition
- chapterB requires the completion of chapterA to be accessible
- chapterC requires the completion of chapterB to be accessible
- chapterE requires the completion of both chapterA and chapterD to be accessible

Chapter 6

Offline Behaviour

As explained in the chapter about PWAs, the content of the app is cached by the app's service worker. Nonetheless the syllabus also contains exercises hosted on inginious. This is where the problems arise, because the exercises are loaded within iframes, they can't be cached by the app's service worker as the fetch requests don't go through the service worker.

In order to remedy to this problem, we have decided to add a service worker to inginious by using inginious plugin system[7].

6.1 Service Worker Plugin

In order to have a service worker running on inginious, we have to add 2 new pages to inginious. This is made possible thanks to the inginious plugin_manager.

The first page being the service worker itself, which we place at the root of the inginious domain, so that it can have a control over every files hosted on inginious.

And the second page being the service worker registrant, in other word a simple HTML page containing a script that registers the service worker added to inginious upon being visited.

6.1.1 Caching and Fetching strategy

As opposed to the caching strategy used by the app's service worker, here there is no pre-caching as LTI makes it impossible to directly pre-cache the inginious tasks with the service workers API.

In counterpart, we use the **on fetch** strategy that was described during the PWA chapter. To remedy to possibility that the cache might grow quite voluminous for students using inginious for a long period during their scholarship, it might be

interesting to put some restraint over what the service worker will try to cache. However this idea require a study on a large scale and has not been explored any further as of yet.

On the matter of the fetching strategy, there is nothing new, we use the **network falling back to cache** discussed in the chapter about PWAs. We chose this strategy despite **cache falling back to network** having a faster load time because otherwise cache content would never be updated in case of a change in the exercises.

6.2 Pre-caching INGInious Tasks

As said before, it's not possible to pre-cache the inginious tasks by using the service worker's API. Therefore we have to directly load them ourselves by using the the app's script, so that the inginious service worker can cache them for the first time and make them available for offline usage.

To make the requests for each specific task, we need some LTI parameters that weren't readily available in the app, so we decided to write a small API in the syllabus to access them. Once we have the required LTI parameters, we can simply load all the tasks within iframes, the same way it's already done in chapters containing exercises.

As the loading of all the exercises at once can take a lot of time, we decided to let the user decide when to pre-load them by adding a button visible only to logged in users. Pressing the button starts the pre-loading, freezes the app and displays a buffering bar lasting until the loading is over and then unfreeze the app. This is made to not let the user feel the lag generated by the massive number of request being made simultaneously.

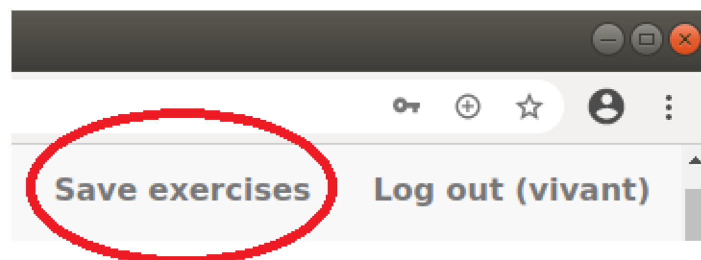


Figure 6.1: Button to pre-cache exercises

6.3 Background Sync

Now that we have found a way to access the exercises even when offline, the next step is to be able to solve those exercises when offline, or more precisely, to be able to send our result back to inginious. This is made possible by the utilization of background sync.

As explained in the chapter about PWAs, background sync[1] is a feature of the service workers. It allows the app to guarantee that a given action will be performed even if the user closes the app or loses connectivity.

This guarantee is made possible by deferring the responsibility for the given action to the service worker, which is running on its own thread separately from the app. Deferring the responsibility to the service worker is done by firing a sync event, which is listened to by the service worker. The service worker is programmed to perform the desired action in the stead of the code segment that fired the event upon listening to the event. The only requirement being to program the said action to return a javascript promise[8]. This guarantees that the sync will be rescheduled in case of a failure and this until it is successful. With the re-sync scheduling utilizing an exponential back-off, to avoid any flooding.

