Implementing

Progressive Web Apps

using

React

A Practical Guide to create Web Apps that provides a native experience to the users

Enrique Pablo Molinari

Contents

About the Author

My name is Enrique Pablo Molinari. I have been working in the software industry for the last 22 years, working in different software projects from different companies as developer, technical lead and architect. I'm a passionate developer and also a passionate educator. In addition to my work on the software industry, I'm teaching Object Oriented Design and Advance Database Systems at Universidad Nacional de Río Negro.

Implementing Progressive Web Apps is my third book. I have also written Understanding React, a book to start learning React. And Coding an Architecture Style, a book about hands-on software architecture. You can find more about my thoughts on software development at my blog: Copy/Paste is for Word. I would be very happy if you want to ping me by email at enrique.molinari@gmail.com to send thoughts, comments or questions about this book, the others or my blog.

What is this book about?

In this book we are going to implement a Progressive Web App (PWA) using a step-by-step approach. This learning experience starts with an already crafted React application called Task List. To create this application I have used the create-react-app tool with the cra-template-pwa template, which gives us a good starting point for building a progressive web app.

We will go into the details about how to make the web app installable and after that how to improve the user experience by adding offline support to the web app. To make the learning process smooth I will first add offline support for the read-only use cases of the application. After that, I will show how to provide full offline support by using the IndexedDB database and the Background Sync.

I will also explain how to update progressive web apps. How users get notified that there is a new release of the application waiting to be installed and how that actually happens.

This book requires prior knowledge of JavaScript and React.

Development Environment

There are many development environments out there, and you can choose the one you are more comfortable with. In any case if you don't have a preference, I recommend Visual Studio Code (VS Code). And to be more productive I suggest installing the extension VS Code ES7 React/Redux/React-Native/JS snippets which provides JavaScript and React snippets. I would also suggest installing Prettier, which is a JavaScript/React code formatter.

To install an extension, in Visual Studio Code, go to the File menu, then Preferences and then Extensions. You will see a search box that will allow you to find the extensions that you want to install.

Finally, I really recommend configuring VS Code to format your source files on save. You can do that by going to the File menu, then Preferences and then Settings. On the search box type Editor: Format On Save. This will format your code right after you save it.

Chapter 1

Introduction

1.1 Progressive?

No one has doubts about what a Web App mean. But progressive? What does it mean?

The word **progressive** comes from the design philosophy known as progressive enhancement coined by Steven Champeon and Nick Finck in 2003. The idea is that you have to design your Web App to work in old devices, old browsers, with bad connectivity or no connectivity at all, to reach as many users as possible. But, it provides the best user experience for newer devices and browsers. In other words, as described by MDN (Mozilla Development Network):

> "The word progressive in progressive enhancement means creating a design that achieves a simpler-but-still-usable experience for users of older browsers and devices with limited capabilities, while at the same time being a design that **progresses** the user experience up to a more-compelling, fully-featured experience for users of newer browsers and devices with richer capabilities".

As we will see later in this book, there are many new features implemented by modern browsers. For instance, there is one particularly nice that allows your Web App to work offline. However, this and many other capabilities have been added during recent years to browsers at different times and some of them partially supported. So, feature detection is the technique that we usually use to determine if that functionality can be used or not. In this way, you can build a Web App that **progresses**.

1.2 And... Progressive Web Apps?

Progressive Web Apps are Web Apps that thanks to modern browser features, give users an experience similar to native mobile apps.

As described by Google web.dev, on the web platform you can create and run applications to **reach** anyone, anywhere and on any device with a single codebase. On the other hand, platform specific applications (like mobile native apps) are very **rich** and **reliable**. They work regardless of network connection. They can interact with the camera, read your contacts, share your calendar and many others. Progressive Web Apps fit in the middle of these. They can deliver enhanced capabilities like working offline, be reliable, installable and receive push notifications, like platform specific apps. The good thing is that they run on a browser. This is possible thanks to a technology implemented by modern browsers called **Service Worker**. The Service Worker specification was developed by engineers from Google, Mozilla and Samsung and released in 2014. As you can see Apple, the other big player, was not involved. However, Safari does support it. We will describe this technology in the next chapter.

1.3 Capabilities of Progressive Web Apps

Thanks to the service worker and other features that browsers have implemented, we now have several capabilities that allow us to provide a native experience to end users. In this section, we will briefly describe the main progressive web apps capabilities. Some of these might not be (or partially) supported in some browsers, so please make sure you check that as browsers might include them in future releases.

Installable Progressive web apps can be installed in desktops or in mobile devices. Installing a PWA in mobile devices gives the users a similar experience as they have with native apps. An icon launcher is created the same as a native app, it will be listed in the applications section of the mobile operating system to be removed just like any other native app. It will run on its own windows without any browser evidence.

Splash Screen Related to the previous capability, an installed PWA when launched a splash screen is shown while loading. Same experience as native apps.

Offline Support Most of the PWA features were created to enable developers to create web apps that run even when connectivity has gone. The

Service Worker is the piece of software that enables us to provide offline capabilities.

Background Sync This capability allows us to defer the access to server-side APIs once connectivity is back. I mean, under connectivity issues the web app might continue working, storing data on a browser's database like indexedDB and syncing that data once connectivity is back.

Push Notifications This capability enables web apps to receive push notifications in a similar way that is possible for native apps.

Other Capabilities There are other capabilities like Web Share API that allows us to share content with other apps. The Contact Picker API allows us to interact with our contact list to share some limited details. The Media Session API allows us to interact with media keys on keyboards, headsets, remote controls, etc. And additionally, it is also possible to have our PWA on the Android app store, just like a native app. And the same might occur on the Apple Store, depending on the type of the application you are building. PWABuilder can help with this.

Chapter 2

Crafting your First PWA

In this chapter we will convert with a step-by-step approach an existing React application, called Task List, into a progressive web app. It is a full stack application with back-end services written in Java.

2.1 Configure the Development Environment

In this section I will show how to configure and set up the Task List application in your local development machine and explain its architecture. In the next section we will explore what functionality provides and what React components were created to build the front-end.

The Task List application was built using the microservice architecture style. We have two back-end services: the User Authentication and the Task List. To set up these services, you will need Java 11 (or greater) and maven. Please, install them if you don't have them.

There are two ways to set up the system in your local PC: Using a reverse proxy or using localtunnel.

2.1.1 Using a Reverse Proxy

To set up the development environment we first need to start the back-end services. Open a console window, go to the directory you would like to clone the source code to, then run the following commands:

```
$ git clone https://github.com/enriquemolinari/userauth.git
$ cd userauth
```

```
$ mvn install
$ mvn exec:java
↪  -Dsecret=bfhAp4qdm92bD0FIOZLanC66KgCS8cYVxq/KlSVdjhI=
```

These commands will clone the **UserAuth** service source code, then install all the necessary dependencies, compile the source code and finally the `mvn exec:java` will start the **UserAuth** service on port `1234`. The `-Dsecret` argument is the secure key used to encrypt the access token. Once the service finish to start, you will see on the console what is next:

```
... INFO io.javalin.Javalin - Listening on http://localhost:1234/
... INFO io.javalin.Javalin - Javalin started in 247ms \o/
```

Now, to set up the **TaskList** back-end service, run the following commands:

```
$ git clone https://github.com/enriquemolinari/tasklist.git
$ cd tasklist
$ mvn install
$ mvn exec:java
↪  -Dsecret=bfhAp4qdm92bD0FIOZLanC66KgCS8cYVxq/KlSVdjhI=
```

Again, once finished, you will see the following entries in the console window. This service runs on port `1235`.

```
... INFO io.javalin.Javalin - Listening on http://localhost:1235/
... INFO io.javalin.Javalin - Javalin started in 247ms \o/
```

With these in place, you have the back-end services running. Both services use the Derby embedded (in memory) database. Every time you start the service, it generates the database schema again and populate it with some data. The **User Authentication** database is populated with two users: guser/guser123 and juser/juser123. Each of them will have some tasks already created in the **Task List** database.

Let's now proceed with the React application, by cloning the **starter project** repository, install and serve it, with the following commands:

```
$ git clone
↪  https://github.com/enriquemolinari/react-starter-tasklist
$ cd react-starter-tasklist
$ npm install
$ npm start
```

After that you will have the Task List React application running on port `3000`. However, you won't be able to connect with the back-end services due to the same-origin policy. The bank-end services don't have CORS enabled by default, and even enabling it, since authentication works using cookies, the browsers won't send them if you are on `http` (to use `SameSite: None;` you also have to use `Secure`). There are a couple of options to solve this. And yes, this is a bit more complicated than I would like it to be, but it is a real situation you will face if you want to build a single-page application with a microservice architecture that uses a cookie-based authentication mechanism.

I will explain two options to make this work. One is using a **reverse proxy** and the other is using localtunnel. You will be able to start and work with the system using any of these options. Use the one you are more comfortable with.

To go with the first option, you will have to install a **reverse proxy**. With a reverse proxy in place, the browser will only talk to it guaranteeing the same-origin policy. This architecture is depicted in figure 2.2. You can install any reserve proxy, there are many out there. I will provide configuration files for both Kong and Nginx. If you decide to install Kong, after installing it, I recommend configuring it to use the declarative config file (instead of using a database). To do that, first generate the config file with the following command:

```
$ sudo kong config init
```

Then, you must tell Kong, by using the `kong.conf` configuration file, that you want to use a declarative configuration file. Open the file `kong.conf` and set the `database` option to `off` and the `declarative_config` option to the path of your `kong.yml` file as in the example below:

```
database = off
declarative_config = {PATH_TO_KONG.YML}
```

And then, open your `kong.yml` file, and paste there the following content:

```
services:
- name: backend-auth
  url: http://localhost:1234
  routes:
  - name: backend-auth-route
    paths:
```

```
    - /auth
- name: backend-tasks
  url: http://localhost:1235
  routes:
  - name: backend-tasks-route
    paths:
    - /app
- name: frontend
  url: http://localhost:3000
  routes:
  - name: frontend-route
    paths:
    - /
```

This will tell Kong to forward requests from `http://localhost:8000/auth` to the User Authentication back-end service running on `localhost:1234`. From `http://localhost:8000/app` to the Task List back-end service running on `localhost:1235`. And from `http://localhost:8000/` to the React application running on `localhost:3000`. Finally don't forget to start the Kong service:

```
$ sudo kong start
```

Now, if you navigate to `http://localhost:8000/`, you will see the Task List front-end.

If instead of using Kong, you decide to use Nginx (if you are on Windows OS this is a good option, due to Kong at the time of writing this book does not support Windows), please go ahead and install it. After that, find the nginx config file (it will depend on your OS) and paste the following configuration:

```
server {
  listen 8000;
  listen [::]:8000;

  location / {
         proxy_pass http://localhost:3000;
  }

  location /auth/ {
```

```
        proxy_pass http://localhost:1234/;
  }

  location /app/ {
        proxy_pass http://localhost:1235/;
  }
}
```

Note that we are forwarding to the same services as we did with Kong. Pretty similar to that. Make sure nginx is running and then if you navigate to `http://localhost:8000/`, you will see the Task List front-end.

2.1.2 Using LocalTunnel

If you don't want to go with a reverse proxy, the alternative I will propose is to use localtunnel. Localtunnel allows you to expose your local services with a public **https** URL. This option permits to share your apps while you are working and it will also allow us to test the PWA application using a mobile device. To install it, run the following command:

```
$ npm install -g localtunnel
```

Then, using VS Code (or your favorite editor) open the `.env` file from the Task List React application, cloned before in the `react-starter-tasklist` directory. In this file we store the URL of the back-end services that the React application consumes. By default, they will have the URL pointing to the reverse proxy, like below:

```
REACT_APP_URI_AUTH=http://localhost:8000/auth
REACT_APP_URI_TASK=http://localhost:8000/app
```

But, if you are going to use Localtunnel, you have to change the content of the file, as shown below:

```
REACT_APP_URI_AUTH=https://auth1.loca.lt
REACT_APP_URI_TASK=https://task1.loca.lt
```

This change requires to re-start the Node server. Note that instead of pointing to localhost it will point to real secure URLs. Now, you need to start a tunnel for each service:

```
$ lt --port 3000 --subdomain web-epm
```

```
$ lt --port 1234 --subdomain auth1
```

```
$ lt --port 1235 --subdomain task1
```

Each command above creates a public secure URL, starting with the specified subdomain and each request will forward to the localhost on the specified port. After setting this up, open a browser and navigate to each URL: `https://web-epm.loca.lt`, `https://auth1.loca.lt` and `https://task1.loca.lt`. You will have to read a friendly reminder and click a button to proceed, as illustrated by figure 2.1.

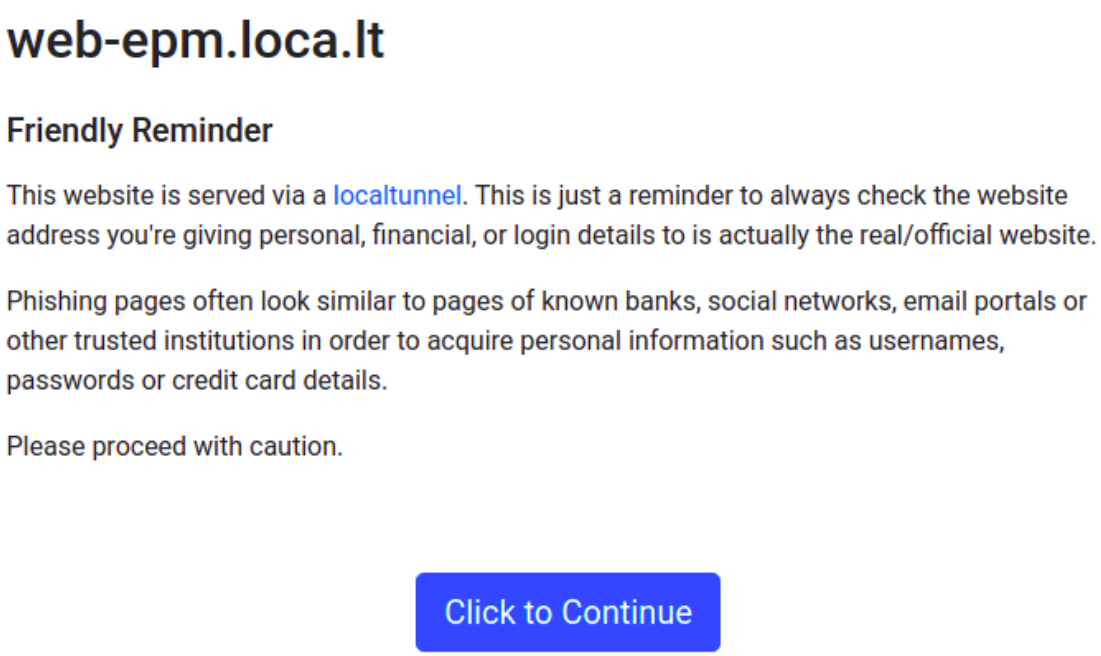


Figure 2.1: Localtunnel Friendly Reminder

And finally, you have to start the back-end services **UserAuth** and **TaskList** with an additional JVM parameter. Use the following command:

```
$ mvn exec:java
↪  -Dsecret=bfhAp4qdm92bDOFIOZLanC66KgCS8cYVxq/KlSVdjhI=
↪  -Dtest-with-lt=true
```

By using the `-Dtest-with-lt=true` parameter, we are enabling CORS for the origin `https://web-epm.loca.lt` and change cookie parameters to HttpOnly; domain=loca.lt; Secure; SameSite=None in order to make it work.

Then, if you navigate to `https://web-epm.loca.lt`, you will see the Task List front-end. This setup will be used later in the book to test our progressive web app from a mobile device.

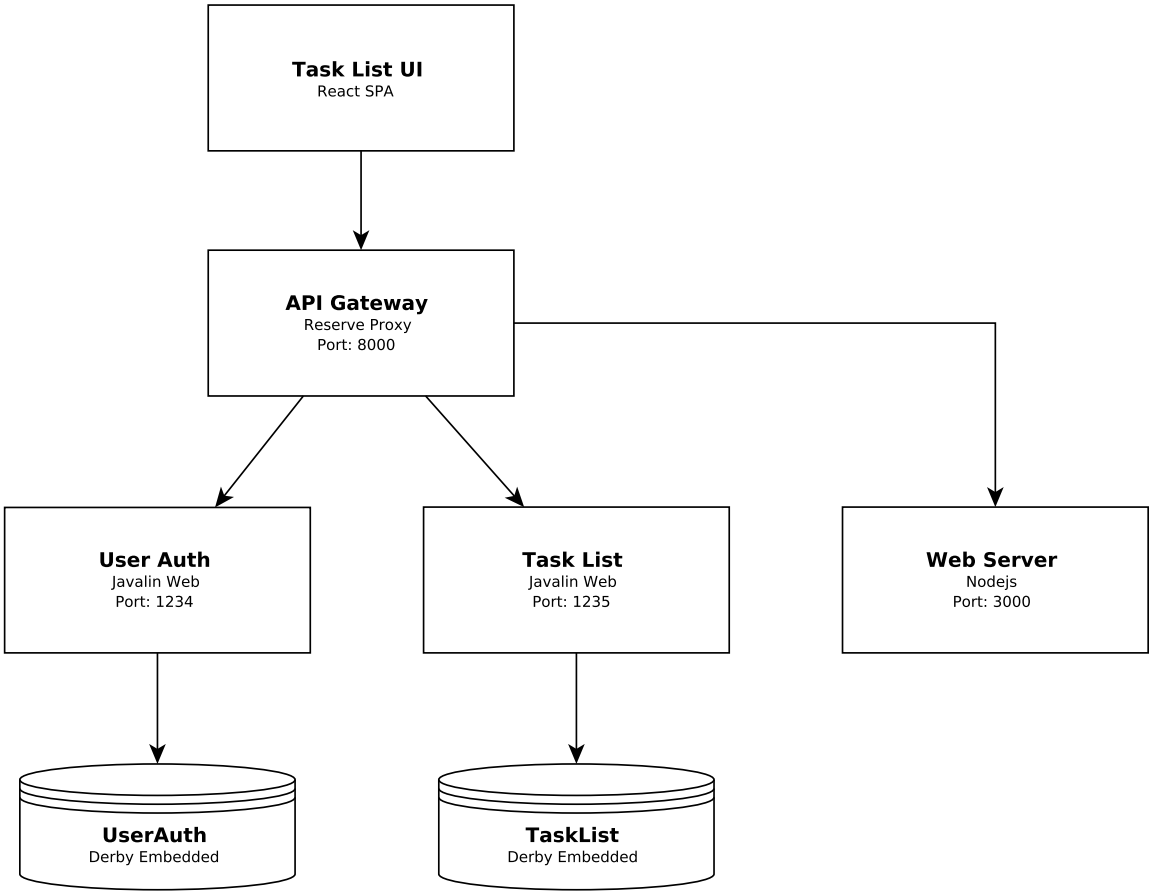


Figure 2.2: Task List - Microservice Architecture

2.2 Task List React Application

The main screenshots of the Task List application are illustrated in figures 2.3 and 2.4. In order to access their task lists a user must type first their username and password (in the login screen 2.3). That will generate a request to the UserAuth back-end service which validates the user's credentials, and if successful it will return an access token. The access token allows the user

to access their tasks consuming the Task List back-end services. The token is stored in an **httpOnly** cookie. Once authenticated, the user can retrieve their tasks. They are presented in a list as shown in figure 2.4 with their expiration date. The expiration date will have different background colors depending how close to the deadline they are. To mark a task as done (or in progress), the end user can click on the checkbox. The second task on figure 2.4 shows how a task looks like when it is marked as completed. You can also delete tasks or add new tasks.