However, this great feature comes with one inconvenience. This being that it is not possible to forward data alongside firing the synchronization event. the only possible workaround is to store this data in the browser's indexedDB upon firing the synchronization event, and then retrieve the data on the side of the service worker when necessary. To answer why indexedDB rather than local storage, the reason is simply that the local storage can't be accessed by the service worker.

Refer to the figure below to summarize visually what was explained in this section.

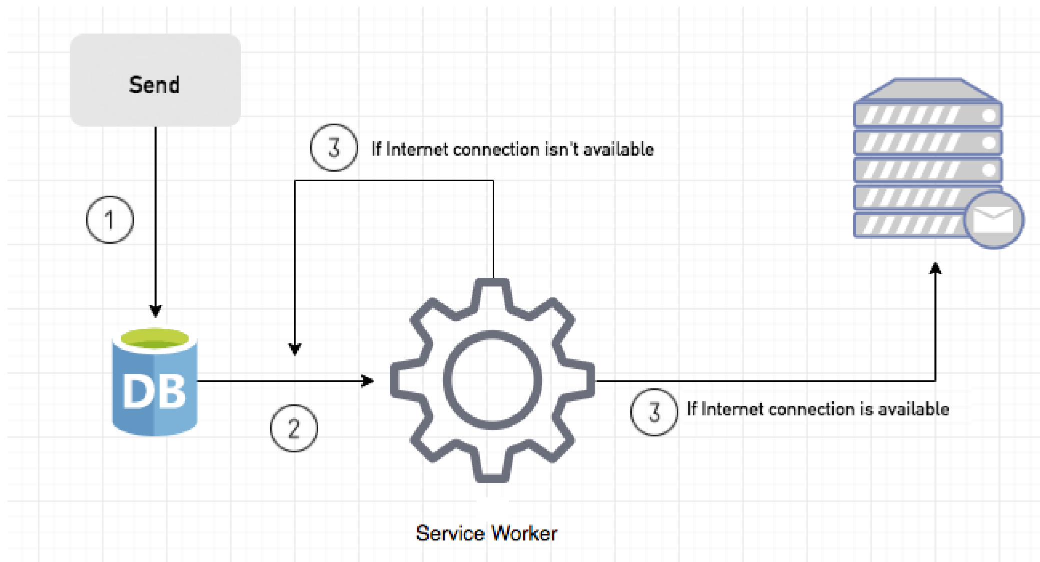


Figure 6.2: Background Sync

From https://davidwalsh.name/demo/background-sync/bg-sync-lifecycle.png

Unfortunately, we didn't have the time to properly implement this in our project. Nevertheless, we will still explain how we would have done it if given more time in the next section.

6.3.1 Theoretical Implementation

Given our theoretical understanding of the matter, the implementation of background sync would require the 2 following modification :

On the syllabus side : replacing the code in charge of sending the answer back to inginious upon pressing the submit button by a code firing the synchronization event for this exercise and storing the answer into indexedDB

On the sphinx-build side : adding to the generated app service worker a code segment listening to the sync event with some pattern matching able to match a separate event for every single exercises contained in the syllabus. With every match returning a promise that forwards the answer for the given exercises, which is retrieved from indexedDB, to inginious as the syllabus would have normally done itself.

Chapter 7

Interactive Features

As said previously, one of the goal of the thesis is for the ebook to be more interactive than simple HTML with exercises.
The reason we wish to add interactive features is to improve the user's experience. Thi can be done in many ways such as improving the enjoyability by adding game mechanics or improving the efficiency by reducing the amount of time wasted by the user.

Hence we decided to add the features discussed withing this chapter to the app.

7.1 Resume Feature

The resume feature as its name suggests is a feature that keeps track of the user's location within the app, such that when the user closes the app and opens it again, the app can resume from where the user previously left off.