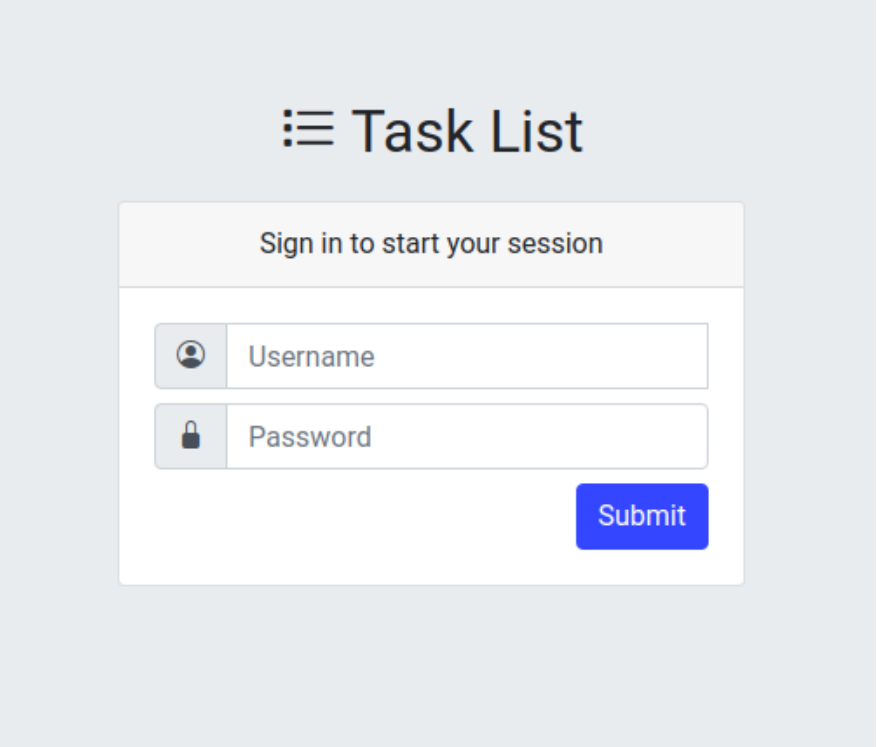


Figure 2.3: Login Page - Task List

Now, if you open the source code of the application, you will see, starting with an uppercase letter, the React components created to build the app. Let's explore each component's responsibility. They are arranged by the following hierarchy:

Listing 2.1: Task List Components Hierarchy

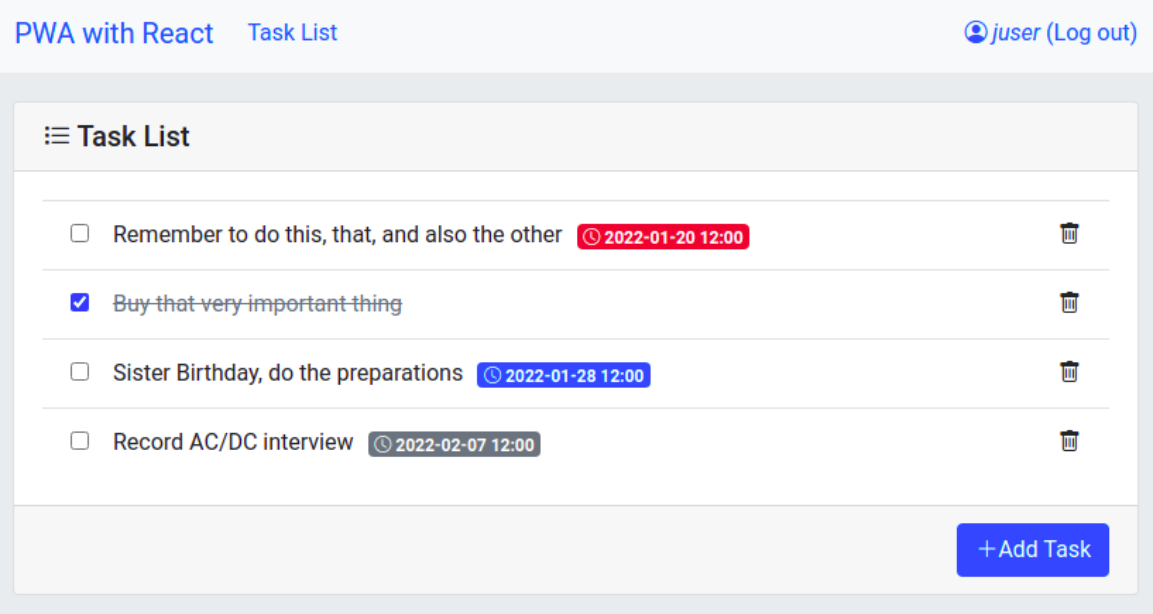
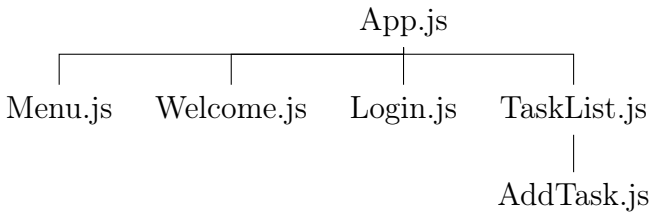


Figure 2.4: juser's Task List



The `Welcome.js` component is responsible for painting the welcome message illustrated in figure 2.5. The `Menu.js` component paints the top header menu, see figure 2.6 (red square). The `TaskList.js` component shown in figure 2.6 (green square) is in charge of painting the list of tasks. It also allows the user to create, delete and mark as done task items. The `AddTask.js` component is responsible for painting the modal window with the form to allow the user to create new tasks (see figure 2.7, yellow square). And finally, the `Login.js` component paints the login form shown on figure 2.3.

Looking at the source code again, in the `src/server` folder, you will see two JavaScript files: `tasks.js` and `users.js`. `tasks.js` encapsulates all the `fetch` requests to the Task List back-end services. While `users.js`, encapsulates all the `fetch` requests to the User Authentication back-end services. Each of the React components described above that requires access

to a back-end service, consumes the functions exposed by `tasks.js` and `users.js`.

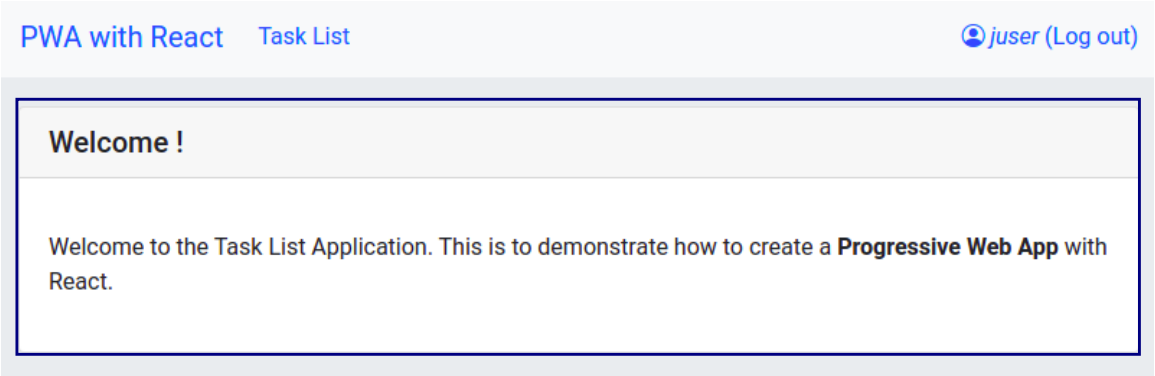


Figure 2.5: Welcome.js component

Having the development environment running and an understanding of the React components that render the Task List application, let's start describing what is needed to transform the application into a progressive web app.

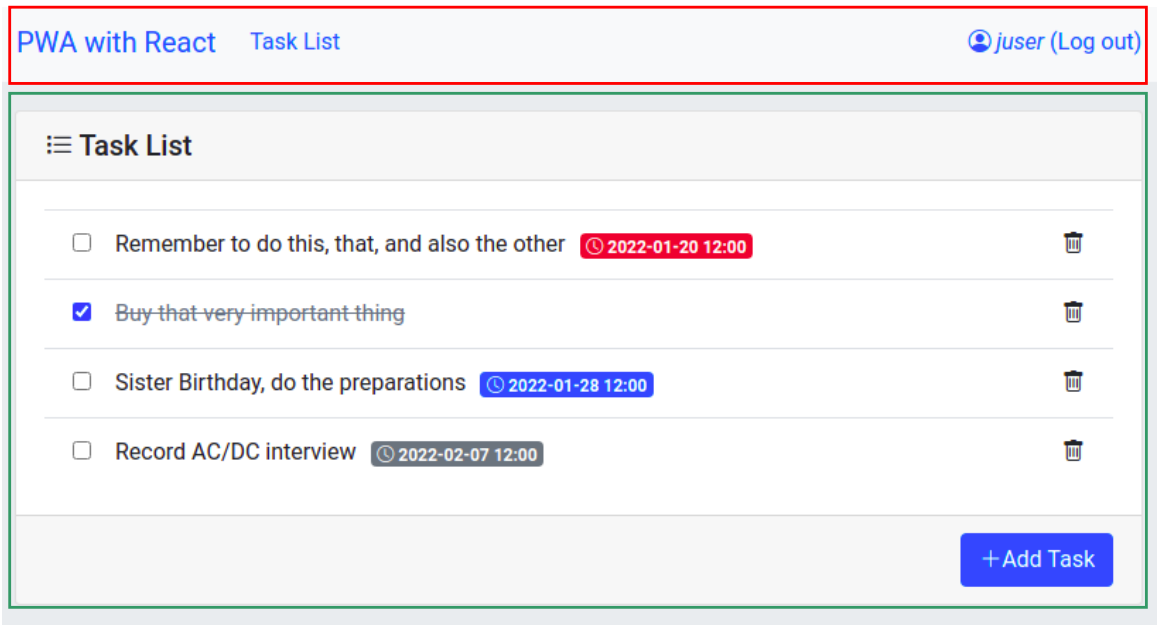


Figure 2.6: TaskList.js

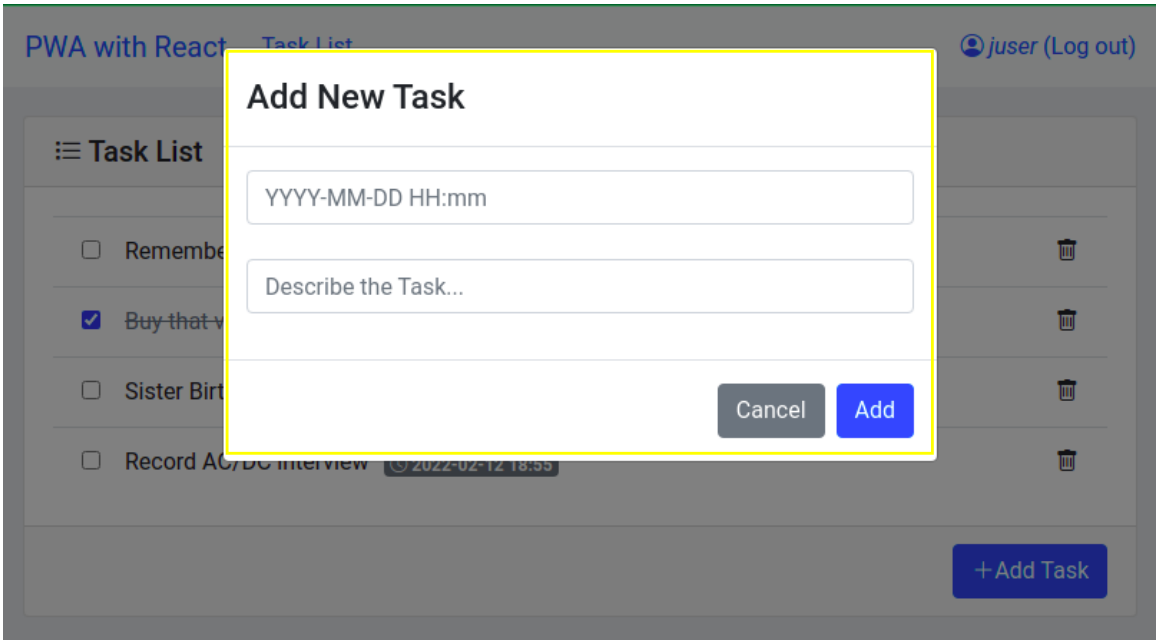


Figure 2.7: AddTask.js

2.3 The Service Worker

You might have used the tool called create-react-app to create and manage React applications. Luckily for us, this tool allows us to create Progressive Web Apps too. To create a new React application with PWA support, you can run the following command:

```
$ npx create-react-app my-app --template cra-template-pwa
```

The Task List application we have described before, was created using that template[1]. However, the PWA capabilities are not enabled yet, we will start with that in a few more paragraphs. When you create a React application using this tool and with this template, the PWA capabilities are not enabled by default, you have to enable them.

Progressive Web Apps capabilities are possible thanks to a technology called **Service Worker**. All the magic you can incorporate to a progressive web app is thanks to it. A Service Worker is a JavaScript program that runs in a background thread. It allows you to implement, in Web Apps, capabilities similar to those offered by native apps. Before service workers existed, Web Workers were introduced. By using Web Workers, the browser allows you to run JavaScript programs in the background without blocking the main browser thread where your web app is running. The Service Worker is a Web Worker with steroids[2]. The most important difference is that they can **intercept** any request that your web app does to a remote service. And with that, you are able to implement web apps with offline support. This ability to intercept, for instance, a network request is done by using an event-driven design. The browser triggers events which your service worker can subscribe to.

Now that you know what a service worker is, let's see where is it inside the Task Application source code. Creating a React app using the create-react-app tool with the cra-template-pwa gives you a React project with three differences. These differences, luckily, gives you much of the work required to make your web app work offline. Out of the three differences, the two most important are the addition of two JavaScript files located in the `src` folder, named: `service-worker.js` and `serviceWorkerRegistration.js`. Cool!

[1] After creation I have upgraded all the workbox packages to latest version, which at the time of writing this book is 6.4.2

[2] There is a great presentation about why Web Workers exists and their difference between Service Workers by Nolan Lawson that I recommend you to see.

service-worker.js is the JavaScript program mentioned above, in which I can subscribe to events and do some work when they are triggered by the browser (you might be thinking). But what about the serviceWorkerRegistration.js ? There is a required step to start using a service worker and that step is called **registration**. In the registration step you tell the browser which is the JavaScript program (the service worker) that you want it to have it run on a background thread to intercept and control your web app. It is important to mention that this registration is against a specific origin (the one that your web app is using). Now, if you open the src/index.js file, you will see the third difference:

```
1  import React from "react";
2  import { createRoot } from "react-dom/client";
3  import "./index.css";
4  import App from "./App";
5  import * as serviceWorkerRegistration from "./serviceWorkerRegistration";
6  import { BrowserRouter } from "react-router-dom";
7
8  const container = document.getElementById("root");
9  const root = createRoot(container);
10
11 root.render(
12   <React.StrictMode>
13     <BrowserRouter>
14       <App />
15     </BrowserRouter>
16   </React.StrictMode>
17 );
18
19 // If you want your app to work offline and load faster, you can change
20 // unregister() to register() below. Note this comes with some pitfalls.
21 // Learn more about service workers: https://cra.link/PWA
22 serviceWorkerRegistration.unregister();
```

As you might have noted, on line 22 above, the registration of the service worker is not done by default, as I was mentioned. If you want it, you have to change that line to call the function register() instead of unregister(). When you change that, the service worker starts the registration process. During this complex process there are state transitions and events are triggered. We will describe this by looking at the most important sections of the service-worker.js and serviceWorkerRegistration.js source code.

Let's have a look at the `register()` function from the `serviceWorkerRegistration.js`:

Listing 2.2: serviceWorkerRegistration.js/register()

```
1  export function register(config) {
2    if (process.env.NODE_ENV === "production" &&
     ↪  "serviceWorker" in navigator) {
3
4      window.addEventListener("load", () => {
5        const swUrl =
         ↪  `${process.env.PUBLIC_URL}/service-worker.js`;
6
7        if (isLocalhost) {
8          checkValidServiceWorker(swUrl, config);
9          navigator.serviceWorker.ready.then(() => {
10           console.log(
11             "This web app is being served cache-first by a
               ↪  service " +
12               "worker. To learn more, visit
                 ↪  https://cra.link/PWA"
13           );
14         });
15       } else {
16         registerValidSW(swUrl, config);
17       }
18     });
19   }
20 }
```

The first important thing to note from the code above on line 2, is that the service worker is registered only in a production build. This means that for testing a progressive web app, you have to run the following command:

```
$ npm run build && serve -s build
```

Note additionally on line 2 the condition `"serviceWorker" in navigator` that checks if the browser you are using supports a service worker. If not, your web app will work without their benefits. This is part of the progressive enhancement we talked about on section 1.1. After that, on line 4, we define a subscription to the `load` event, and inside there, as explained next, we proceed with the registration. The registration of the service worker

might trigger in a different thread the download of your static files in order to cache them. This work might affect the performance of the main thread that renders the web app, that is the reason to perform the registration after (hence inside the `load` event) your web app has been finished loaded[3]. The block of code from lines 7 to 17 has a branch where we are on `http://localhost/` or not. If we are on `http://localhost/` there is a call to the function `checkValidServiceWorker(swUrl, config)`[4] which first will check if the `` `${process.env.PUBLIC_URL}/service-worker.js` `` exists. You might use the `http://localhost/` URL for many other web apps that might not have a service worker, so before registering it, it verifies if it exists. If it exists, it will call the `registerValidSW(swUrl, config)`, in the same way that if you aren't on `http://localhost/` (`else` branch on line 15).

Let's then have a look at the `registerValidSW` function below:

Listing 2.3: serviceWorkerRegistration.js/registerValidSW

```
1  function registerValidSW(swUrl, config) {
2    navigator.serviceWorker
3      .register(swUrl)
4      .then((registration) => {
5        registration.onupdatefound = () => {
6          const installingWorker = registration.installing;
7          if (installingWorker == null) {
8            return;
9          }
10         installingWorker.onstatechange = () => {
11           if (installingWorker.state === "installed") {
12             if (navigator.serviceWorker.controller) {
13               console.log(
14                 "New content is available and will be used
                   ↪  when all " +
15                   "tabs for this page are closed. See
                     ↪  https://cra.link/PWA."
16               );
17
18               // Execute callback
19               if (config && config.onUpdate) {
```

[3] You might want to read this from Google's docs: User's first visit

[4] I have not added here the source code of this function, because it end up calling to `registerValidSW(swUrl, config)`.

```
20                  config.onUpdate(registration);
21                }
22              } else {
23                console.log("Content is cached for offline
                  ↪ use.");
24
25                // Execute callback
26                if (config && config.onSuccess) {
27                  config.onSuccess(registration);
28                }
29              }
30            }
31          };
32        };
33      })
34      .catch((error) => {
35        console.error("Error during service worker
          ↪ registration:", error);
36      });
37  }
```

On line 2 (and 3) above, is where the registration is finally done. The line `navigator.serviceWorker.register(swUrl)` will **download** the `servic` `e-worker.js` file and then is **executed** (in a context without access to the DOM). After that, the service worker starts the life cycle shown on figure 2.8, which I will describe while I explain the source code above.

Note that if the registration is successful (service worker downloaded and executed) it subscribes to the `updatefound` event, on line 5. That event is triggered every time the registration of a service worker enters in the installing state. Since that is exactly what is happening, the function starts its execution. At this point, and before moving to the **installed** state, the service worker is dispatched with the **install** life cycle event (as shown in figure 2.8). This event is the place to perform a **precache** of the static assets of the application. We will go back to this later. On line 6 above, the service worker instance is assigned to the `installingWorker`, and subscribes to the `statechange` event (see line 10). Every time a service worker moves from one state to another the function started at line 11 will be executed. If we are in the installed state (`if` condition on line 11), it will check if we have another service worker active (`if` condition on line 12). The `navigat` `or.serviceWorker.controller` (see here: controller) property gives you an

instance of the service worker that is currently controlling the web app pages (the active one). The **first time** your app register a service worker, it won't be any other service worker activated for the same origin. Any **subsequent** deployment of the application, due to we will be in the case of trying to register a service worker when there is one already active, will require some additional work that we will discuss in the next chapter. For now we will put the focus on first time registration. So, now that we understand this, we know that the condition on line 12 will be false, jumping to the line 23 where `"Content is cached for offline use"` is printed on the console. At this point in this process, if you have the service worker subscribed to the **install** life cycle event (this is done on the `service-worker.js`, as we will see later), whatever you do there, it has finished. And finally, if `config.onSuccess` is set, a call to the callback `onSuccess` is executed. That callback can be used to show to the end user a toast message informing that the application has been cached. At some point later, the service worker is dispatched with the **activate** life cycle event (used for cleaning up the cache, if necessary), and after that the service worker becomes **activated**.

Ok, this was the review of the `serviceWorkerRegistration.js` source code, while I explained how the service worker lice cycle works. Let's now review the source code of the `service-worker.js` below. As it was mentioned it is involved in the process explained above because of their subscription to the **install** and **activate** life cycle events.

Listing 2.4: service-worker.js

```
1  import { clientsClaim } from "workbox-core";
2  import { ExpirationPlugin } from "workbox-expiration";
3  import { precacheAndRoute, createHandlerBoundToURL } from
   ↪  "workbox-precaching";
4  import { registerRoute } from "workbox-routing";
5  import { StaleWhileRevalidate } from "workbox-strategies";
6
7  clientsClaim();
8
9  precacheAndRoute(self.__WB_MANIFEST);
10
11 const fileExtensionRegexp = new RegExp("/[^/?]+\\.[^/]+$");
12 registerRoute(
13   ({ request, url }) => {
14     // If this isn't a navigation, skip.
15     if (request.mode !== "navigate") {
```

```
        return false;
      }

      if (url.pathname.startsWith("/_")) {
        return false;
      }

      if (url.pathname.match(fileExtensionRegexp)) {
        return false;
      }

      return true;
    },
    createHandlerBoundToURL(process.env.PUBLIC_URL +
    ↪  "/index.html")
  );

  registerRoute(
    ({ url }) =>
      url.origin === self.location.origin &&
      ↪  url.pathname.endsWith(".png"),
    new StaleWhileRevalidate({
      cacheName: "images",
      plugins: [
        new ExpirationPlugin({ maxEntries: 50 }),
      ],
    })
  );

  self.addEventListener("message", (event) => {
    if (event.data && event.data.type === "SKIP_WAITING") {
      self.skipWaiting();
    }
  });
```

The first important thing to mention, and you might have noted from looking at the import statements above, is that it uses workbox. Workbox is a set of modules created by Google to help with the implementation of service workers and caching. As mentioned, the service worker is the place to implement native apps capabilities. One of them most important capabilities to implement is the idea that the web app will still work even without

connectivity. In order to do that, we have to cache the web application. By creating the React application with the cra-template-pwa, we have most of this job done, and is what I will explain next. One of the features that the implementation above gives us is to **precache**[5] the static content of the web app. That is: JavaScripts and CSSs files located in the `src` directory plus the `public/index.html`. Additionally, it will precache the JavaScript and CSS files that your React components imports. The thing is that when we create a production build with the command `$npm run build`, webpack takes all these static files and create some specifics files minimized and combined (see these details here). To deal with this the create-react-tool uses the workbox-webpack-plugin. This plugin is integrated in the production build[6] and it will take care of generating the list of webpack-generated assets plus compiling a production version of the service worker, based on the one we have presented on listing 2.4.

Let me repeat that to make it crystal clear, the `service-worker.js` file from listing 2.4, is not the one used in production. A new one will be created by the production build, based on what is specified in the `service-worker.js`. This is the reason why the service worker is only activated in a production build. If we look at line 9 on listing 2.4, the function `precacheAndRoute(self.__WB_MANIFEST)` is the one that precache all the static assets. The variable `self.__WB_MANIFEST` is replaced with the URLs of all the files in the `build/static` folder, plus the file `public/index.html`. Like below:

```
1  precacheAndRoute([
2   {url: '/index.html', revision: '383676' },
3   {url: '/static/css/2.2c85515e.chunk.css', revision: null},
4   {url: '/static/css/main.53a66fab.chunk.css', revision:
     ↪  null},
5   {url: '/static/js/main.26ceb046.chunk.js', revision: null},
6   // ... other entries ...
7  ]);
```

And when the build process creates the production service worker, it will finally looks like this[7]:

[5] precaching is the process of caching content before it is even used. In contrast with dynamic caching where the cache is populated after is requested for the first time

[6] check it here

[7] This is not exactly how the production source code looks like, it is just to help you understand how this process works.

```
self.addEventListener('install', (event) => {
  if (!('caches' in self)) return;
  event.waitUntil(
    caches.open('workbox-precache').then((cache) => {
      return cache.addAll(
        [
          '/index.html',
          '/static/css/2.2c85515e.chunk.css',
          '/static/css/main.53a66fab.chunk.css',
          '/static/js/main.26ceb046.chunk.js',
          // ... other entries ...
        ]
      );
    })
  );
});
```

Note from the code above that the precache is executed when the install event is dispatched, as mentioned before. And to cache content the CacheStorage is used.

As it was mentioned, a service worker can intercept network requests and it may respond to the browser with cached content, content from a remote service (network) or content generated in the service worker. This is called routing in Workbox terms. So, in addition to precache, the `precacheAndRoute` function will **route** any static content request, that the web app perform, to the cache. This is done by adding a listener in the production service worker to the **fetch** event[8]. Something similar to the code below:

```
1  self.addEventListener('fetch', (event) => {
2    event.respondWith(
3      caches.match(event.request).then((response) => {
4        return response || fetch(event.request);
5      })
6    );
7  });
```

The code above will check if the resource requested is in the cache (line 3), and return that if it is there. If not, it will perform a network request using the `fetch` function (*or* condition on line 4). Then, it uses the FetchEvent.respondWith

[8]The fetch event is a functional event, not a life cycle one.

method (line 2) to return the response. The response will be either the one from the cache or the one obtained from the network. The service worker generated by the production build will use the cache-first strategy, meaning that if the content exists in the cache, it will be returned from there. Only when the content is not in the cache, it will be requested through the network. In the next chapter I will explain how to update the cache when a new release of the web app is deployed in production.

On lines 12 and 32, there are two method calls to the same function: `registerRoute`. The one on line 12 will route any navigational request to the `index.html` stored in the cache. And the one on line 32 will register a route to dynamically cache[9] `png` files. This is done due to the two `png` files we have on the public folder (React logos). Those are not managed by webpack, so they are not cached. If you add more images here, if they are `png`s they will be cached. Of course, that method can be customized to cache other types of files. Note that on the second parameter of the `registerRoute` function, we are passing an instance of a class called `StaleWhileRevalidate`. That specifies the **strategy** we want to use to cache the content. With this one, on each request to a png file, the service worker retrieves them from both the cache and the network. Responding with the one from the cache (if available) but updating the cache with the one from the network. There are other strategies you can use if you need it.

The subscription to the message event on line 43 is used on subsequent service worker registration. I will explain that in the next chapter. And finally, on line 7 the `clientsClaim()` function (docs here). I left this one to the end, because I wanted to explain how caching works first. What does this do? When a service worker is activated it won't take control of opened browser tabs, until the next reload. With this method, those open tabs get controlled by the service worker immediately. This method adds the following code to the production service worker:

```
1 self.addEventListener("activate", (function() {
2  return self.clients.claim()
3 }
```

Calling the `self.clients.claim()` method (see official docs here) before the activation of the service worker will throw an error, that is why it is done

[9]Runtime or dynamic caching, as opposed to precaching, refers to cache resources once your application requests them.

on the activate event.

That was the explanation of the source code of the `serviceWorkerRegistration.js` and the `service-worker.js`. With them, we have the static content caching resolved. Let's now jump to show how all of this looks on the browser. Open the starter project on your favorite editor and change the file `src/index.js` to register the service worker, it should looks like below:

```
1  import React from "react";
2  import { createRoot } from "react-dom/client";
3  import "./index.css";
4  import App from "./App";
5  import * as serviceWorkerRegistration from "./serviceWorkerRegistration";
6  import { BrowserRouter } from "react-router-dom";
7
8  const container = document.getElementById("root");
9  const root = createRoot(container);
10
11 root.render(
12   <React.StrictMode>
13     <BrowserRouter>
14       <App />
15     </BrowserRouter>
16   </React.StrictMode>
17 );
18
19 // If you want your app to work offline and load faster, you can change
20 // unregister() to register() below. Note this comes with some pitfalls.
21 // Learn more about service workers: https://cra.link/PWA
22 serviceWorkerRegistration.register();
```

Then, run the following commands to create a production build and start serving it with a static server:

Listing 2.5: Creating a Production Build

```
$ cd react-starter-tasklist
$ npm run build
$ serve -s build/
```

After that, navigate to `http://localhost:3000/` (in Chrome if possible) and open the development tools, pressing the F12 key. Let's see how the

service worker looks like. In the development tools go to the Application tab, and on the left menu press the Service Workers item menu (figure 2.9). Note that it is activated and running. There you have the possibility to unregister it, update it, and more. We will use some of them on the next chapter. Now, in the Application Tab, on the left menu go to the Cache Storage menu item and expand it. As illustrated in figure 2.10, you will see two caches, one for the precached content and the other one called images[10].

That was the explanation about how service workers make it possible to have web apps with offline capabilities. It was explained along with the code generated by the create-react-app with the template cra-template-pwa. In the following section I will explain how to make the Task List application installable. And after that, we will start showing how to implement full offline support for web apps.

[10]If you don't see the images cache, press reload the page. The first time you visit the home page, since the images cache is not precached, the request to it should happen after the service worker has been registered. If you don't like that behavior, you might want to precache the images too.

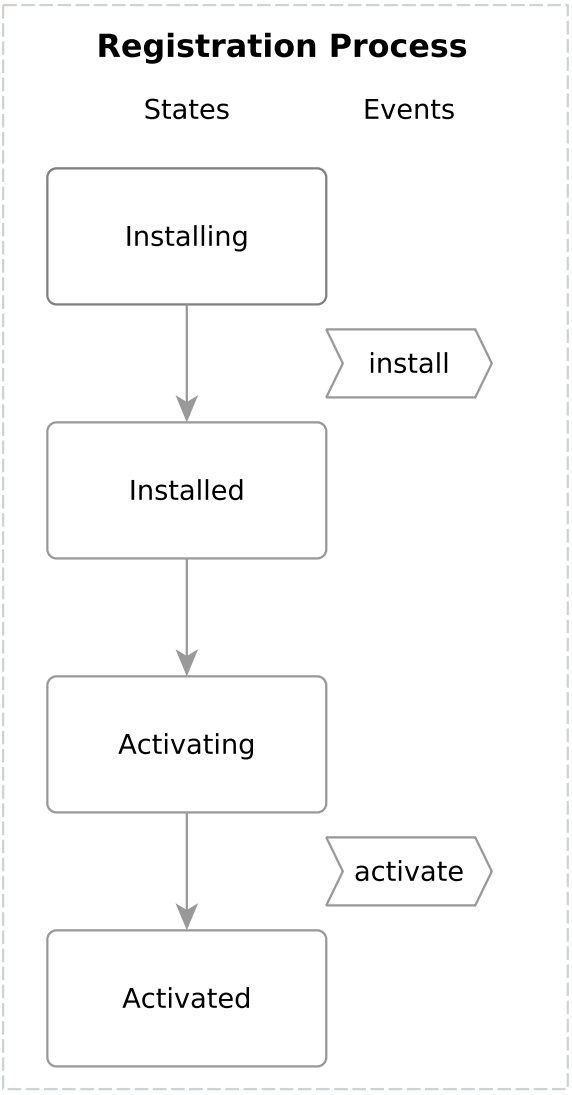


Figure 2.8: Service Worker's Life Cycle

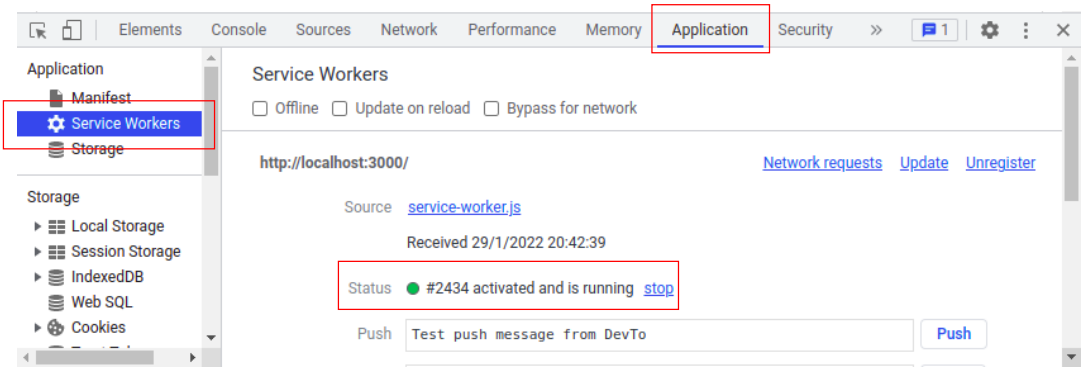


Figure 2.9: Chrome DevTools - Service Worker

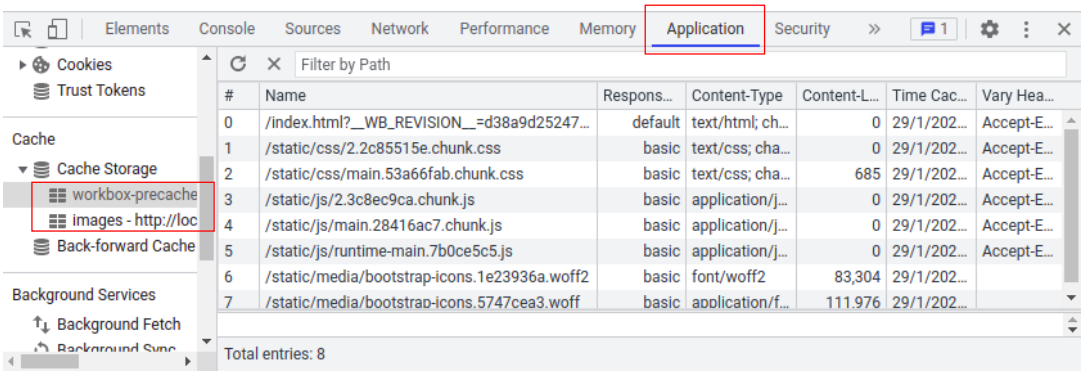


Figure 2.10: Chrome DevTools - Cached content

2.4 Making the Web App Installable

In addition to having a successfully registered and activated service worker, there is one more thing you have to do if you want your web app to be installable. You have to define a manifest file. The manifest file contains information about the web app in a JSON format. It gives to the browser where the web app is running, information about how it should work and looks to the end users once installed or bookmarked.

Like with the service worker, we have the manifest file created thanks to the create-react-app tool, we just need to change it a little bit. If you open the `manifest.json` file located in the `public` directory of the React starter project, you will see the following content:

```
{
  "short_name": "React App",
  "name": "Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    },
    {
      "src": "logo192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "logo512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

Below is the explanation of each property from the file above. There are others described here that might be useful to you.

short_name The name of the application used when there is not enough space for the name.

name The name of the application. This one or the short_name must be specified.

icons Application icons. Different sizes are provided to be used in different contexts. For Chromium browsers, at minimum, you have to provide icons of 192 × 192 and 512 × 512 pixels.

start_url It is the home URL used when the web app is launched.

display This property is used to determine how much of the browser UI is shown to the user when the web app is launched. The available options are: browser (when the full browser window is shown), minimal-ui (minimal browser header, depending on the browser is how much of it is shown), standalone (no browser header is shown, it will display your device header like happens with native apps), and fullscreen (when the app is full-screened).

theme_color It tells the browser which color to use to surrounds the web app. In Android, this color is seen in the header that surround the web app when you tap on the task switcher of your mobile phone.

background_color The background color of the web app before styling is applied. This background color is used on the splash screen (the screen displayed while the web app is loading).

Let's do some changes to the default manifest file to adapt it to the Task List web app and then see how it looks on a mobile device. Proceed with the following changes:

- Change the value of short_name to Task List.
- Change the value of name to Task List Progressive Web App.
- Remove the React logo icons from the `public` folder: logo192.png and logo512.png. And save the new ones, by obtaining them following the links: tasklist192.png y tasklist512.png.
- Change the manifest adding the new icons file names to the icons property.
- Replace the favicon.ico (react logo), with this one: favicon.ico.

- And finally, in the `public/index.html` change the apple-touch-icon link element to point to `%PUBLIC_URL%/tasklist192.png`. This is what iOS uses as a home screen icon.

The new manifest file should looks like the one below:

```
{
  "short_name": "Task List",
  "name": "Task List Progressive Web App",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    },
    {
      "src": "tasklist192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "tasklist512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

Let's now build and serve the React application by running the commands on 2.5. Then set up the system using localtunnel as it was explained on section 2.1.2. Finally, with your mobile phone (it was tested on Android) navigate to: `https://web-epm.loca.lt/`. You will see the "Add Task List to Home screen" message as shown in figure 2.11. Note that it is using one icon (the smaller one and scaled) and the short_name from the manifest. If you tap in the "Add Task List to Home screen", you will be presented with a confirmation dialog as shown in figure 2.12. Note that in that dialog the manifest's name is used to show the web app name. Once you confirm, by

tapping on the install link, the web app gets installed. On your mobile phone, you will see the icon that launches the app like you have for any other native app, as illustrated on figure 2.13. If you launch the app, you will immediately see the splash screen shown on figure 2.14. Once the web app finishes loading, the splash screen disappears and you will see the welcome screen of the Task List application.

With few minor adjustments to the code generated by the create-react-app tool, the web app is now installable. In the next section we will improve the user experience of the web app once there is no connectivity.

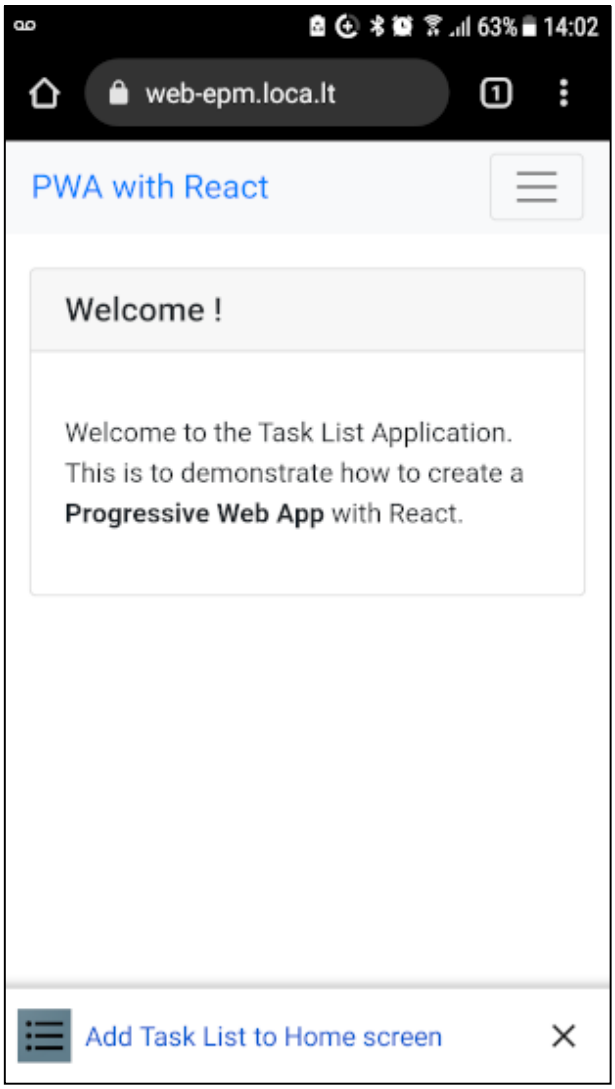


Figure 2.11: Add to Home Screen

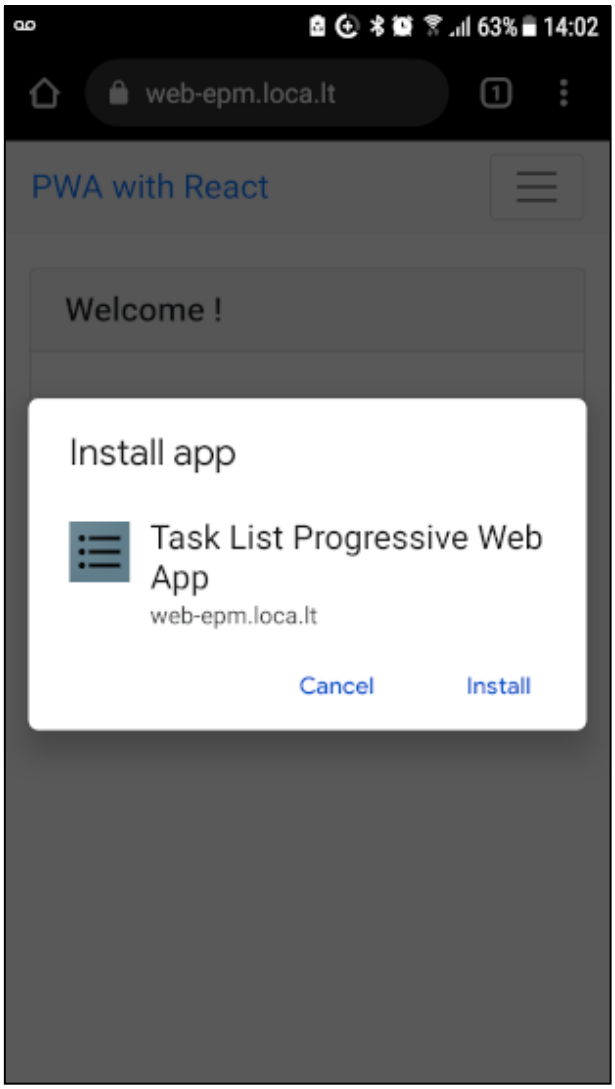


Figure 2.12: Confirm Install

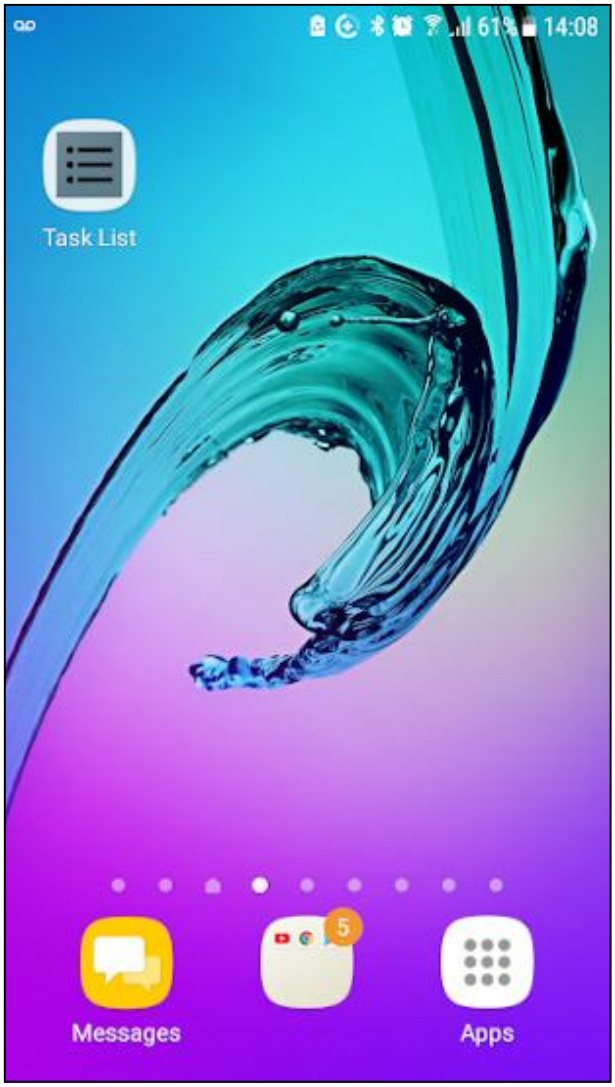


Figure 2.13: Home Icon

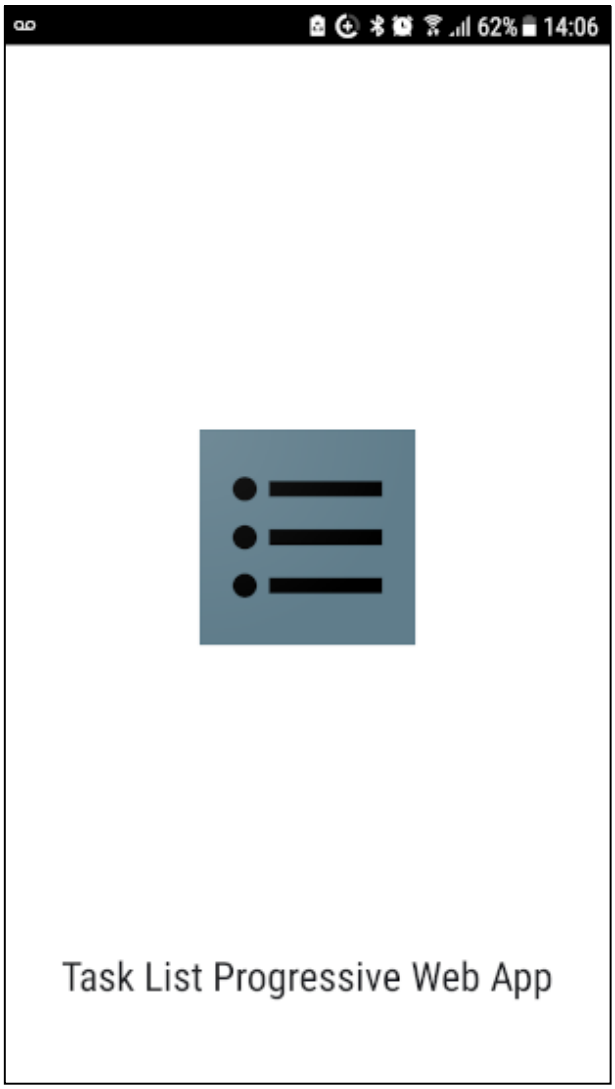


Figure 2.14: Splash Screen

2.5 Supporting Offline

At this point the Task List application is installable, cool!. But if we try to use the application, while we are **offline**, it will crash. This experience is not really nice for a web app that should look like a native app. Let's fix it. We first have to decide if we want to just show a fancy screen to inform the user that the connectivity is lost, or in addition to that, we want to provide some functionality. With very little effort we can provide some of the Task List functionality. We would like our end users to be able to see their tasks while they are offline. To be able to do that we have, somehow, to have the tasks stored in the cache. To implement that, we have to change the service worker to intercept the **fetch** request in order to cache the tasks every time they are retrieved from the remote service. And in the case of the remote service becoming unreachable, (due to connectivity issues) then, we will return the tasks from the cache. In other words, we are going to implement the **Network First** strategy. The network first strategy will first try to get the data from the remote service. If that is successful, the cache is updated and the data is returned to the browser. If not, the data is obtained from the cache and returned to the browser. This is described by google in Network falling back to cache.