This feature's purpose is to save the student's time from trying to remember where he left off and going there himself. Ideally this feature should be an option that the user can toggle, but as its downside isn't consequent, we didn't take the time to implement it as of yet.

This feature is implemented by keeping track of the **window.location.href** at every page changes and **window.scrollY** at every scrolls, and storing them in the **local storage** (which is permanent) in such a manner that when loading a new page, the app checks the **session storage** (which resets after every usage) to see if it's the first page visited during this session, if so it resumes to where the user left off thanks to the information contained within the **local storage**.

A thing to be wary of is that the scroll event of the window executes a segment of code on each user scroll. This might lead to a performance concern under some condition.

In the case the event would trigger a substantial piece of code, it would be interesting to implement some sort of "safe-mechanism" to avoid overloading the CPU. However in our case, the script executed when receiving a scroll event is so light that integrating such a mechanism would be counter-productive, as the check required to execute such a mechanism would use more resources than the code keeping track of the scroll position itself.

7.2 Interactive Learning Path

Our second feature is a bit more complex, it aims to create a way to restrict the access to different chapters of the syllabus until certain prerequisites are met. In our case, the prerequisite is having a passing grade on a required set of inginious exercises.

As of now, a passing grade is a score above the global threshold specified in the app config, however it should be possible to add a specific threshold for each exercises by performing some slight changes to the structure of exercises config and the code segment tasked with deciding whether an exercise counts as passed or not.

Adding a specific threshold for each exercises rather than a global threshold would be interesting because it would allow the professor to put more weights to the exercises he deems as absolutely necessary and less weights on the more advanced ones.

This feature's purpose is to force the students to have a good enough understanding of the basis before diving into the harder parts of the course.

We already discussed the different config files that this feature requires during the **sphinx uild** chapter. There we made an analogy comparing the chapterConfig which file, which is a direct representation of this feature, to a dependency graph. Indeed this feature can be formalized as a dependency graph where the vertices without any other vertices pointing to it are the chapters that are available at the beginning, and the vertices which are pointed to by other vertices as chapters requiring the chapters pointing to it to be completed in order to be accessible.

As the grades for the exercises are hosted on inginious, we need a way for the syllabus to access them. To do so, we decided to write an inginious plugin that works as an API that returns a list containing the chapters that the user can't access given his current progress.

7.2.1 Hide Chapters Plugin

This subsection is dedicated to explaining the plugin mentioned in the above paragraph

7.2.1.1 Plugin Inputs

The plugin creates a post page on inginious to whom we must provide the passing threshold and the name of the inginious course that corresponds to this syllabus' course. This is where the **app config** mentioned in the Sphinx Build chapter comes in handy as its **course** and **passingThreshold** fields contains exactly the information we need here.

The inginious course name is necessary for the two following reasons :

- We need it to retrieve the exercises grades of the student with the inginious API.
- We need it to distinguish which **exercise config** and **chapter config** to use with the algorithm. Indeed as the end goal is to have interactive syllabuses for many different courses, the config files contained in the plugin are labeled with the inginious course name during the sphinx build.

As for the passing threshold, we need it to discern what exercises count as passed or not passed.

However, the inginious course name by itself isn't enough to retrieve the student's exercises grades by using the inginious API, indeed we also need the student's username. Moreover, as the student's username on the syllabus and on inginious aren't necessarily the same, meaning that we can't simply forward it in the same manner as the inginious course name.

Thankfully, inginious is able to retrieve the username if it is accessed by LTI. In order to establish the LTI session, we added an API to the syllabus that makes use of already existent code within the syllabus to retrieve a LTI token and then uses it to create a LTI session with the inginious plugin. The API utilizes the **exercises** field and the **course** field from the **app conf** in order to get the LTI token instead of hard coding a course name or a task name that might disappear over time.

There is no real downside to hard coding those values, but as there is not a huge gap in time required between simply hard coding those values and adding those fields to the app config. Moreover, it's possible that other needs for those values might appear in the future, thus further reducing the gap of time invested between the 2 solutions. Furthermore, as a result of app config being stored in the cache, the gap in performance between the 2 solutions is small. All of these points made us lead toward implementing the second option as we considered it to be cleaner.