Let's see the changes to the `service-worker.js` below to implement what was just explained:

Listing 2.6: Task List Network First Strategy Caching

```
1  self.addEventListener("fetch", (e) => {
2    if (e.request.method === "GET" &&
     ↪  e.request.url.indexOf("/tasks") !== -1) {
3      e.respondWith(
4        fetchWithTimeout(e.request)
5          .then((fetchResponse) => {
6            return caches.open("latest-tasks").then((cache) =>
             ↪  {
7              cache.put(e.request, fetchResponse.clone());
8              return fetchResponse;
9            });
10         })
11         .catch(() => {
12           return caches.match(e.request).then((response) =>
             ↪  {
13             if (response) {
```

```
14                return response;
15              }
16              return new Response('{"tasks":[]}', {
17                headers: { "Content-Type": "application/json"
                    ↪ },
18              });
19            });
20          })
21      );
22    }
23  });
24
25  async function fetchWithTimeout(resource) {
26    const controller = new AbortController();
27    const id = setTimeout(() => controller.abort(), 1000
      ↪ /*timeout*/);
28    const response = await fetch(resource, {
29      signal: controller.signal,
30    });
31    clearTimeout(id);
32    return response;
33  }
```

On line 1 above we are subscribing to the fetch event. That event is triggered every time a fetch request is executed by the browser. The `if` condition on line 2 will ignore all the fetch requests done by the application except the one that retrieves the task list. In case that condition is satisfied (retrieve tasks is executed), the line 3 will be executed. It will first try to fetch the tasks from the network (line 4), using a function defined there to abort the request if it takes longer than one second. If the request is completed within one second, the cache is populated with the response (line 7) and the response is returned to the browser (line 8). Note that from here, the flow continues on the first `.then()` of the `retrieve` function from the file `src/server/tasks.js` (line 35 on the starter project). If there is no connectivity (or a very poor one) the `catch` on line 11 is executed. It will first find the response on the cache (line 12) and if there is a response stored for that request, it is returned (line 14). If there is no entry or the cache is empty, it will return a `Response` with an empty task array (line 16). The `caches.match(...)` returns a `Promise` that resolves to a `Response` with the first matching request or `undefined` if there is no match (that is why I put the `if` statement on line 13). Additionally, it is important to mention that

the `e.respondWith` on line 3 expects as argument a `Response` or a `Promise` that resolves to a `Response`. Make sure you always provide that. This is a home made implementation of the Network First strategy.

However, instead of the above implementation and since we are using workbox, we will take advantage of their utilities. We will use the registerRoute method. We will register the retrieve tasks URL and the handler will be the NetworkFirst strategy class. See the code below:

Listing 2.7: Network First Strategy using Workbox

```
1  const pluginCallbacks = {
2    handlerDidError: async () => {
3      return new Response('{"tasks":[]}', {
4        headers: { "Content-Type": "application/json" },
5      });
6    },
7    cachedResponseWillBeUsed: async ({ cachedResponse }) => {
8      console.log("The response comes from the cache");
9      return cachedResponse;
10   },
11   fetchDidSucceed: async ({ response }) => {
12     console.log("The response comes from the network");
13     return response;
14   },
15   cacheDidUpdate: async () => {
16     console.log("cache was updated");
17   },
18 };
19
20 registerRoute(
21   ({ url }) => url.pathname.indexOf("/tasks") !== -1,
22   new NetworkFirst({
23     cacheName: "latest-tasks",
24     networkTimeoutSeconds: 1,
25     plugins: [pluginCallbacks],
26   })
27 );
```

The first parameter of the `registerRoute` above (line 21), is the condition I wrote on line 2 on listing 2.6. By default, `registerRoute` will only intercept requests with the http GET method. If you need to change that, you can use

the third parameter of the `registerRoute` to specify which http method you are interested in. In the second parameter we can pass a callback function and implement something ourselves or use a strategy class. By using an instance of the `NetworkFirst` strategy class (line 22), we obtain the same functionality we have implemented on listing 2.6. The class receives an optional object as the constructor parameter. In this case we specify how the cache will be named (we use the same name as before, see line 6 on listing 2.6). In addition, by specifying the `networkTimeoutSeconds`, we are telling it to fallback to the cache if the network request takes longer than one second. And finally, the plugins property (line 25) allows us to customize how the strategy behaves under some situations. In our case, the implementation of the method called `handleDidError` (line 2) will be called if an error occurs. The error will occur if there is no connectivity and there is no cache created yet. If this happens, an empty array is returned to the browser (line 3). The other implemented methods are just used for debug purposes to prove the service worker is behaving as expected. We will see that next.

Let's then incorporate the changes from listing 2.7 into our `service-worker.js` file on the starter project. Then, build and serve the React application by running the commands on 2.5. Start the services and the reverse proxy as explained in section 2.1.1 [11]. Open the browser and navigate to `http://localhost:8000/`, you should see the welcome page as shown in figure 2.5. To verify if the registration of the service worker was successful, open the browser's DevTools, and go to the Application tab as shown in figure 2.9. See the text of the status (green) "#NNNN activated and is running". Checked this, we can proceed testing the application. To retrieve the list of tasks you will first need to sign in. Go click the Sign in link, enter the user/password: juser/juser123. Once authenticated, press the Task List link. If you open the browser's DevTools again and go to the Network tab, you will see the tasks request with a small icon (referring to the service worker), as shown in figure 2.15. That means that the request passes through the service worker. This also means that the `latest-tasks` cache was created and populated (and will be updated every time we retrieve the task list from the network). Let's check that, on the browser's DevTools go to the Application tab and on the left menu expand the Cache Storage. You will see the `latest-tasks` cache as shown in figure 2.16. Finally, on the Console Tab, you will see the messages: `The response comes from the network` and `cache was updated`.

[11] Or use the local tunnel set up on section 2.1.2. Just make sure to use the correct URL to navigate the application.

Now that we have checked that the cache was created, we can test the changes we have made to the service worker. The browser's DevTools allows us to emulate different connectivity presets. We will use that to emulate an offline situation. Open the browser's DevTools and go to the Network tab. Find the dropdown, which by default is in "No throttling", and expand it. See figure 2.17. Choose the Offline option and reload the task list. You won't note any difference, however, the list of tasks were retrieved from the cache. If you go to the Console tab on the browser's DevTools, you will now see the message `The response comes from the cache`.

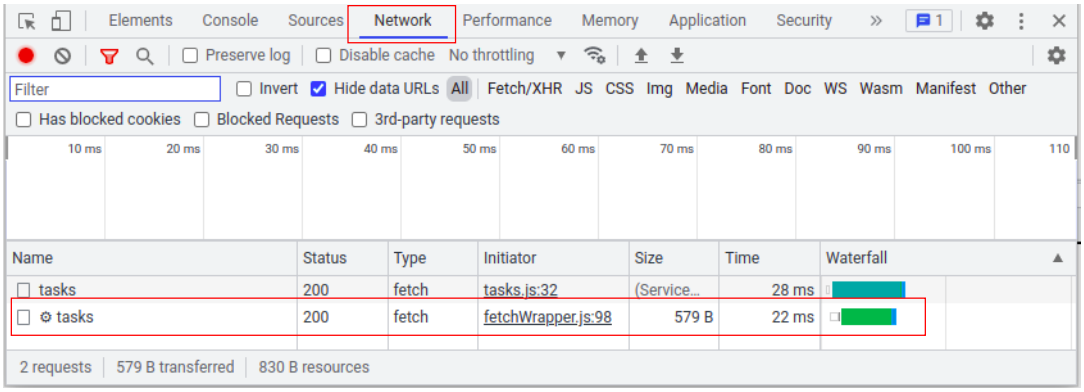


Figure 2.15: Chrome DevTools - Task List from Service Worker

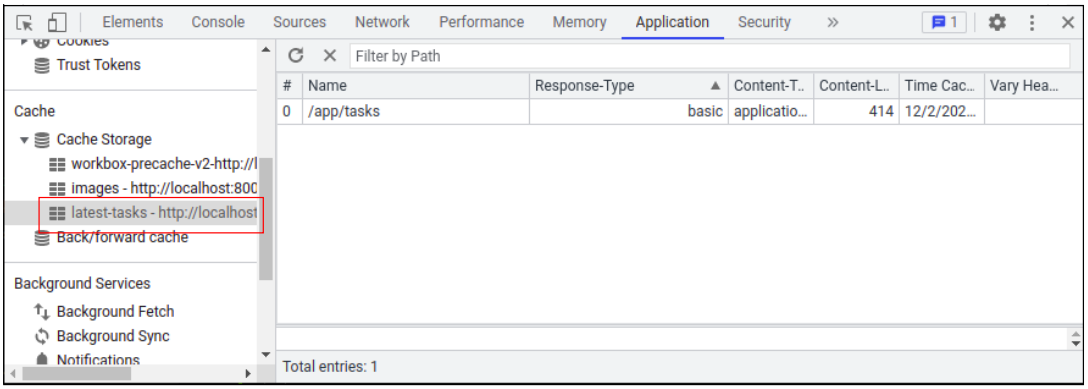


Figure 2.16: Chrome DevTools - latest-tasks cache populated

We have improved how the application manages the retrieval of the task list under connectivity issues. Now, we will do some additional improvements.

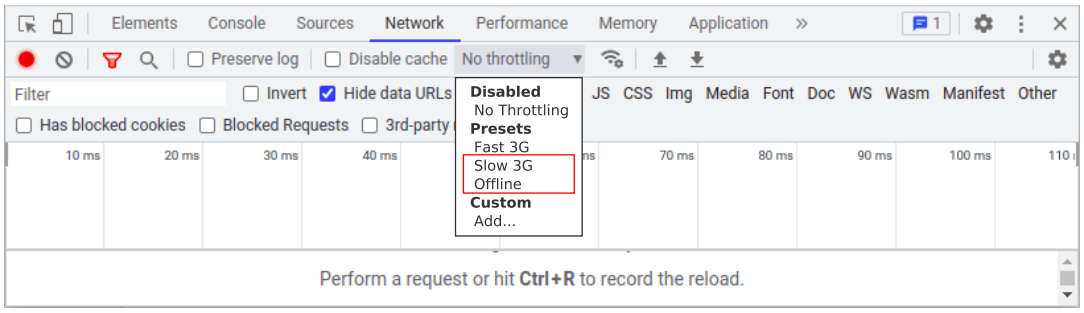


Figure 2.17: Chrome DevTools - Network Presets

First, we will install a package I have created to help us detect when we are offline. With that installed, we will use it to inform the user about connectivity issues and also to disable the actions from the task list grid. The package called react-hook-offline-detector is a custom hook that uses a combination of the navigator.onLine property and a polling request with timeout. The polling is necessary because, as it is described by MDN, the navigator.onLine might give you false positives[12]. Additionally, it might happen that you are online but under a very low or unreliable connectivity (called Lie-fi). In that case the hook will switch the status from online to offline in order to handle that better, instead of having the end user to wait while resources are loaded, which degrades UX.

Let's proceed with the installation of the package in the starter project, by running the following command:

```
$ npm install @enrique.molinari/react-hook-offline-detector
```

Let's also install react-toastify to present a message to the end user when connectivity status changes:

```
$ npm install react-toastify
```

Let's now create a React component that will use react-toastify to present a message to the user when the connectivity is lost. In the `src` folder of the starter project, create a file called `OfflineAlert.js`. Then, copy and paste there the following implementation:

[12]As they describe: You could be getting false positives, such as in cases where the computer is running a virtualization software that has virtual ethernet adapters that are always "connected".

Listing 2.8: Offline Message Component

```
import React, { useEffect } from "react";
import { ToastContainer, toast } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";

export default function OfflineAlert({ offline }) {
  const toastId = React.useRef(null);
  const wasOffline = React.useRef(false);

  useEffect(() => {
    if (offline) {
      wasOffline.current = true;
      toastId.current = toast.warn(
        "There is no Internet connection. Veryfing...",
        {
          position: toast.POSITION.TOP_CENTER,
          autoClose: false,
        }
      );
    } else {
      toast.dismiss(toastId.current);
      if (wasOffline.current)
        toast.info("Internet connection is back!", {
          position: toast.POSITION.TOP_CENTER,
          autoClose: true,
        });
      wasOffline.current = false;
    }

    return () => toast.dismiss(toastId.current);
  }, [offline]);

  return <ToastContainer />;
}
```

Now open the `TaskList.js` component, we are going to do minor changes to incorporate the offline detector hook and the `OfflineAlert` component. Below are highlighted the lines of code you have to add to the component.

Listing 2.9: TaskList.js

```
1  //...
2
3  import OfflineAlert from "./OfflineAlert";
4  import { useOffLineDetector } from
   ↪  "@enrique.molinari/react-hook-offline-detector";
5
6  export default function TasksList(props) {
7    const [tasks, setTasks] = useState({ tasks: [] });
8    const [show, setShow] = useState(false);
9    const [error, setError] = useState(null);
10   const [showAddTask, setShowAddTask] = useState(false);
11   const [taskToDelete, setTaskToDelete] = useState(0);
12   const isOnLine = useOffLineDetector({});
13
14   //...
15   //...
16
17   return (
18     <Container fluid className="mainBody">
19       <OfflineAlert offline={!isOnLine} />
20       <Alert
21         show={error}
22         variant="danger"
23         onClose={() => setError(null)}
24         dismissible="true"
25       >
26         <Alert.Heading>Ops...</Alert.Heading>
27         <p>{error}</p>
28       </Alert>
29
30     //...
31     //...
32
33     </Container>
34   );
35 }
```

The changes are the import statements on lines 3 and 4. A call to the `useOffLineDetector` hook on line 12, and the JSX element on line 19 to render the `OfflineAlert` component. The custom hook will update the `isOnline` state variable each time it detects a change in the connectivity status. When

this happens it will trigger the render of the `TaskList` component, provoking the render of the `OfflineAlert` component. Figure 2.18 illustrates how the toast message is shown when there is no internet connection[13].

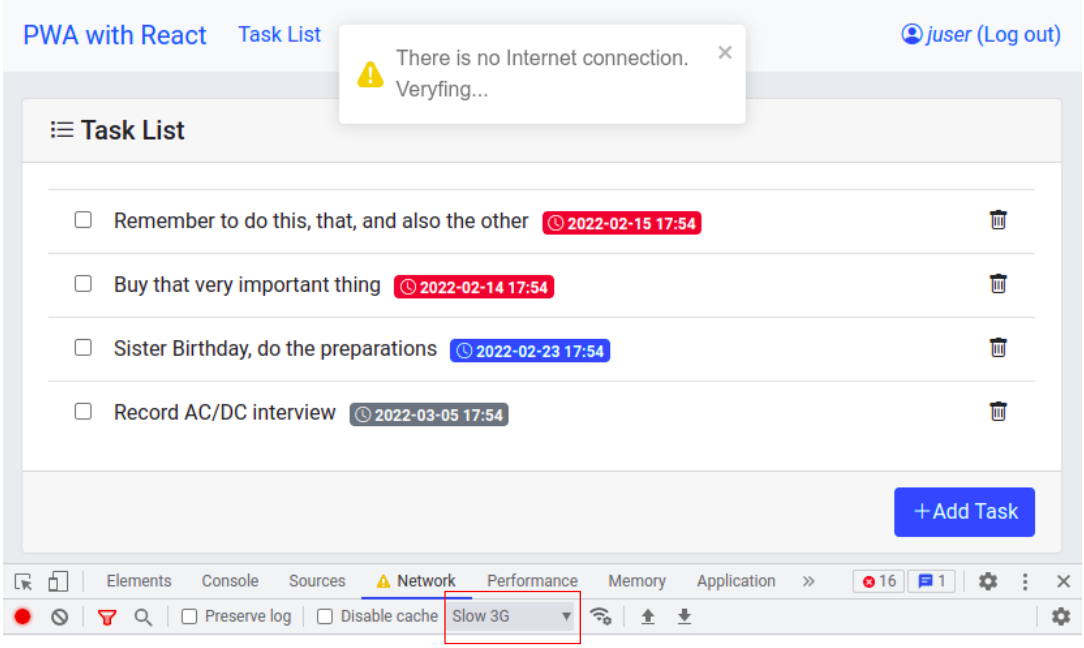


Figure 2.18: Task List without Connectivity

Let's now proceed with a few more additional changes to prevent the users from using the task list functionality when there is no Internet connection. Below are highlighted the lines I have changed to disable the checkboxes, the trash icons and the Add Task button.

Listing 2.10: TaskList: Disabling actions

```
1  // imports here...
2
3  export default function TasksList(props) {
4    // state variables here ...
5
6    const isOnLine = useOffLineDetector({});
7
```

[13]Note that the browser's connectivity preset is on Slow 3G, which means that there is connectivity but is too slow for the default configuration of the offline-detector hook which is one second.

```
8     // handlers and other functions here ...
9
10    return (
11      <Container fluid className="mainBody">
12        <OfflineAlert offline={!isOnLine} />
13
14        //Alert ...
15
16        <Card>
17          <Card.Header as="h5">
18            <i className="bi bi-list-task" /> Task List
19          </Card.Header>
20          <Card.Body>
21            {tasks.tasks.map((t, index) => (
22              <ListGroup key={index} variant="flush">
23                <ListGroup.Item className="border-top">
24                  <Form.Check
25                    type="checkbox"
26                    onChange={(e) => handleDoneOrUnDone(e,
                      ↪  t.done, t.id)}
27                    checked={t.done}
28                    inline={true}
29                    disabled={!isOnLine}
30                  />
31                  {t.done ? <span
                    ↪  className="done">{t.text}</span> :
                    ↪  t.text}
32                  {!t.done && (
33                    <Badge variant={status[t.status]}>
34                      <i className="bi bi-clock"></i>
                        ↪  {t.expirationDate}
35                    </Badge>
36                  )}
37                  {isOnLine && (
38                    <a
39                      title="delete task"
40                      role="button"
41                      onClick={(e) =>
                        ↪  handleDeleteOpenConfirm(t.id)}
42                    >
```

```
                      <i className="float-right bi
                       ↪  bi-trash"></i>
                    </a>
                  )}
                </ListGroup.Item>
              </ListGroup>
            ))}
          </Card.Body>
          <Card.Footer className="text-muted">
            <Button
              variant="primary float-right"
              disabled={!isOnLine}
              onClick={handleAddTask}
            >
              <i className="bi bi-plus-lg" />
              Add Task
            </Button>
          </Card.Footer>
        </Card>

        // AddTask ...

        // Modal ...

      </Container>
    );
}
```

The changes above prevents the users from performing any action that adds, deletes or modifies the state of the web app. Finally, I will disable the login/logout link from the Menu. To do that, I have to somehow provide to the `Menu.js` component the `isOnLine` state variable. To make that possible, since `TaskList.js` and `Menu.js` components are siblings (see the component hierarchy on 2.1), I have to move the use of the `useOffLineDetector` hook to the parent component: `App.js`. And from there, pass the `isOnLine` value using `props`. See the code of the `App.js` component below:

Listing 2.11: TaskList: Disabling actions

```
import React from "react";
import "./App.css";
import TasksList from "./TasksList";
```

```
4  import "bootstrap/dist/css/bootstrap.min.css";
5  import "bootstrap-icons/font/bootstrap-icons.css";
6  import Menu from "./Menu";
7  import Login from "./Login";
8  import Welcome from "./Welcome";
9  import { Route, Routes } from "react-router-dom";
10 import { useOffLineDetector } from
   ↪ "@enrique.molinari/react-hook-offline-detector";
11
12 function App() {
13   const isOnLine = useOffLineDetector({});
14
15   return (
16     <Routes>
17       <Route
18         path={"/"}
19         element={
20           <>
21             <Menu isOnLine={isOnLine} />
22             <Welcome />
23           </>
24         }
25       />
26       <Route
27         path={"/tasklist"}
28         element={
29           <>
30             <Menu isOnLine={isOnLine} />
31             <TasksList isOnLine={isOnLine} />
32           </>
33         }
34       />
35       <Route exact path={"/login"} element={<Login />} />
36     </Routes>
37   );
38 }
39 export default App;
```

You will have to change the `TaskList.js` component to remove the hook and change the reference from `isOnLine` to `props.isOnLine`[14]. And finally,

[14]This is left as an exercise to you. You can see the final version of the component here.

make these changes to the `Menu.js` component:

Listing 2.12: Menu: Disabling login/logout

```
import React from "react";
import Navbar from "react-bootstrap/Navbar";
import Nav from "react-bootstrap/Nav";
import { Link, useNavigate } from "react-router-dom";
import { users as usersService } from "./server/users.js";

export default function Menu(props) {
  const userName = usersService.userName();
  const navigate = useNavigate();

  function handleLogout(e) {
    e.preventDefault();
    usersService.logout().then(() => navigate("/login"));
  }

  return (
    <Navbar bg="light" expand="sm">
      <Navbar.Brand href="#">
        <Link to="/">PWA with React</Link>
      </Navbar.Brand>
      <Navbar.Toggle aria-controls="basic-navbar-nav" />
      <Navbar.Collapse id="basic-navbar-nav">
        <Nav className="mr-auto">
          <Nav.Link href="#">
            <Link to="/tasklist">Task List</Link>
          </Nav.Link>
        </Nav>
        <Nav>
          {!userName && props.isOnLine && <Link
          ↪  to="/login">Sign in</Link>}
          {userName && props.isOnLine && (
            <a href="#task" onClick={handleLogout}>
              <i className="bi bi-person-circle">
              ↪  {userName}</i> (Log out)
            </a>
          )}
          {userName && !props.isOnLine && (
```

```
36              <i className="bi bi-person-circle">
                ↪  {userName}</i>
37            )}
38          </Nav>
39        </Navbar.Collapse>
40      </Navbar>
41    );
42  }
```

With the changes above, we are removing, when we are offline, the link that allows a user to login or logout. Figure 2.19 shows how the list of tasks looks like when the user is offline. Note that features like add a new task or mark a task as done are disabled.

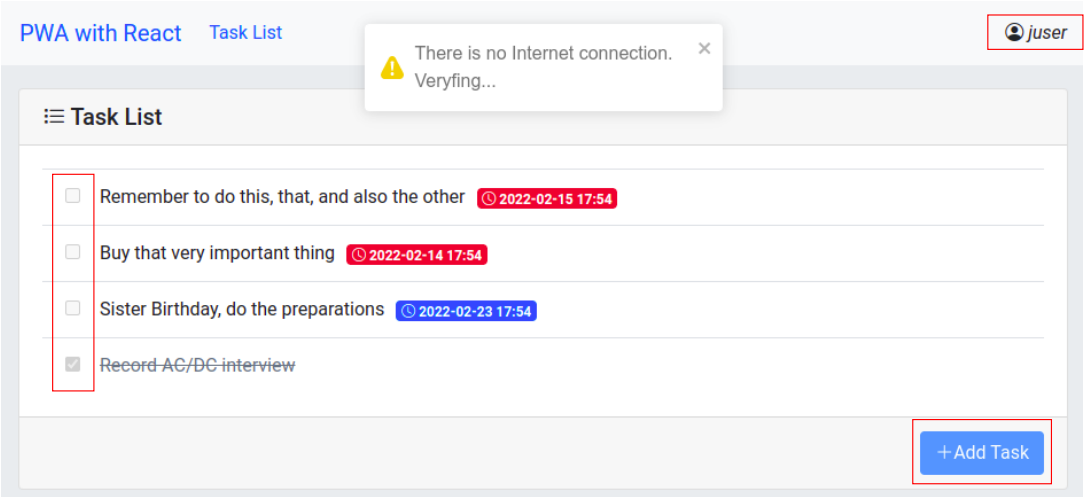


Figure 2.19: Task List Grid - Offline mode

With these changes, we have created a pretty decent progressive web app. The source code can be downloaded from pwa-v1. In the following chapters I will show and implement more capabilities.

Chapter 3

Handling New Releases

Once you have the first release of a progressive web app on the market and installed on your clients, how can you handle a new release? Think about it. You will have all the static assets of the application stored in the browser's cache of your clients and the application works with a cache-first strategy. What should you do to make your application know that there is a new release deployed on the server? Luckily, there is a mechanism implemented in the browsers that check if there is a new version of the service worker on the server (The service worker that, in our case, we registered with the URL `${process.env.PUBLIC_URL}/service-worker.js`. See listing 2.2 at line 5). This check is executed, according to google if any of the following happens:

- A navigation to a page
- A functional event such as push and sync, unless there's been an update check within the previous 24 hours.

There are two very important points to consider with the explanation above. First, this check works comparing the service worker file from the server against the one cached. If there is a difference (a byte-different is enough) an update process will start (we will talk about this in a bit). If both service workers are the same, nothing happens. This means that if you change any file from your application (a css file, or a react component) but there is no change on the service worker file, the application will not be updated on the client's browser. Any new release of your application **must have** a change on the service worker file (at least a variable with a version number that gets incremented on each new release). In our case, as the production service worker is generated by the production build (by running `$npm run build`), we don't need to worry about it. It is covered

for us. You can test this by doing any change on any React component (not in the `service-worker.js`), then create the production build and compare the version of the production service worker (located inside the build folder) with the previous one. You will note that a revision property has changed.

And the second important point is that this check will take place only if the user closes all the browser's tabs and navigates to the application again or reloads the page. If due to the nature of the application you are building, the user keeps the application open without closing or reloading for a long time there is another option we will discuss later in the section 3.2. For now, we will assume the user will close the application after using it (either closing the browser's tab or close the installed app on mobile or desktop).

Having said that, let's move to study in the next section how we take advantage of this check to deploy a new release of a progressive web app on client browsers.

3.1 The Update Process

In the previous chapter, in section 2.3 we said that the first time your app registers a service worker, it won't be any other service worker activated for the same origin. Any subsequent deployment of the application, due to we will be in the case of trying to register a service worker when there is one already active, will require some additional work. Let's discuss here what this additional work means.

Suppose we deploy a new version of the application (it could be a change as simple as a typo in a label). We then create a production build as described in listing 2.5, and then open the browser and navigate to `http://localhost:8000/`. You will note that the change you did on the application won't be there (yet). Press the F12 key to open the browser's DevTools, go to the Application tab and on the left menu, select the Service Workers item. This is illustrated on figure 3.1. Highlighted with a red square in figure 3.1 you will note that there is a service worker #2592 activated and running. That is the one that is currently controlling the application. And there is another service worker #2594 *waiting* to activate. This last one is the new one that has been installed but not yet activated. Figure 3.2 illustrates this situation. On one side you have the current service worker activated (green box), controlling all the clients (browser's tabs) and on the other side, the new service worker installed and waiting to be activated (yellow box). The new service worker

was installed which means that the install event was triggered. As mentioned, by being subscribed to this event the new service worker creates new entries in the cache (with the new updated content). It won't delete anything from the cache at this point as the application is still being served by the first service worker and requires those entries. Old entries will be deleted once the new service worker gets activated. The new service worker will be in the waiting state until all the clients (browser's tabs) that are currently controlled by the first service worker get closed. This mechanism **ensures** that only **one version** of your service worker is running at a time.

At this point, we need a way to handle this situation which requires shutting down the first service worker and activating the new one. First of all, it would be nice to inform the user that there is a new version of the application available. And additionally, give them the option to update it right away or at some point later. Let's now implement this.

When the browser detects that there is a new service worker on the server, it downloads it and the registration process starts again, with the difference that now there is already a service worker activated. So, if you look again at the service worker registration source code presented on listing 2.3, on line 13 we know that there is a new service worker waiting to be activated[1]. This is detected because at line 11 it is checked that there is a service worker just installed (the new one) and on line 12 we check if there is currently a service worker controlling the clients (the current one). So, inside this branch on line 20 there is a call to the `onUpdate` callback. A body for that function can be passed as parameter to the `register()` function, see line 1 on listing 2.2 (the `config` argument). We will use that callback to show a message that informs the user that there is a new version of the application available. We will implement this message using the react-toastify package that is already installed in the starter project, since we use it to inform the user when the internet connection has gone.

The first change we will do is to move the render of the `OfflineAlert` component. Open an editor and remove it from the `TaskList` component and add it to the `App` component, as shown below on line 18:

Listing 3.1: App component

```
1 import React from "react";
```

[1]Note the console log that prints "New content is available and will be used when all tabs for this page are closed"

```
import "./App.css";
import TasksList from "./TasksList";
import "bootstrap/dist/css/bootstrap.min.css";
import "bootstrap-icons/font/bootstrap-icons.css";
import Menu from "./Menu";
import Login from "./Login";
import Welcome from "./Welcome";
import { Route, Routes } from "react-router-dom";
import { useOffLineDetector } from
↪  "@enrique.molinari/react-hook-offline-detector";
import OfflineAlert from "./OfflineAlert";

function App() {
  const isOnLine = useOffLineDetector({});

  return (
    <>
      <OfflineAlert offline={!isOnLine} />
      <Routes>
        <Route
          path={"/"}
          element={
            <>
              <Menu isOnLine={isOnLine} />
              <Welcome />
            </>
          }
        />
        <Route
          path={"/tasklist"}
          element={
            <>
              <Menu isOnLine={isOnLine} />
              <TasksList isOnLine={isOnLine} />
            </>
          }
        />
        <Route exact path={"/login"} element={<Login />} />
      </Routes>
    </>
  );
```

```
42  }
43
44  export default App;
```

With this change, we will see the toast message not only when the task list component is rendered but also when the other components are rendered. Have this in place, let's apply the following changes to the `index.js` file:

Listing 3.2: Adding New Content Available Message

```
1   import React from "react";
2   import { createRoot } from "react-dom/client";
3   import "./index.css";
4   import App from "./App";
5   import * as serviceWorkerRegistration from
    ↪  "./serviceWorkerRegistration";
6   import { BrowserRouter } from "react-router-dom";
7   import "react-toastify/dist/ReactToastify.min.css";
8   import { toast } from "react-toastify";
9   import Button from "react-bootstrap/Button";
10
11  const Msg = (props) => (
12    <div>
13      A new version is available. Click{" "}
14      <Button size="sm" onClick={() =>
        ↪  onAlertToastClick(props.reg)}>
15        here
16      </Button>{" "}
17      to update it now... Or, close this message and the
        ↪  application will be
18      updated the next time you open it.
19    </div>
20  );
21
22  function onNewRelease(registration) {
23    toast(<Msg reg={registration} />, {
24      position: toast.POSITION.BOTTOM_RIGHT,
25      autoClose: false,
26    });
27  }
28
29  function onAlertToastClick(registration) {
```

```
30      console.log("Do something with registration");
31    }
32
33    const container = document.getElementById("root");
34    const root = createRoot(container);
35
36    root.render(
37      <React.StrictMode>
38        <BrowserRouter>
39          <App />
40        </BrowserRouter>
41      </React.StrictMode>
42    );
43
44    serviceWorkerRegistration.register({ onUpdate: onNewRelease
      ↪  });
```

On line 44 above, note how now we are passing an object with the `onUpdate` property to the service worker registration function. With that, when there is an update, the function `onNewRelease` (line 22) will be called (passing an instance of the service worker that is being registered). On that function, we are just showing a toast message to the user using the `Msg` component defined on line 11. This component receives as a prop the instance of the service worker being registered to prepare the code for the update as we will see later. Now, to see this working in a browser, you first need to create a production build and start the server (see listing 2.5). Then navigate to `http://localhost:8000/` and open the browser's DevTools, go to the Application tab and then to the Service Workers item. On the waiting service worker press the "skipWaiting" link, as shown in figure 3.3. That will shutdown the current service worker and activate the new one. Now, close all the browser's tabs and go to your favorite editor to do a small change like removing the `console.log` print on line 30 (from listing 3.2). This is just to trigger an update process to now see the toast message. After that, create a production build and navigate to `http://localhost:8000/` with the browser's DevTools open. You will now see what the figure 3.4 shows.

Now, what is left to do is to implement the "skip waiting" programmatically when the user clicks on the "here" button from the toast message (as figure 3.4 depicts). To do that, we will use the `message` event listener declared on the service worker on listing 2.4 at line 43. Using the postMessage we can communicate the application with the service worker. In this case, we

can pass the "SKIP_WAITING" message and that will discard the current service worker and promote to active the new one. In your editor, open again the `index.js` and add the following highlighted changes:

Listing 3.3: Adding New Content Available Message

```
1  //imports here...
2
3  const Msg = (props) => (
4    <div>
5      A new version is available. Click{" "}
6      <Button size="sm" onClick={() =>
       ↪  onAlertToastClick(props.reg)}>
7        here
8      </Button>{" "}
9      to update it now. Or, close this message and the
       ↪  application will be
10     updated the next time you open it.
11   </div>
12 );
13
14 function onNewRelease(registration) {
15   toast(<Msg reg={registration} />, {
16     position: toast.POSITION.BOTTOM_RIGHT,
17     autoClose: false,
18   });
19 }
20
21 function onAlertToastClick(registration) {
22   registration.waiting.postMessage({ type: "SKIP_WAITING"
     ↪  });
23 }
24
25 navigator.serviceWorker.addEventListener("controllerchange",
   ↪  () => {
26   window.location.reload();
27 });
28
29 const container = document.getElementById("root");
30 const root = createRoot(container);
31
32 root.render(
```

```
33      <React.StrictMode>
34        <BrowserRouter>
35          <App />
36        </BrowserRouter>
37      </React.StrictMode>
38    );
39
40    serviceWorkerRegistration.register({ onUpdate: onNewRelease
      ↪  });
```

From the code above, when the user click on the "here" button from the toast message (line 6) to update the version of the application, we call the `onAlertToastClick(registration)` function which finally post the `{"type": "SKIP\_WAITING"}` message to the service worker in the waiting state. That message is received by the callback function on listing 2.4 at line 43 which ends up calling the `self.skipWaiting()` which shuts down the current service worker and activates the new one.

Finally, starting at line 25 above, we are subscribing to the `controllerchange` event, which will be triggered when the new service worker starts controlling the clients (browser's tabs). And when that happens a reload is executed. This reload is necessary because the version of the application we are seeing is the previous one as it was controlled and served with the previous version of the service worker. By reloading, the new service worker will now serve the application showing the new content.

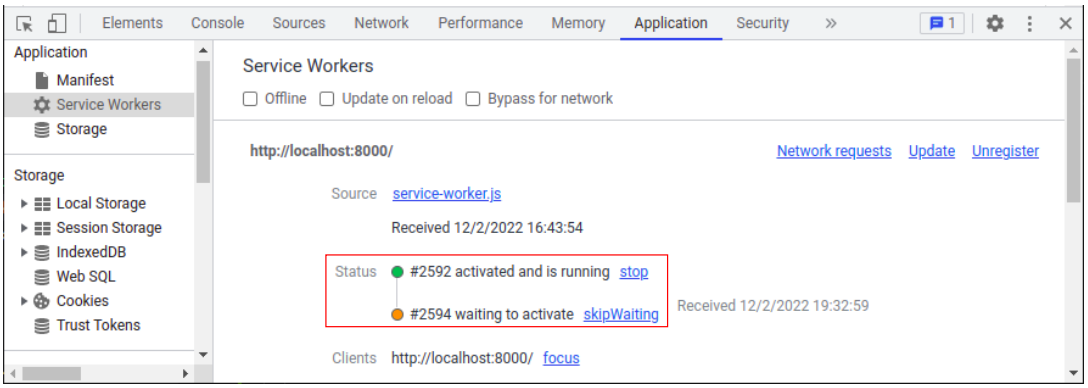


Figure 3.1: Chrome DevTools - New service Worker Waiting

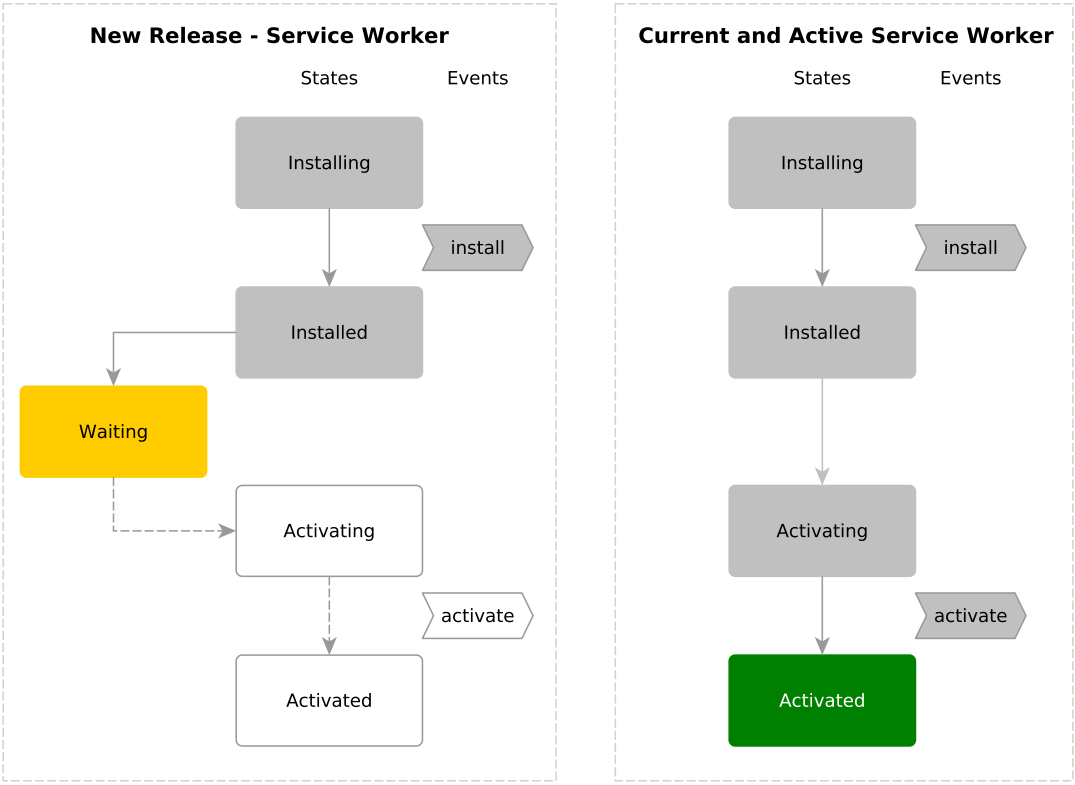


Figure 3.2: Service Worker Waiting

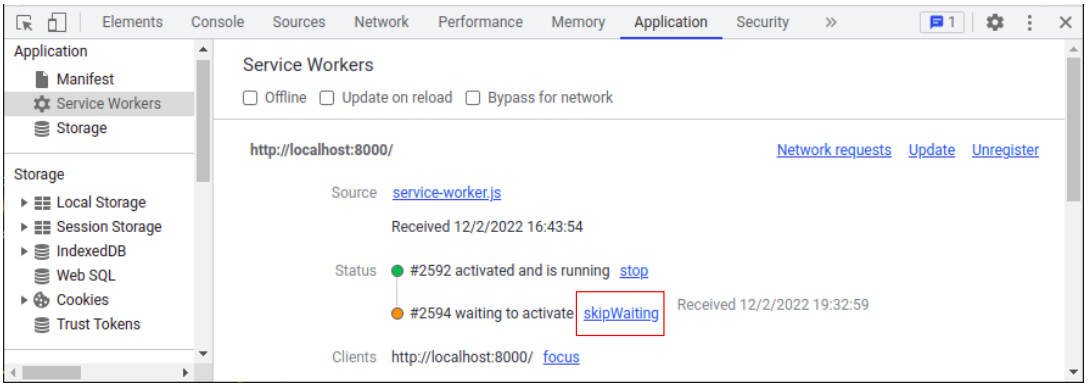


Figure 3.3: Chrome DevTools - Skip Waiting

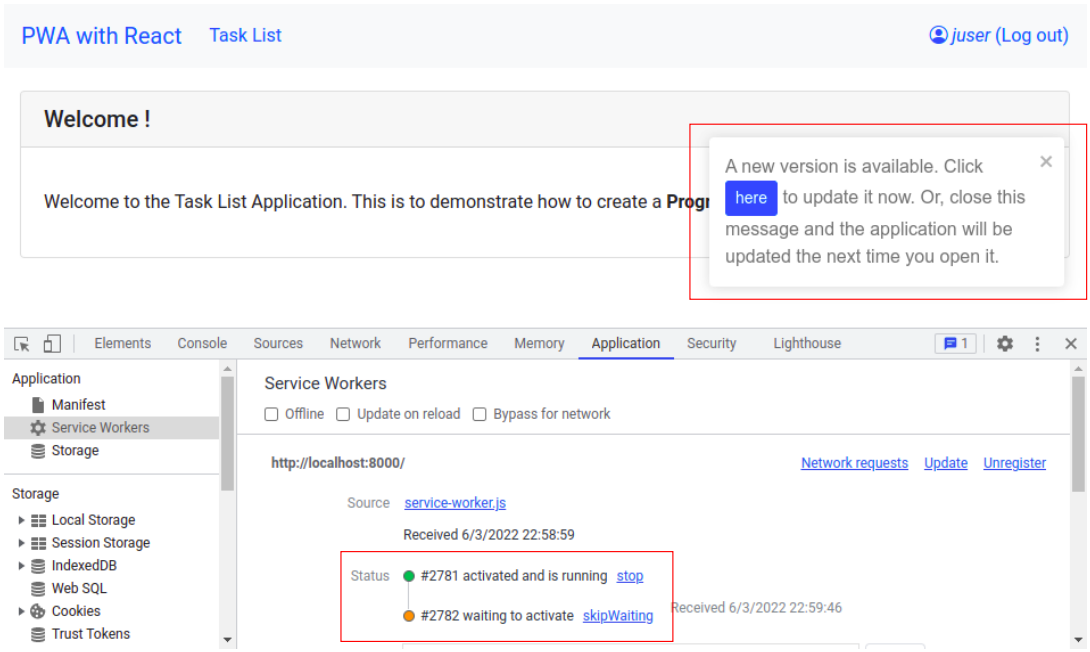


Figure 3.4: Task List Application - New content available

3.2 Manual Update

If due to the nature of the application, the users will have it open for a long time, there is a way to programmatically perform the check to see if there is a new service worker on the server. The ServiceWorkerRegistration API provides an update method to do exactly that. This method, when called, will check if there is a new service worker on the server and if there is a byte-different to the one on the cache it will trigger the update process described in the previous section. So, an alternative that I recommend is to set up an interval at registration to perform this check every 1 or 2 hours (or what you consider is best for you). Open the `serviceWorkerRegistration.js` file on an editor and on the `registerValidSW` function add the following highlighted lines:

Listing 3.4: Adding New Content Available Message

```
1  function registerValidSW(swUrl, config) {
2    navigator.serviceWorker
3      .register(swUrl)
4      .then((registration) => {
5        setInterval(() => {
6          registration.update();
7        }, 1000 * 60 * 60 * 2); // each 2 hs
8
9        registration.onupdatefound = () => {
10         const installingWorker = registration.installing;
11         if (installingWorker == null) {
12           return;
13         }
14
15         // source code continue here...
16
17       }
18     }
19 }
```

To test this, first of all, change the interval time from two hours to 10 seconds (line 7 above). Create a production build and navigate to `http://localhost:8000/`. Click the "here" button on the toast message. Now, without closing the browser's tab, stop the server. Go to the editor, make any change to the application and then create the production build again (starting the server one more time). You will note that now the toast message will

appear without refreshing.

The full source code is available pwa-v1-update.

Chapter 4

Incorporating Background Sync

In this chapter we are going to make the Task List application to work fully offline. All the features described in section 2.2 will work without connectivity. To make this work, we are going to use the browser's database called IndexedDB. In the next section I will describe in detail what is and how to use IndexedDB, but for now, it is just important to understand that it will be used to persist the tasks. Instead of consuming Task List's remote services, we are going to implement the persistence in this browser's database. And to keep the tasks synchronized with the remote storage exposed by Task List's services we will take advantage of the Background Sync capability that the service worker offers.

4.1 The Background Sync Capability

The Background Sync capability (at the time of writing this book supported by Chrome, Edge and Opera) is pretty easy to use. From the web app you can trigger a sync by registering an event like below:

Listing 4.1: Registering the Background Sync

```
1 navigator.serviceWorker.ready.then((reg) => {
2  reg.sync.register("sync-queued-data");
3 });
```

In the code above, we are registering on the current service worker (obtained by calling `navigator.serviceWorker.ready`) the sync event with the tag `"sync-queued-data"`. And then, in the service worker file, you can listen to a sync event like below:

Listing 4.2: Listening to the Background Sync event

```
1  self.addEventListener("sync", (event) => {
2    event.waitUntil(doSync(event));
3  });
4
5  async function doSync(event) {
6    if (event.tag === "sync-queued-data") {
7      //do something here...
8    }
9  }
```

Note on line 1 above that we are listening to the sync event. On line 6, I query the tag to determine which is the action I need to process. Tags are used to identify the action to be executed on each event. In a web app you might need to register a sync event for syncing data and another to send an email for instance. Tags are used to identify them. The `doSync` function must return a Promise, as this is what the event.waitUntil requires. If the promise resolves the service worker will mark the event as completed. If not, it will re-try some minutes later.

Having said that, figure 4.1 illustrates how the web app will work. On the left we have the browser's thread that is running the web app, storing tasks in the IndexedDB database and registering sync events. On the right we have the service worker listening to and processing sync events, obtaining the data from IndexedDB and sending it to the remote service.

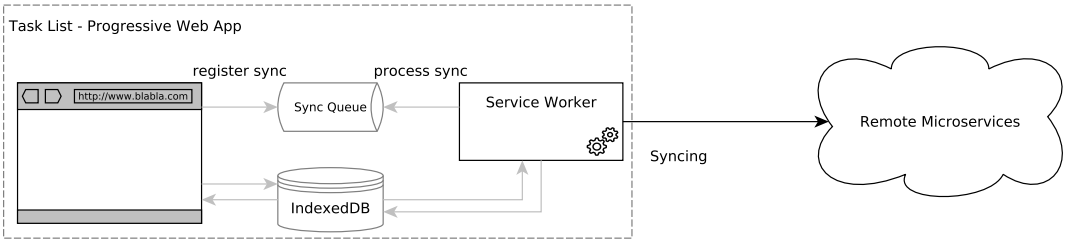


Figure 4.1: Using Background Sync and IndexedDB

4.2 Using IndexedDB

IndexedDB is a NoSQL database that falls into the key-value category. It supports **transactions** which is a required feature for the implementation

we are going to do in this chapter, as you will see later.

An IndexedDB database has a name (which is associated with the domain/origin) and a version. When the database is created the first time the version is one. The version determines how the current structure of the database is.

A database has one or more object stores. The object store is the storage mechanism of an IndexedDB database. An object store has a list of records where each record consists of a key and a value (like any other key-value NoSQL database). Let's continue showing some code examples to explain how IndexedDB works.

The example below, listing 4.3, shows how to open a database, create an object store and insert some data.

Listing 4.3: IndexedDB - Event Based API

```
const request = indexedDB.open("peopledb");
let db;

request.onupgradeneeded = function () {
  const db = request.result;
  const store = db.createObjectStore("people", { keyPath:
  ↪  "id" });
  store.createIndex("by_name", "name");
  store.createIndex("id", "id", { unique: true });
};

request.onsuccess = function () {
  db = request.result;

  const tx = db.transaction("people", "readwrite");
  const store = tx.objectStore("people");

  store.put({ name: "Enrique", surname: "Molinari", id: 1
  ↪  });
  store.put({ name: "Jorge", surname: "Marcos", id: 2 });
  store.put({ name: "Nicolas", surname: "Armein", id: 3 });
  store.put({ name: "Josefina", surname: "Alliani", id: 4
  ↪  });

  tx.oncomplete = function () {
```

```
23        console.log("done!");
24      };
25    };
```

On line 1 above, we are opening the database called peopledb. The first time, if a database with that name does not exist, it will be created. From looking at the code above, you might have guessed that the IndexedDB API is **event based**, meaning that we have to register callbacks on the events we are interested in. On line 4, we are registering the callback for the `onupgr⌋ adeneeded` event. This event is triggered when you open a database with a new database version (the second parameter of the `indexedDB.open`). Every time you need to change the structure of your database (adding/changing indexes or adding/changing object stores), you have to specify a new version and that will trigger the `onupgradeneeded` event. In the example above, on line 1, the version parameter is not specified and when that is the case one is the default value. Of course, the very first time you open (and gets created) a database the `onupgradeneeded` event is triggered. On line 6 above, I'm creating an object store called people and defining the `id` key as the primary key. On lines 7 and 8, I'm defining two indexes, the first one to search people by their name and the second one to search people by their primary key.

The `onsuccess` event, on line 11, is triggered after successfully opening the database. On line 14 I'm opening a transaction. The first parameter indicates that the transaction will expand over the people object store and the second parameter that it will be for reading and writing data. On line 15, I get a reference to the people object store to be used on lines 17, 18, 19 and 20 to create new entries. Note that a JavaScript object is passed to the `store.put` method. That JavaScript object must include the key defined as the primary key (see `keyPath` on line 6). The primary key is the only mandatory property that must be included. Finally, on line 22, I'm registering the `tx.oncomplete` callback that will be triggered once the transaction completes.

Figure 4.2 shows the database structure we created with the example above in the Chrome development tool. Note that it shows the origin, the version and how many object stores it has. It can also be deleted from the *Delete database* button. Figure 4.3 shows the data stored in the people object store. Note also that hanging from the object store we have the two indexes created. The indexes are special object stores created to get access to the data by a specific key. See figure 4.4 that illustrates how the by_name index looks like. Note that another column keyPath: name is added, which is used every time there is a retrieval by name.

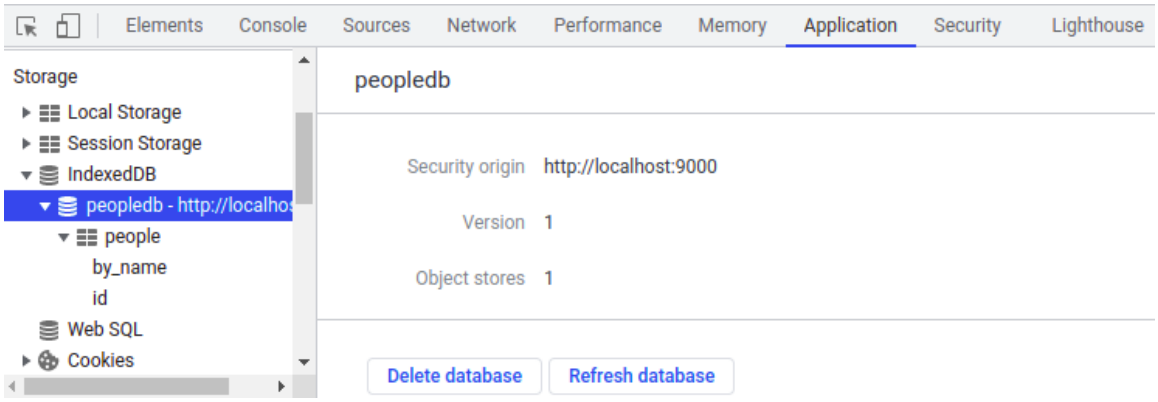


Figure 4.2: IndexedDB - Databases

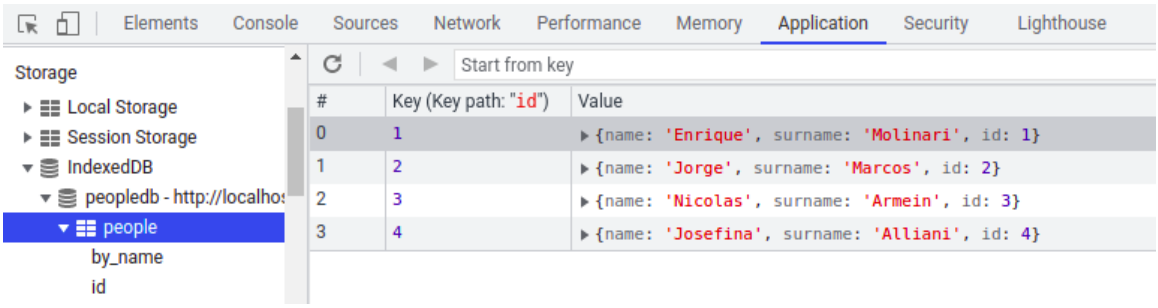


Figure 4.3: IndexedDB - Object Store People

Instead of using the IndexedDB API directly, we are going to use a very thin wrapper of it that was developed by Jake Archibald called idb. It is a promise based API, which simplifies the code that needs to be written to interact with IndexedDB. Now, to introduce idb, I'm going to show a couple of examples to help you understand the code that we will write in the next section to provide full offline support for the Task List application.

Listing 4.4: idb - Open the Database

```
1 const DB_NAME = "idb1";
2 const STORE_NAME = "idb_store";
3 const DB_VERSION = 1;
4
5 const db = await openDB(DB_NAME, DB_VERSION, {
6   upgrade(db, oldVersion, newVersion, transaction) {
```

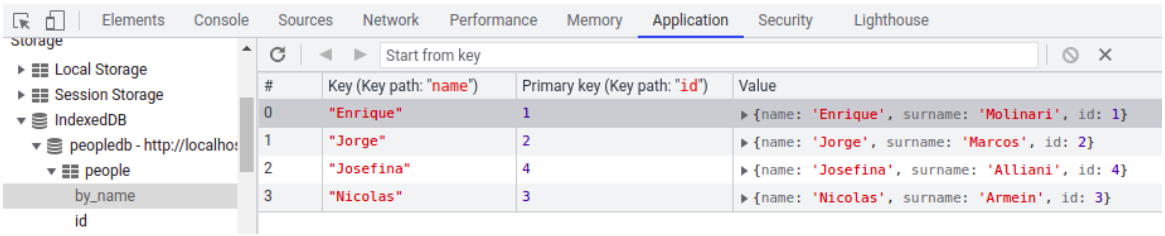


Figure 4.4: IndexedDB - by_name Index

```
7       if (!db.objectStoreNames.contains(STORE_NAME)) {
8         let dbStore = db.createObjectStore(STORE_NAME, {
9           keyPath: "id",
10            //uncomment the line below if you want auto
              ↪  increment primary keys
11            //autoIncrement: true,
12        });
13        dbStore.createIndex("id", "id");
14      }
15    },
16  });
```

In listing 4.4 we use the `openDB` method to create a connection to the IndexedDB database. This method returns a promise that resolves to an improved IDBDatabase. The `openDB` method receives as parameter the database's name, the version and then an object with the `upgrade` function. The `upgrade` function is called if the version passed as the second parameter differs from the current version of the created database. This is equivalent to the `onupgradeneeded` event function we presented before. We use this callback function to create the object stores and indexes.

To add objects into an object store we can proceed in the following way as illustrated on listing 4.5. If you try to add an object with a key that already exists you will get an error.

Listing 4.5: idb - Add an object

```
1  db.add(STORE_NAME, { id: 1, prop1: 1, prop2: 2 });
```

In case you need to update an object that is already stored, use the `put` method as illustrated on listing 4.6. If the key does not exist in the object store, the `put` method will incorporate the object into that object store.

Listing 4.6: idb - Update an object

```
1 db.put(STORE_NAME, { id: 1, prop1: 130 });
```

See the following example, listing 4.7, to understand how transactions works in IndexedDB/idb.

Listing 4.7: idb - Transactions

```
1 const tx = db.transaction(STORE_NAME, "readwrite");
2 const store = tx.objectStore(STORE_NAME);
3 await store.add({ id: 16, prop1: 1, prop2: 2 });
4 await store.add({ id: 19, prop1: 1, prop2: 2 });
5 await tx.done;
6 // or which is the same:
7 //tx.done.then(() => {
8 // console.log("tx commited successfully");
9 //});
```

On line 1 above, the first parameter of the `transaction` method indicates in which object store the transaction will apply. And with the second parameter you indicate which kind of transaction you are going to execute. In this case is a `"readwrite"` transaction. `"readonly"` transactions are valid too. Finally, the `done` method, on line 5 above, resolves when the transaction completes successfully, otherwise rejects with an error.

On listing 4.8, I'm opening a database connection that creates two object stores: `STORE_NAME_1` and `STORE_NAME_2`. Then, on line 23, I'm opening a transaction. Note that now as the first argument of the `transaction` method I'm passing an array with both object store's names. That indicates that the transaction will expand both object stores. After that I'm adding objects to both object stores and finally finishing with the `done` method as in the previous listing.

Listing 4.8: idb - Transactions with Two Object Stores

```
1 const DB_NAME = "idb2";
2 const STORE_NAME_1 = "idb_store_1";
3 const STORE_NAME_2 = "idb_store_2";
4 const VERSION = 1;
5
6 const db = await openDB(DB_NAME, VERSION, {
```

```
    upgrade(db, oldVersion, newVersion, transaction) {
      if (!db.objectStoreNames.contains(STORE_NAME_1)) {
        let dbStore = db.createObjectStore(STORE_NAME_1, {
          keyPath: "id",
        });
        dbStore.createIndex("id", "id");
      }
      if (!db.objectStoreNames.contains(STORE_NAME_2)) {
        let dbStore = db.createObjectStore(STORE_NAME_2, {
          keyPath: "id",
        });
        dbStore.createIndex("id", "id");
      }
    },
});

const tx = db.transaction([STORE_NAME_1, STORE_NAME_2],
↪  "readwrite");
const store = tx.objectStore(STORE_NAME_1);
await store.add({ id: 18, prop1: 1, prop2: 2 });
await store.add({ id: 10, prop1: 1, prop2: 2 });

const store2 = tx.objectStore(STORE_NAME_2);
await store2.add({ id: 2, prop1: 11, prop2: 21 });
await store2.add({ id: 22, prop1: 12, prop2: 22 });
tx.done.then(() => {
  console.log("tx commited successfully");
});
```

Transactions across two object stores is something we are going to use in the next section when we explain the changes applied to the Task List application. Next, I'm going to show different ways of retrieving objects.

Listing 4.9: idb - Retrieving Objects

```
//retrieves the object with key == 22
await db.get(STORE_NAME, 22);

//retrieves the key with key == 22
await db.getKey(STORE_NAME, 22)

```

```
//retrieves an array with all objects from STORE_NAME object store
await db.getAll(STORE_NAME)

//retrieves an array with all keys from STORE_NAME object store
await db.getAllKeys(STORE_NAME)

//retrieves the object with key == 22 from the "id" index
await db.getFromIndex(STORE_NAME, "id", 22)

//retrieves an array with all objects from the "id" index
await db.getAllFromIndex(STORE_NAME, "id")
```

Finally, see below how to delete an object and clear an entire object store.

Listing 4.10: idb - Delete and Clear

```
//delete the object with key == 22
await db.delete(STORE_NAME, 22);

//clears the entire object store
await db.clear(STORE_NAME);
```

4.3 Adding IndexedDB and Background Sync to Task List

In this section, we will describe step by step the changes we are going to make to the Task List application to transform it into a fully offline first Progressive Web App. The repository from the previous chapter pwa-v1-update will be our starting point.

Figure 4.5 illustrates how we are going to deal with the IndexedDB and the Background Sync capability. As the figure describes, we are going to use two object stores. One for the task items called task-store and another one that we will use as a *queue* called queue-store. The queue-store will contain the data required to perform the replication to the back-end service. Every time we add a new task or we mark a task as done (or in progress) or we delete a task, we push the change into the queue-store to indicate the operation that needs to be replicated to the back-end service. That is why it is important to execute these operations in a transaction. We need to

be consistent, if we change somehow the task-store, we have to also push that change into the queue-store. And finally, if the transaction succeeds we register the Background Sync event. The skeleton code in the listing 4.11 shows how this will be implemented. On line 2 we open a transaction that expands the two object stores. Then, we do the operation requested plus pushing the change into the queue-store and finally, on line 6, if the transaction succeeds we register the Background Sync event. The service worker will then proceed with the replication if there is connectivity.