7.2.1.2 Plugin's Inner Working

Once the plugin has access to both the inginious course name and username, it can access both of the config files (chapter config and exercise config) for this course as well as the student's exercises grades.

Those data are then forwarded to the algorithm tasked with returning what chapters cannot be accessed by the user given his current progress. The reasons we need the config files are the following :

- We need a way to know what chapters are contained in the syllabus and what are the necessary for it to be accessible by the user. This need is met by the content of chapter config
- As explained above to be considered as passed, every exercises contained in the chapter must have a grade greater than or equal to the global threshold contained in app conf. But in order to check whether a chapter counts as passed or not, we need a way to know what exercises are contained within each chapters. This need is met by the content of exercise config

Given the above information, it's fairly easy to compute the chapters that count as passed, hence we will only focus on the part computing the non-accessible chapters. Toward this end, we provide a pseudo code of the algorithm we use to compute the non-accessible chapters.

```
def getNotAllowedChapters(allowedAndPassedChapters, chaptersLeft,
passedChapters, chaptersDict):
    newChaptersDict = {}
    newChaptersLeft = []
    if chaptersLeft is empty:
        # algo is over, every chapter are accessible
        # thus returning an empty list
        return left
    else:
        for chapter in chaptersLeft :
            if chapter is allowed in regard to allowedAndPassedChapters
            + its accessibility condition
            and  if chapter is in passedChapters:
                append chapter to allowedAndPassedChapters
            else:
                append chapter to newChaptersLeft
                append chaptersDict[chapter] to newChaptersLeft
        if newChaptersLeft == chaptersLeft:
```

```
            #algo is over as it has reached a point where no new
            #chapters can be removed from the non accessible list
            return left
        else:
            #recursing till we reach one of the 2 others return
            return getNotAllowedChapters(allowedAndPassedChapters,
            newChaptersLeft, passedChapters, newChaptersDict)
```

We also provide the line code showing the parameters state upon the initial call to the algorithm.

```
return getNotAllowedChapters([],[" "], passedChapters, chaptersDict)
```

With passedChapters containing the name of the chapters whose all exercises have a passing grade. That's where we must establish the difference between passed chapters and allowed chapters. While the concept of passed chapters has already been described above, it isn't the case for allowed chapters. An allowed chapter isn't necessarily a passed chapter, it only needs its "condition" to be fulfilled. As for chaptersDict, it simply is a dictionary with the content of **chapter config**.

As you can see, the algorithm is recursive and ends either when there are no chapter left to consider or when the algorithm can't add anymore chapters. Its complexity is $\Theta(n^2)$, with n being the number of chapters inside the syllabus. This upper bound should be reached only if the dependency graph equivalent to our path is a hamiltonian[6] path. A path being hamiltonian only if it passes by each exactly once.

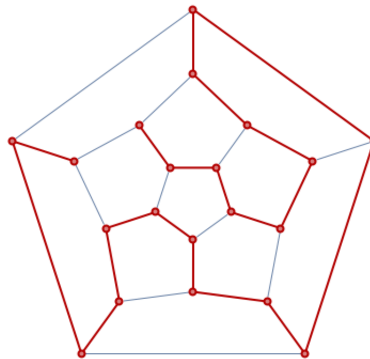


Figure 7.1: Example of hamiltonian path

From https://i.stack.imgur.com/4yHMv.png

The data returned at the end, "left", is an array containing the name of every chapter that aren't allowed to be displayed, such that when this array is sent back to the app, it can filter them out of the HTML page.

By default, the maximum recursion depth set by python is 1000[11], but it's possible to set it up to 1 000 000 if desired.

In order to avoid overloading the inginious server that's running this plugin, we decided to add a code segment to the sphinx build and the plugin that checks if the number of chapter is too big.