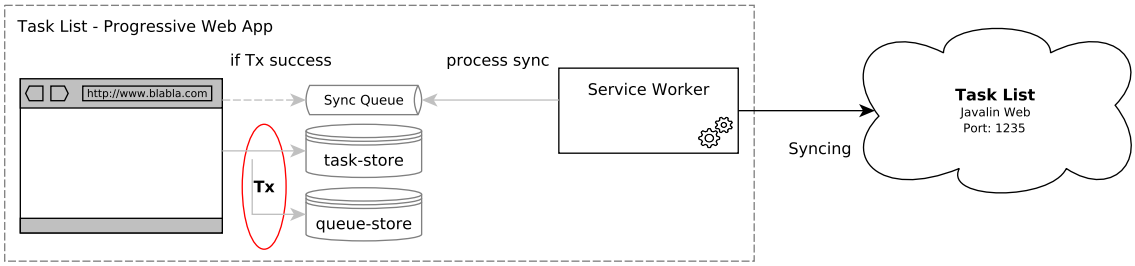


Figure 4.5: Detailed Architecture

Listing 4.11: Explaining the Detailed Architecture figure with Code

```
1  //...
2  const tx = db.transaction([TASK_STORE, QUEUE_STORE],
   ↪  "readwrite");
3  //...
4  //do the changes and push to the queue
5  //...
6  tx.done.then(() => {
7   //and if everything goes well, register a new sync event
8   navigator.serviceWorker.ready.then((reg) => {
9    reg.sync.register("sync-queued-data");
10  });
11 });
```

Let's move now to make the necessary code changes to implement what we have described. Looking at the source code project from the previous chapter pwa-v1-update, in the `src/server` folder, you will see the `tasks.js` module that encapsulates all the fetch requests to the Task List back-end services.

This module exposes four functions that are shown on listing 4.12. These functions are used from the `TaskList.js` and the `AddTask.js` components.

Listing 4.12: server/tasks.js exposed functions

```
// ...
// ...
// ...
return {
 doneOrUndone: doneOrUndone,
 retrieveAll: retrieve,
 delete: deleteOne,
 addNew: add,
};
// ...
```

What we are going to do now is to create a folder called `local` and then create a `tasks.js` module that implements those four functions but using the idb wrapper of IndexedDB API. Listing 4.13 shows how I have implemented the `tasks.js` module.

Listing 4.13: local/tasks.js

```
import { openDB } from "idb";
import { status } from "./taskStatus";

export let tasks = (function () {
  const DB_NAME = "taskdb";
  const STORE_DB = "task-store";
  const STORE_QUEUE = "queue-store";
  const OP_ADD = "add";
  const OP_UPDATE = "update";
  const OP_DELETE = "del";

  async function add(expirationDate, text) {
    let db = await openIndexedDb();
    let tx = db.transaction([STORE_DB, STORE_QUEUE],
    ↪  "readwrite");
    let dbStore = tx.objectStore(STORE_DB);
    let syncId = uid();
    let id = uid();

```

```
19        dbStore.add({
20          text: text,
21          expirationDate: expirationDate,
22          done: false,
23          syncId: syncId,
24          id: id,
25        });
26
27        let queueStore = tx.objectStore(STORE_QUEUE);
28
29        queueStore.add({
30          text: text,
31          expirationDate: expirationDate,
32          done: false,
33          syncId: syncId,
34          id: id,
35          op: OP_ADD,
36          queuedTime: Date.now(),
37        });
38
39        return tx.done.then(() => {
40          navigator.serviceWorker.ready.then((reg) => {
41            reg.sync.register("sync-queued-data");
42          });
43        });
44      }
45
46      async function deleteOne(taskToDelete) {
47        let db = await openIndexedDb();
48        let taskDel = await db.get(STORE_DB, taskToDelete);
49        let tx = db.transaction([STORE_DB, STORE_QUEUE],
          ↪ "readwrite");
50        await tx.objectStore(STORE_DB).delete(taskToDelete);
51        let queueStore = tx.objectStore(STORE_QUEUE);
52
53        queueStore.add({
54          text: "", //not needed
55          expirationDate: "", //not needed
56          done: false, //not needed
57          syncId: taskDel.syncId,
58          id: uid(),
```

```
59          op: OP_DELETE,
60          queuedTime: Date.now(),
61        });
62
63        return tx.done.then(() => {
64          navigator.serviceWorker.ready.then((reg) => {
65            reg.sync.register("sync-queued-data");
66          });
67        });
68      }
69
70      async function doneOrUndone(done, idTask) {
71        let db = await openIndexedDb();
72        let taskUpd = await db.get(STORE_DB, idTask);
73        taskUpd.done = !done;
74        taskUpd.id = idTask;
75        let tx = db.transaction([STORE_DB, STORE_QUEUE],
          ↪ "readwrite");
76        await tx.objectStore(STORE_DB).put(taskUpd);
77        let queueStore = tx.objectStore(STORE_QUEUE);
78
79        queueStore.add({
80          text: "", //only done or !done is updated
81          expirationDate: "", //only done or !done is updated
82          done: !done,
83          syncId: taskUpd.syncId,
84          id: idTask,
85          op: OP_UPDATE,
86          queuedTime: Date.now(),
87        });
88
89        return tx.done.then(() => {
90          navigator.serviceWorker.ready.then((reg) => {
91            reg.sync.register("sync-queued-data");
92          });
93        });
94      }
95
96      async function retrieve() {
97        let db = await openIndexedDb();
98        db.transaction([STORE_DB], "readonly");
```

```
99        let all = await db.getAll(STORE_DB);
100       let addedStatus = all.map((item) => {
101         return { ...item, status: status(item.expirationDate)
            ↪ };
102       });
103       return Promise.resolve({ tasks: addedStatus });
104     }
105
106     async function openIndexedDb() {
107       return await openDB(DB_NAME, 1, {
108         upgrade(db) {
109           if (!db.objectStoreNames.contains(STORE_DB)) {
110             let dbStore = db.createObjectStore(STORE_DB, {
111               keyPath: "id",
112             });
113           }
114           if (!db.objectStoreNames.contains(STORE_QUEUE)) {
115             let queueStore = db.createObjectStore(STORE_QUEUE,
              ↪ {
116               keyPath: "queuedTime",
117             });
118           }
119         },
120       });
121     }
122
123     function uid() {
124       return Date.now().toString(36) +
          ↪ Math.random().toString(36).substring(2);
125     }
126
127     return {
128       doneOrUndone: doneOrUndone,
129       retrieveAll: retrieve,
130       delete: deleteOne,
131       addNew: add,
132     };
133   })();
```

By looking at the implementation of the module I'm sure most of the code will be familiar to you as I have used the constructions shown in the

previous section. To not be boring, I'm not going to explain line by line the code above, but the important parts that help you understand how this works.

First of all, look at the `openIndexedDb()` function on line 106. Note that on the `upgrade` callback, on line 108, I'm creating two object stores. The task-store that will contain the task items and the queue-store that will contain the changes made to the task-store to keep it synchronized with the back-end service. As it was previously explained, these stores are illustrated on figure 4.5.

On line 12 above, you will see the `add` function. It basically adds new task items into the task-store (line 19). Note that in addition to giving to the new object the primary key `id` (line 24), I'm adding another unique identifier called `syncId` (line 23). It will be used to keep the task in sync with the back-end database. After that, I'm inserting the change into the queue-store. In addition to the task data required for the replication, I'm adding the `syncId`, the `op` which indicates which operation make the change (an addition in this case), and the primary key `queuedTime` which can be ordered to later apply the changes in the back-end in the same order as they were pushed.

On line 46 you can see the `deleteOne` function. After deleting the task from the task-store (line 50), I'm pushing the change into the queue-store. In this case the important properties are the `syncId` (it will be the reference to delete the task on the back-end database) and the `op` which is delete in this case. Of course, similar to the previous I added the primary key `queuedTime`.

On line 70 we have the `doneOrUndone` function. Following the same pattern as above, after updating the task (line 76), I push the change into the queue-store. From the use cases of the application we know that the only thing we can change of a task is if it is completed or not, so the `done` property is important here plus the `syncId`, `op` and the primary key `queuedTime`. Finally, note that in all the functions above (`add`, `deleteOne` and `doneOrUndone`), after making the changes, if the transaction succeeds registers the Background Sync event.

Those were the command functions that provoked the background sync service to run. Now, on line 96 we have the `retrieve` function used to display the list of tasks. This function gets all the items from the task-store (line 99) and then creates a new object that incorporates for each task item which status each task has. That is calculated based on the expiration date. See

below on listing 4.14 how the status function looks like.

Listing 4.14: local/taskStatus.js

```
export function status(expirationDate) {
  const diffInMs = Date.parse(expirationDate) - Date.now();
  const diffInDays = diffInMs / (1000 * 60 * 60 * 24);

  if (diffInDays <= 2) {
    return "DANGER";
  }
  if (diffInDays > 2 && diffInDays <= 5) {
    return "WARNING";
  }
  if (diffInDays > 5 && diffInDays <= 15) {
    return "FINE";
  }
  return "FUTURE";
}
```

To see these changes working on the browser we have to change the `AddTask.js` and `TaskList.js` components to use the `local/tasks.js` module instead of the `server/tasks.js` module. Go ahead and apply these changes. After that, start the application with the `npm start` command and navigate to `http://localhost:8000/`. You will first need to sign in and then go to the Task List. Of course, until you don't add new entries the list will be empty. However, as illustrated in figure 4.6 note that the taskdb database is created with both object stores. They are created because the `TaskList.js` component invokes the `local/tasks.js/retrieve` function which first opens the database and since the database is not there yet it gets created.

Now, if we add a task, see how they look on both object stores in figures 4.7 and 4.8.

And if we delete that task, the task-store will be empty but the queue-store will have two entries now. See figure 4.9. Note that there are two entries, one that corresponds to the add operation we did, and the last one that corresponds to the delete operation. See the property `op:"del"`.

Now that we have all the operations implemented using IndexedDB it is time to add the Background Sync service. In order to synchronize the data the Task List back-end exposes the following POST API https://{host}/tasks/bulk, which expects as POST's body a list of queue-store objects. Like below:

```
[
    {
```

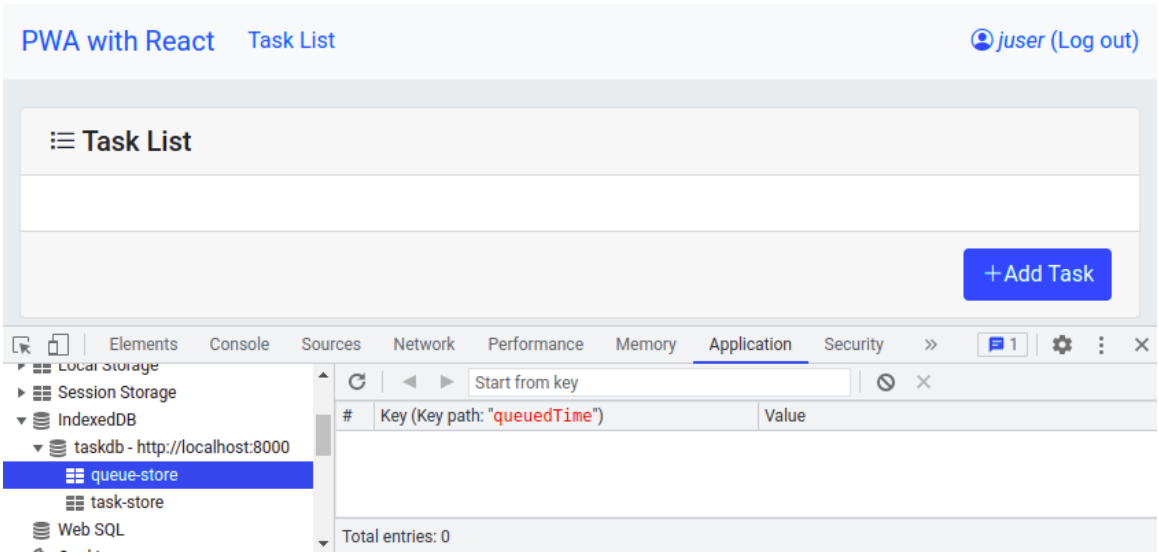


Figure 4.6: Both object stores created and empty

```
        "text": "That very important thing !",
        "expirationDate": "2022-10-22 15:00",
        "done": false,
        "syncId": "l49wmop164at0t3y0vc",
        "id": "l49wmop142wy9g39a04",
        "op": "add",
        "queuedTime": 1654953577093
    },
    {
        "text": "",
        "expirationDate": "",
        "done": false,
        "syncId": "l49wmop164at0t3y0vc",
        "id": "l49wq99pst66gio94",
        "op": "del",
        "queuedTime": 1654953743725
    }
]
```

Here you can find the implementation of the back-end service. It is pretty simple. It will first reorder the list by queuedTime and in that order will apply each operation. Everything wrapped in a transaction, which means that all will be applied or nothing. To consume this API I'm going to add a new

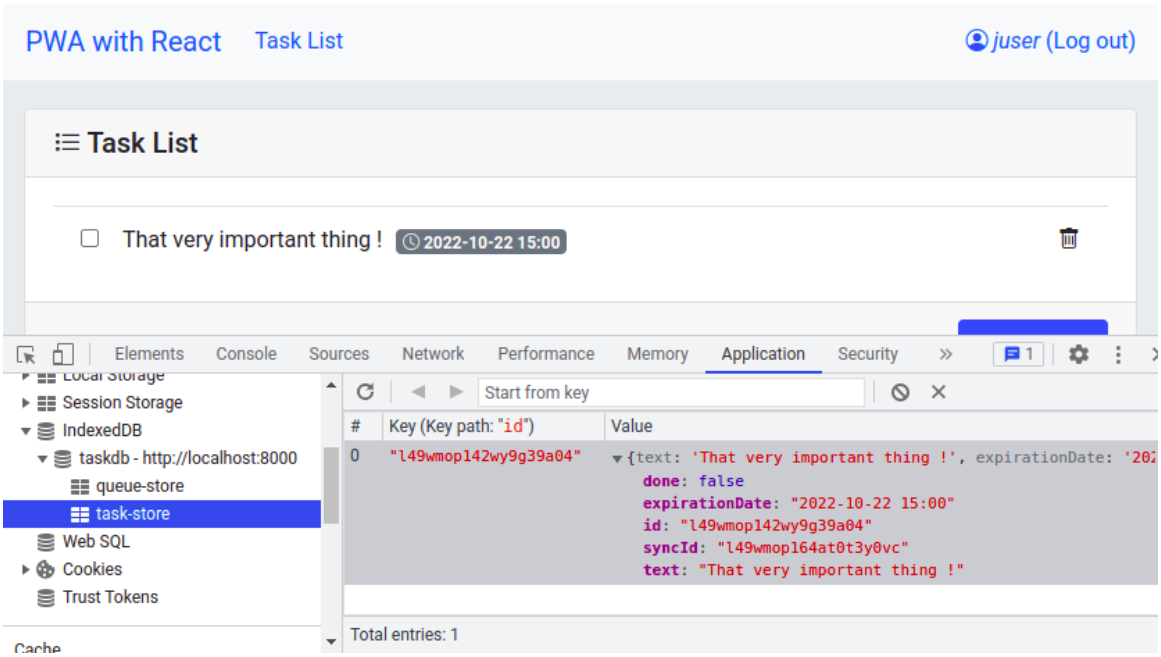


Figure 4.7: task-store with one entry

function called `syncTasks()`. See the details on listing 4.15 below.

Listing 4.15: server/syncTasks.js

```
export let syncTasks = (function () {
  const apiTask = process.env.REACT_APP_URI_TASK;

  function bulk(tasks) {
    return fetch(apiTask + "/tasks/bulk", {
      method: "POST",
      credentials: "include",
      body: JSON.stringify(tasks),
      headers: {
        "Content-type": "application/json; charset=UTF-8",
      },
    })
      .then((r) => {
        if (r.status === 401)
          return Promise.reject({
            status: 401,
```

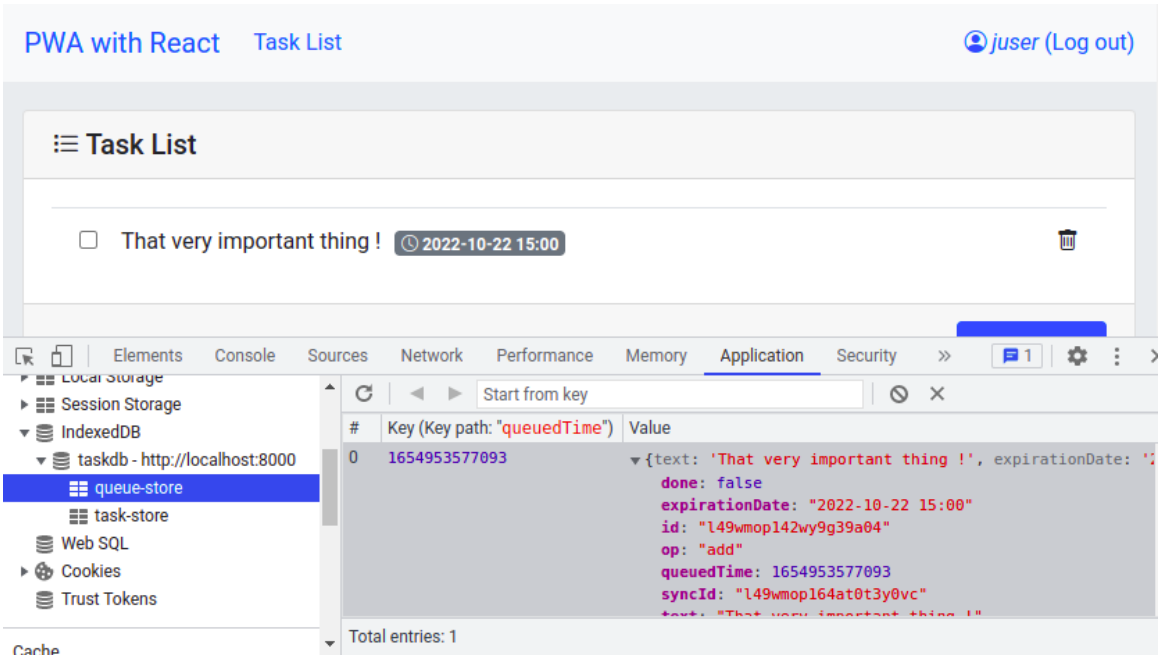


Figure 4.8: queue-store with one entry

```
17              msg: "You are not authenticated",
18            });
19          if (!r.ok) return Promise.reject(r.status);
20          return r.json();
21        })
22        .then((json) => Promise.resolve(json))
23        .catch((e) => Promise.reject(e));
24    }
25
26    return {
27      bulkTasks: bulk,
28    };
29  })();
```

The function exported above receives a list of tasks as parameters, that we will obtain, as we will see later, from the `queue-store` and creates a fetch POST request. If the request succeeds it will resolve, otherwise reject. Add this module to the project in the `server/syncTasks.js` file.

Let's now add the changes to the service worker. In the listing 4.16 below,

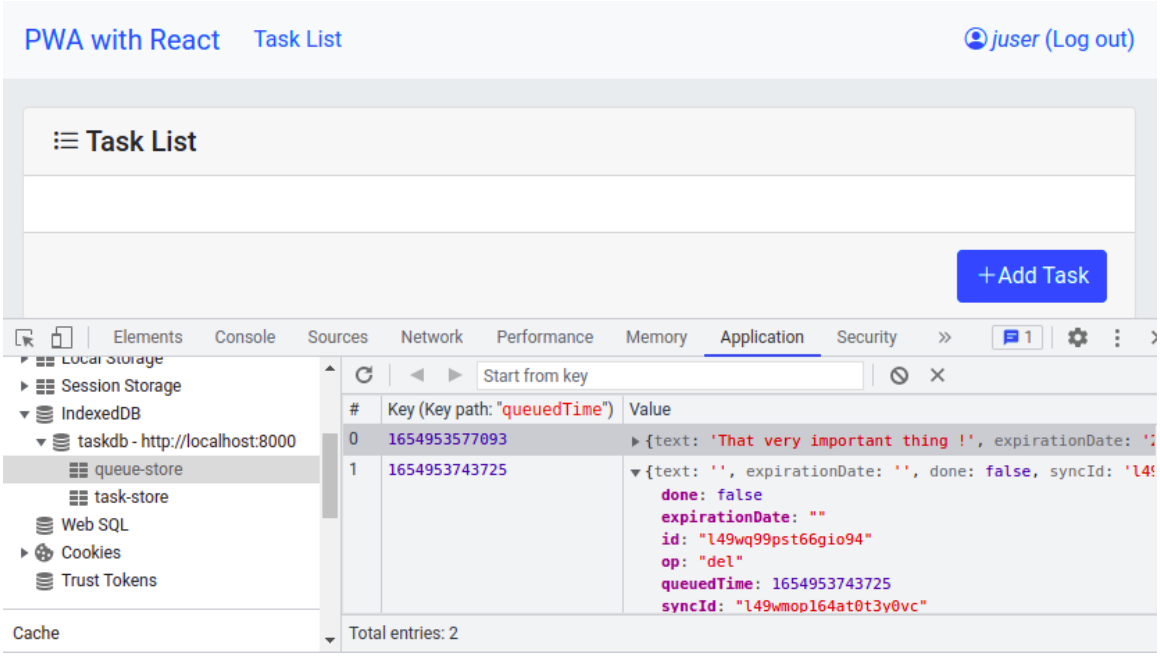


Figure 4.9: queue-store with two entries while task-store is empty

you will find the changes to add the sync event listener and the doSync() function that does the syncing work.