On sphinx's side :

- if the syllabus has more than 200 chapters, a warning notifying that it's will be printed on the console
- if the syllabus has more than 500 chapters, a warning notifying that the interactive learning path feature will be disabled for online users.

.

On the plugin's side, if the syllabus has more than 500 chapters, it will return an empty list, which means that every chapters will be accessible.

However in the context of syllabuses for courses given over only one semester as it's commonly done at EPL, those cases are very unlikely to happen. As an example, the syllabus we worked with during the development of this project, which is one of the most voluminous syllabus amongst the one we had access to during our scholarship contained only around 50 chapters.

7.2.1.3 Plugin Access

At the moment, the plugin's API is accessed on every page change. Another approach would have been to access the API only when visiting the master document (index.html).

As the syllabus makes it possible to go from a chapter to the previous/next one, we chose the first approach with the goal of providing the most up-to-date advancement for the user without forcing him to navigate back to the master document. However if we were to notice that this decision leads to too much requests being made to inginious, it should be easy to modify the app and switch to the second solution even though it might confuse some students.

7.2.2 Chapters Access Outside The Plugin

What was discussed above is the behaviour of the feature when the user is logged in and has access to an internet connection (also supposing the servers for inginious

and the syllabus are accessible).

When the user is logged in and there is no connection available, the app utilizes the last list of chapters to filter out that was sent by the plugin, as the latest value is always stored in the local storage by the app upon receiving it.

As for the feature's behaviour when the user is logged out. The app utilizes either basicChapterConfig or freeChapterConfig from the app config.

We decided to give the choice to the professor between those two possibilities depending on whether they are fine with the students having a way to bypass the feature or not. For now, the choice of which config is used is hard coded in the app's **CourseManager.js** file, more specifically in the return value of the **getBaseChapters** function. For more information about the app's architecture, refer to the section about the topic in a later chapter.

7.3 Restrict The Access

Finally, let's discuss how the app manages to restrict the chapter access.

We decided to simply make the links that are filtered non-clickable as it was the easiest solution at the time given our knowledge. The downside of this solution is that students looking at the source code can still access the restricted content despite the alteration performed by the app.

Later on, when we were informed of the possibility to modify the syllabus code itself instead of only using javascript within the app, we noticed that the best idea would have been to directly tamper with the HTML's content that is sent to the app by the syllabus server, as to not even contain links for chapters that aren't allowed to be accessed for the moment. This would make cheating more difficult but not impossible, as the students could still tell each other the url linking to various chapters that other students might not have reached yet.

Unfortunately, by the time we noticed it, we didn't have enough time remaining to completely modify our approach for this feature.

7.3.0.1 Learning Path Configuration

In order to modify the learning paths set by the chapter config as the default one generated by the sphinx build is probably not the best. It can simply be done by modifying the ids contained in the condition field of chapterConfig.

7.4 Unrealised features

In this section, we will describe the features that we imagined during the development but didn't implement.

7.4.1 Dark mode

A dark mode would be useful for students studying in a dimly lit environment. For it to look good and be effective, it would require us to also modify the color background of the images during the sphinx build.

Alas we decided not to add it as it would be too bothersome in term of time while only bringing us a small learning outcome.

7.4.2 Score board

A score board would be interesting as it can drive some individual to try harder due to their competitive nature. However due to the outcome of the exercises being binary (either correct or false), every student with the max score would end with the same rank. This kind of score board would be more fitting in a more free environment where students have to compete in order to optimize their solution, such as trying to reaching an optimum value in an optimization problem. But the nature of these exercises would go against our **interactive learning path** feature.

Moreover, after some research, we noticed that inginious already possess a score board function. Which means that the only learning outcome for us would have nearly nonexistent.

As of yet the plugin only allows one type of condition which is a strict AND, but the algorithm and the config file were designed with the idea of adding different type of conditions without having to do too much changes.

7.4.3 Integrating The Moodle Forum

Moodle is the learning platform used at ucl to host most of the courses. It generally comes in with one or more forum for each course.

As moodle also uses LTI, it should be possible to integrate them in the syllabus to allow students to share their knowledge between themselves through the app instead of having to go through moodle. We didn't implement this because of a lack of time.

Chapter 8

PWA's Architecture

In this chapter we will describe the purpose of the different javascript modules composing the PWA.

8.1 Base

Base is the app's heart, its purpose is to call the other scripts function when necessary, and to manage the layout change when the user decides to cache the exercises for offline usage.

It also start by initializing the syllabus and inginious module. The initialization are necessary to make sure the syllabus API is accessible, because those modules need it to be functional. If the initialization fails, the app will only be able to keep the resume feature working, but every other added functionality will be halted until the user reloads the page.

8.2 CourseManager

The purpose of course manager is to handle the operations linked to the interactive learning path feature such as :

- restraint the access to chapters that shouldn't be accessible
- save and retrieve the most up-to-date chapter access state for offline usage

8.3 Inginious

The purpose of inginious is to handle operations that need to connect with the inginious server, these are the followings :

- register the inginious service worker, by loading the inginious page that registers it into an invisible iframe
- retrieve the list of chapter that can't be displayed from the inginious hide chapters plugin.

8.4 LocationManager

The purpose of location manager is to handle the resume feature operation such as :

- save in local storage the user's last page position every time the user navigates to another page
- save in local storage the user's last vertical scroll position every time the user scrolls down a page of the app
- navigates to the last user position when the user restarts the app

8.5 Syllabus

The purpose of syllabus is to handle operations related to the ebook and the interactive syllabus API such as :

- check whether the user is logged in or not
- register the app's service worker, only if the user is logged in
- accessing different resources from the syllabus such as :
  - the syllabus course name, from the current URL, as it is a necessary argument for the newly added syllabus API
  - the inginious URL trough the syllabus API, as hard coding it might rise some problems in future releases or in testing environment
  - the app config
  - the LTI parameters of a given task for this course

8.6 Utils

The purpose of utils as its name suggests is to provide some helpful operation for the other modules, there are too many operations to list here, but we still give a small overview of the different types of operation it performs :

- operations related to URLs
- saving and accessing the configs into the local storage
- modifying the DOM (when adding iframes)

Chapter 9

Conclusion

Thanks to the very powerful tools that are service workers, the very handy inginious plugin system and the syllabus API, it was possible to meet our original goals except for the possibility to submit exercises while being offline.

A perspective on the final product would be that despite being functional, the app still has some room to grow mostly in term of customizability, with the following ideas having been in our mind during the implementation :

- adding other types of condition than strict AND for the learning path
- adding a passing threshold specific to each exercises in the ebook
- implementing the background sync idea we discussed in the offline behaviour chapter
- adding the unrealised features discussed in the interactive features chapter

Appendix A

Manual

This chapter is a manual on how to install project and how to modify the configs, it assumes you have a working installation of inginious, sphinx and the syllabus. All the necessary files to follow this manual can be found in the project's github[15].

First of all you should copy paste setupPWA from the repository to the directory containing your ebook.

A.1 Service Worker Plugin

If it isn't already done, you should install the service worker plugin and add it to inginious' configuration.yaml file.

A.2 Modifying conf.py

In order modify conf.py, take a look at new_conf.py inside the github. You should make sure your version of conf.py has at least all the imports of new_conf.py. Then, you should modify the following strings value inside conf.py :

- html_theme = 'pwa'
- html_theme_path = ['setupPWA/theme']

If you were already using a sphinx theme, you should modify **/setupPWA/theme/pwa/** to your desire, as long as you keep the following lines from our layout.html and don't modify **/setupPWA/theme/pwa/static/js/** and **/setupPWA/theme/pwa/static/json/** :

```
{% block extrahead %}
    <!-- layout overloading -->
    <meta name="theme-color" content="#2F3BA2" />
    <meta name="description" content="Notebook App">
    <link rel="manifest" href="/syllabus/sphinx/manifest.json">
{{ super() }}
{% endblock %}
```

You can obviously modify the values for "theme-color" and "description" to your liking.