Listing 4.16: Service Worker Sync Event Listener

```
1  self.addEventListener("sync", (event) => {
2    event.waitUntil(doSync(event));
3  });
4
5  async function doSync(event) {
6    console.log("it was executed...", event);
7    if (event.tag === "sync-queued-data") {
8      let all = await tasksLocalService.getAllQueued();
9      return syncServer.bulkTasks(all).then(() => {
10       return tasksLocalService.deleteAllQueued();
11     });
12   }
13 }
```

Note above, on line 7, if the event.tag is equals to sync-queued-data, it will retrieve all the queued items from the queue-store and call the syncServ

er.bulkTasks. Finally, if that remote API call resolves, all the queued items from the queue-store gets deleted. Below is the implementation of the local/tasks.js/deleteAllQueued() and local/tasks.js/getAllQueued() functions.

Listing 4.17: Retrieve all and Delete all queued items

```
async function deleteAllQueued() {
 let db = await openIndexedDb();
 return await db.clear(STORE_QUEUE);
}

async function getAllQueued() {
 let db = await openIndexedDb();
 return await db.getAll(STORE_QUEUE);
}
```

After adding all these changes to the project we are ready to see it working. Create a production build and start the project. Every time you add, update or delete a task, if there is connectivity, the sync event will be triggered logging in the Chrome dev tool console the action, as illustrated in figure 4.10.

Now, suppose your phone crashes and you buy a new one. You will want to have all your tasks back in your new phone, right?. Let's add a simple button on the Welcome.js component that performs this job. Additionally, I will add another button that prints on the console what are the tasks we have on the server. This one is only for debugging purposes so you can verify easily that the changes are correctly replicated. On the local/tasks.js add and expose the following function:

Listing 4.18: Start from Server function

```
async function startFromServer() {
 let all = await tasksService.retrieveAll();
 let promisesAdd = [];
 let db = await openIndexedDb();
 let tx = db.transaction([STORE_DB, STORE_QUEUE],
  ↪ "readwrite");
 tx.objectStore(STORE_DB).clear();
 tx.objectStore(STORE_QUEUE).clear();
 all.tasks.forEach((task) => {
   promisesAdd.push(tx.objectStore(STORE_DB).add(task));
```

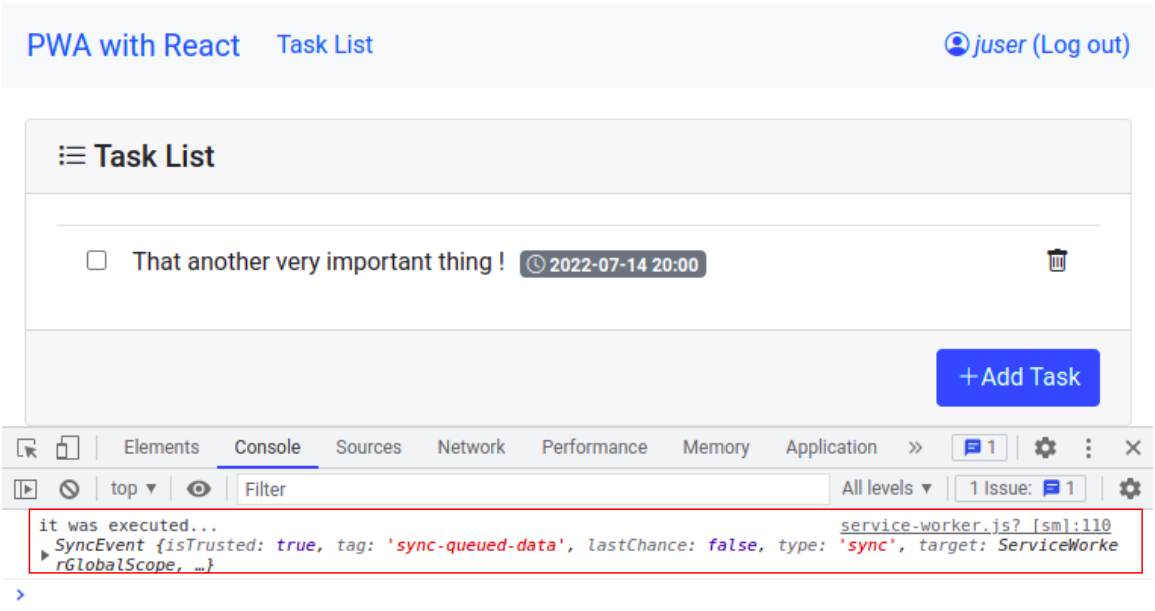


Figure 4.10: Sync Event log in Chrome Dev tool

```
  });
  await Promise.all(promisesAdd);
  return tx.done;
}
```

And replace the `Welcome.js` component with the one below:

Listing 4.19: Welcome.js

```
import Container from "react-bootstrap/Container";
import { useLocation } from "react-router-dom";
import Card from "react-bootstrap/Card";
import { Button } from "react-bootstrap";
import { tasks as tasksService } from "./server/tasks.js";
import { tasks as tasksLocalService } from
↪  "./local/tasks.js";
import { ToastContainer, toast } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";

export default function Welcome() {
  const location = useLocation();
```

```
12
13      function handleCheckWhatsOnServer() {
14        tasksService.retrieveAll().then((json) => {
15          console.log(json);
16          toast.success("Items retrieved from server and printed
              ↪ on console");
17        });
18      }
19
20      function handleStartFromServer() {
21        if (window.confirm("I will proceed, ok?") === true) {
22          tasksLocalService.startFromServer().then(() => {
23            toast.success(
24              "Everything was retrieved from server and created
                  ↪ locally"
25            );
26          });
27        }
28      }
29
30      return (
31        <Container fluid className="mainBody">
32          <ToastContainer />
33          <Card>
34            <Card.Header as="h5">Welcome !</Card.Header>
35            <Card.Body>
36              <p>{location.state}</p>
37              <p>
38                Welcome to the Task List Application. This is to
                    ↪ demonstrate how to
39                create a <b>Progressive Web App</b> with React.
40              </p>
41              <p>
42                Do you want to check which items are on the
                    ↪ server?
43                {"  "}
44                <Button onClick={handleCheckWhatsOnServer}>Click
                    ↪ here!</Button>
45              </p>
46              <p>
```

```
47                Do you want to clear everything and retrieve all
                   ↪  items from server?
48                {"  "}
49                <Button onClick={handleStartFromServer}>Start
                   ↪  from Server</Button>
50              </p>
51            </Card.Body>
52          </Card>
53        </Container>
54    );
55  }
```

With this in place, we can do a full test of the application. Start the back-end services as explained in 2.1.1 or 2.1.2. Create a production build and start the application. Then, navigate to `http://localhost:8000/`. First sign in and then go to the welcome screen. If you click on the button that prints on the console the items that are on the server, you will see the four items that the back-end Task List service has created by default. This is illustrated on figure 4.11. Finally, if you click on the "Start from Server" button, you will get those four items created locally and a toast message will inform the user when finished as illustrated on figure 4.12. Now, you can start making changes to the task list and then verify what there is on the back-end to make sure everything is working.

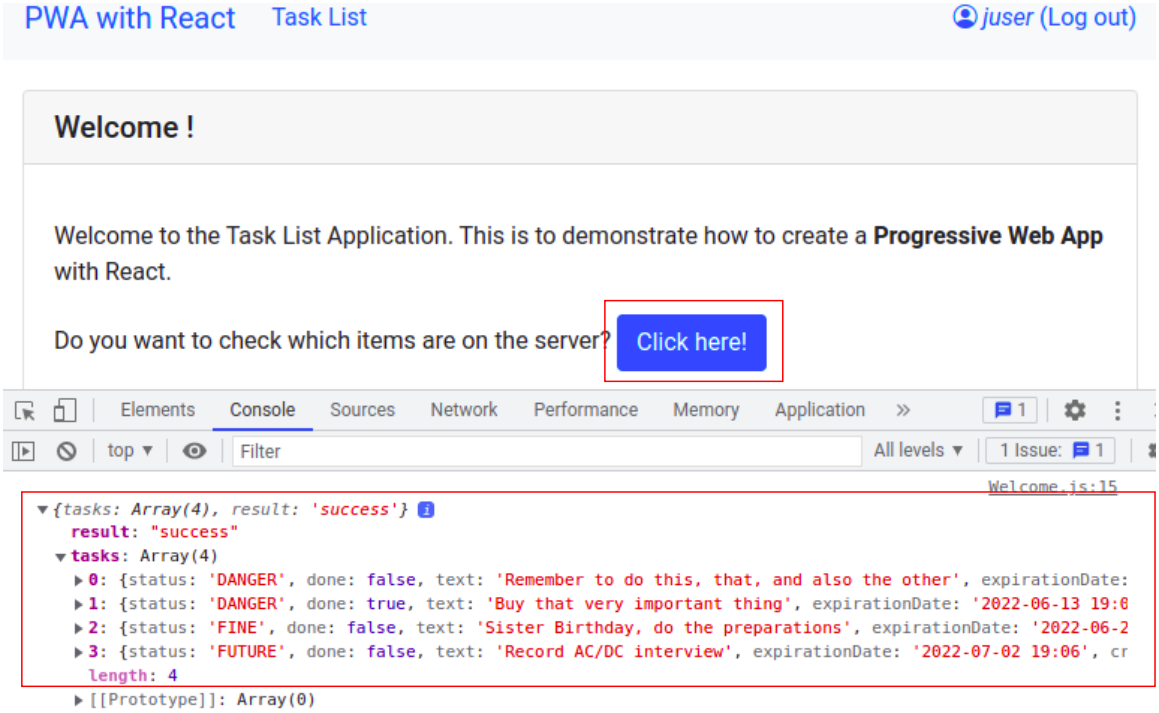


Figure 4.11: Check which items are on the server

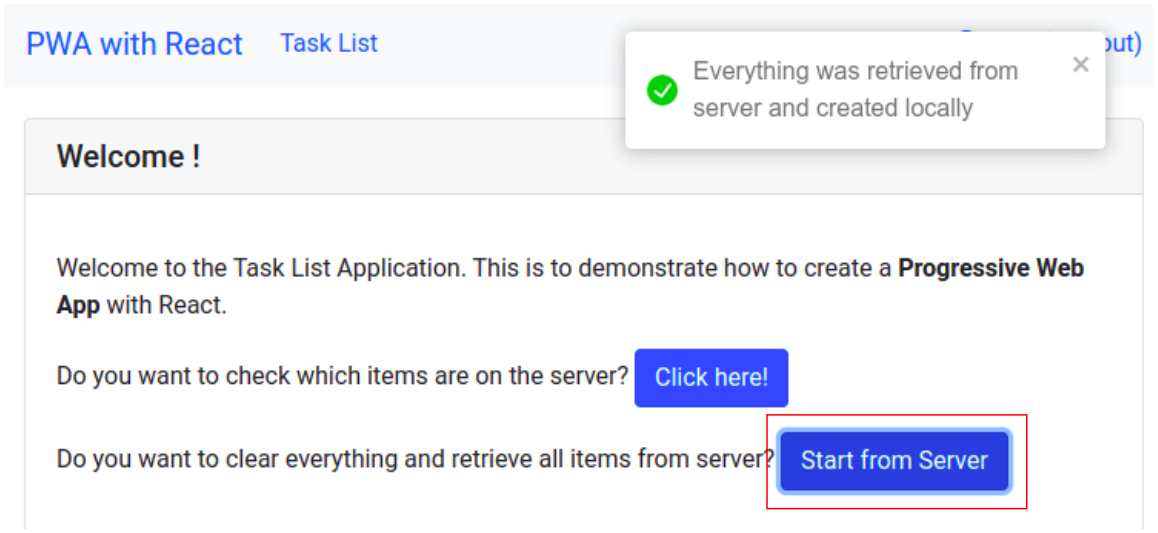


Figure 4.12: Start from the Server

4.4 Handling Syncing Errors

What happens if the sync fails? It may occur that the device in which you are running the Web App does not have good connectivity or a component in the back-end is not functioning properly or is down. We have mentioned, at the beginning of this chapter, that if the `doSync()` returns a rejected promise the sync process will retry later. In Chrome, the second time will happen exactly 5 minutes later from the previous one. If the second time fails again, there is another one (and the last one) that runs 15 minutes later from the previous time. Note that this third attempt is the last one. This last chance is informed by a property called `lastChance` in the `event` object. Figure 4.13 shows the console log of Chrome with these three attempts. Note the `lastChance` property.

Knowing how this works is pretty important. In the case of the Web App we have built in this book, this is not a big problem. All the changes that need to be synchronized are persisted in the `queue-store`, so, if for any reason the sync is discarded, the next time there is any change done by the user in the Web App, the sync is rescheduled and everything will be synced (previous changes and new changes). However, this might not be the case of all the applications and use cases you might have. By reading the `lastCha⌋nce` property we are able to do something else if it is required as we will see later.

On the other hand, as it was described before, any call to a UserAuth or TaskList back-end API requires a valid authenticated token. And this is also the case for the back-end API we consume to replicate the data stored in IndexedDB. When the sync event is triggered, the Web App consumes the back-end API with a token. If the token is valid the service will proceed with the replication, if not, it will return with an unauthorized error, which will reject the promise and the background sync service will try later. This is an important point, authentication is still required to be online. The user must sign in, grab a valid token and after that, it might go offline. What might be an option here is to extend the life of the token (not good for security though), so the application will have few chances to suffer during replication. The UserAuth service can be started with the parameter `offline=true`, as shown below. This will make the token expire in seven days.

```
$ mvn exec:java
↪  -Dsecret=bfhAp4qdm92bD0FIOZLanC66KgCS8cYVxq/KlSVdjhI=
↪  -Doffline=true
```

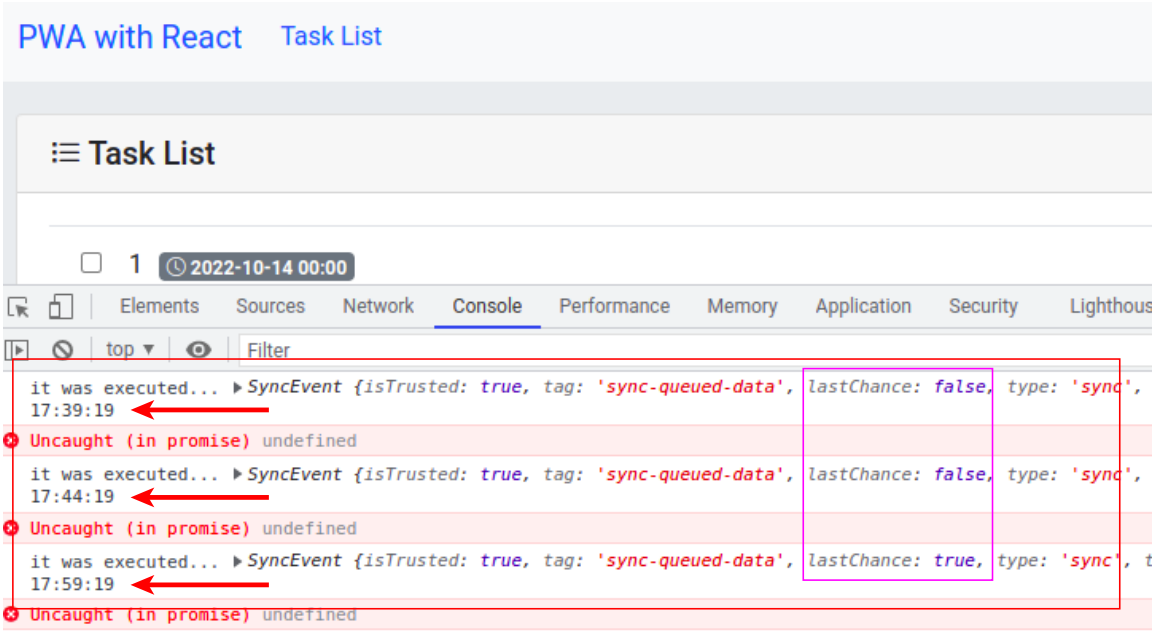


Figure 4.13: Errors, Retries and lastChance

Anyway, whatever the problems are during synchronization it would be ideal to inform that to the end user. And to do that we have to perform a communication from the service worker to the Web App. We have explained in section 3.1 how to perform a communication from the Web App to the service worker. Now, we are going to show how to do it from the service worker to the Web App to inform the end user if there is an error during the syncing process. To implement this, let's first add the message event listener to the `index.js` file.

Listing 4.20: Adding the message Event Listener to the index.js

```
1 navigator.serviceWorker.addEventListener("message", (event)
  ↪ => {
2   if (event.data.type === "SYNC_ERROR") {
3     toast.error(
4       "Something went wrong with the sync service: " +
        ↪ event.data.msg
5     );
6   }
7 });
```

In listing 4.20 above we are listening to the message event coming from the service worker. As parameter, it receives an `event` object in which we can pass data from the calling/triggering client. If the `event.data.type` is `SYNC_ERROR` then we will show the error to the end user.

Now, on the `service-worker.js` we have to change the `doSync()` function to incorporate the `postMessage` call if there are errors.

Listing 4.21: Communicate errors to the Web App

```
1  async function doSync(event) {
2    console.log("it was executed...", event);
3    if (event.tag === "sync-queued-data") {
4      let all = await tasksLocalService.getAllQueued();
5      return syncServer
6        .bulkTasks(all)
7        .then(() => {
8          return tasksLocalService.deleteAllQueued();
9        })
10       .catch((e) => {
11         self.clients.matchAll().then(function (clients) {
12           if (clients && clients.length) {
13             clients.forEach((client) => {
14               client.postMessage({
15                 type: "SYNC_ERROR",
16                 msg: e.msg,
17               });
18             });
19           }
20         });
21         return Promise.reject();
22       });
23   }
24 }
```

Note that on listing 4.21 above starting at line 10, we have added the catch block. On line 11, the sentence self.clients.matchAll() will obtain all the clients (browser window tabs) controlled by the service worker. To each of them, we will post the `SYNC_ERROR` message with the actual error coming from the back-end API in the `msg` property. And finally, the `return Promise.reject()` is important to inform the browser that the syncing must be tried again later. With these changes, the user will be informed about any

error that might occur at sync time. Figure 4.14 illustrates how the message is presented to the user. In this case, the user is not authenticated at the time the sync service ran.

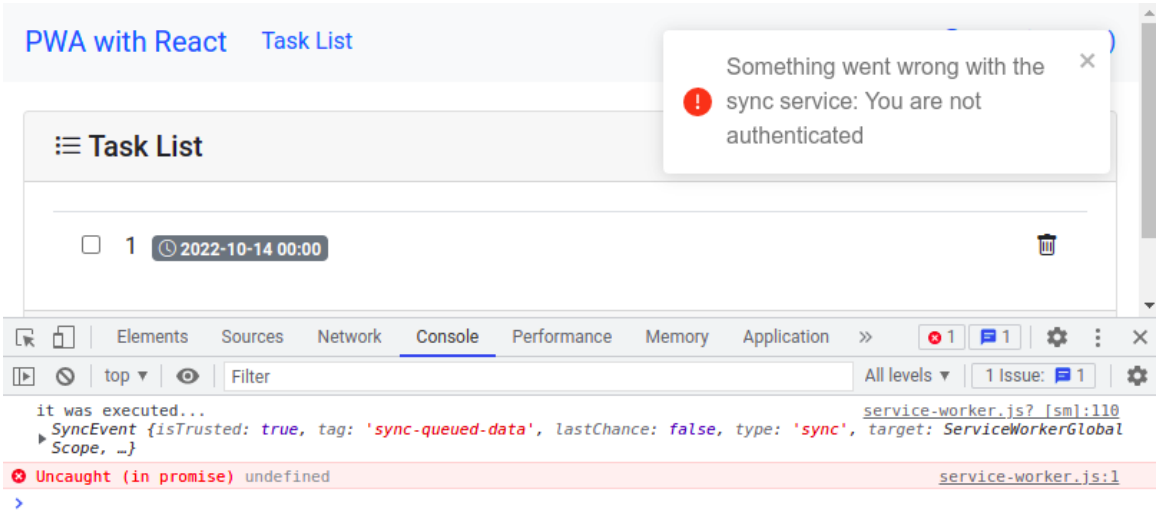


Figure 4.14: Sync Error Shown to the User

Finally, if you want to do something in case the `lastChance` is `true`, we can pass that value to the Web App and re-register the sync event. We can do this by adding the highlighted changes shown on listing 4.22 below.

Listing 4.22: re-register the sync event when lastChance

```
navigator.serviceWorker.addEventListener("message", (event)
↪  => {
  if (event.data.type === "SYNC_ERROR") {
    toast.error(
      "Something went wrong with the sync service: " +
      ↪  event.data.msg
    );
    if (event.data.lastChance) {
      navigator.serviceWorker.ready.then((reg) => {
        reg.sync.register("sync-queued-data");
      });
    }
  }
});
```

And in the service worker `doSync()` function we have to pass the `lastChance` property value, as shown below:

Listing 4.23: Passing lastChance to the Web App

```
async function doSync(event) {
  console.log("it was executed...", event);
  if (event.tag === "sync-queued-data") {
    let all = await tasksLocalService.getAllQueued();
    return syncServer
      .bulkTasks(all)
      .then(() => {
        return tasksLocalService.deleteAllQueued();
      })
      .catch((e) => {
        self.clients.matchAll().then(function (clients) {
          if (clients && clients.length) {
            clients.forEach((client) => {
              client.postMessage({
                type: "SYNC_ERROR",
                msg: e.msg,
                lastChance: event.lastChance,
              });
            });
          }
        });
        return Promise.reject();
      });
  }
}
```

With this, we have finished adding the background sync service to Task List. The full source code of this version of the application can be found at react-pwa-tasklist-v2.