Then you should add the following strings to conf.py :

- ingiCourseName = **"the name of the matching course on inginious"**, in the case of our development it was "cnp3"
- pathToSyllabus = **"the path leading to the main folder of the syllabus**, this value depends on what you wrote inside configuration.yaml. By default it should be "/syllabus/default".

The figure above for the sphinx theme, the path for the manifest must be equivalent to pathToSyllabus, modify the path accordingly.

Once that's done, you should now copy all the code from new_conf.py over to conf.py.

A.3 Creating The Manifest

To create the manifest of your choice,
you should modify **/setupPWA/theme/pwa/static/json/config.yaml** and add your icons into **/setupPWA/theme/pwa/json/img/icons/**
A default config.yaml is already provided, but you can change the values according to what you want.

A.4 Running The Sphinx Build

Now, everything is set for you to run the sphinx-build command, it can be run the usual way, by typing this command :

```
sphinx-build --keep-going -b html <source_dir> <output_dir>
```

With <source_dir> being the directory where your project is located, and <output_dir> being the directory where you want the app to be generated.

A.5 Hide Chapters Plugin

Now that the app is generated, the last necessary step is to install / update the hide chapters plugin.

The first necessary step is to modify the generated **chapterConfig__coursename.yaml** to match the learning path you have in mind.

Then, if you have yet to install the plugin, you should download the plugin from github onto your machine, and then add the freshly generated **exerciseConfig__coursename.yaml** and **chapterConfig__coursename.yaml** to the plugin, next to ___init___.py. Once that's done, don't forget to add the plugins to inginious' configuration.yaml file.

Otherwise, if the plugin is already installed, then you should add the new config files only if they are different from the ones already present in the plugin, or if it's the first time adding the config files for this course. Once they are added to the plugin, you have to reinstall it.

Bibliography

[1] Background sync. https://developers.google.com/web/updates/2015/12/background-sync.

[2] Broker pattern. http://www.dossier-andreas.net/software_architecture/broker.html.

[3] Broker pattern complement. https://www.oreilly.com/library/view/pattern-oriented-software-architecture/9781119963998/chap12-sec005.html.

[4] Cache and fetch strategy service workers. https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/.

[5] Design patterns. https://sourcemaking.com/design_patterns.

[6] Hamiltonian path. https://en.wikipedia.org/wiki/Hamiltonian_path.

[7] Inginious plugin system. https://inginious.readthedocs.io/en/v0.6/dev_doc/plugins.html.

[8] Javascript promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[9] Layered pattern. https://medium.com/@priyalwalpita/software-architecture-patterns-layered-architecture-a3b89b71a057.

[10] List of browsers supporting service workers. https://caniuse.com/#search=service%20worker.

[11] Max recursion in python. https://www.geeksforgeeks.org/python-handling-recursion-limit/.

[12] Mvc. https://developer.chrome.com/apps/app_frameworks.

[13] Mvc complement. https://en.wikipedia.org/wiki/Model-view-controller.

[14] Progressive web app. https://web.dev/what-are-pwas/.

[15] Project's github. https://github.com/xenoryo/memoire-release.

[16] Service worker information reference. https://developers.google.com/web/fundamentals/primers/service-workers.

[17] Sphinx. https://www.sphinx-doc.org/en/master/.

[18] Sphinx api for extension. https://www.sphinx-doc.org/en/master/extdev/index.html#apis-used-for-writing-extensions.

[19] Sphinx theme. https://www.sphinx-doc.org/en/master/theming.html.

[20] Web app manifest. https://web.dev/add-manifest/.

[21] Web app manifest complement. https://www.w3.org/TR/appmanifest/.

[22] Musfiq Rahman Gregory Petersen, Steven Lyall. A flexible learning framework for kids. In *21st Western Canadian Conference on Computing Education 2016 (WCCCE'16)*, page 3, 2016.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**