
COMPLETE GUIDE TO
QWIK
Redefine Web Performance with a Resumable Approach for Instant Startup
GIORGIO BOA
FORWARD BY MIŠKO HEVERY
newline

Qwik in Action

Harness Qwik's Resumable Architecture for Lightning-Fast Startup Times

Written by Giorgio Boa
Forward by Miško Hevery
Edited by Zao Yang

Published by \newline

Contents

Qwik in Action

Dedication

To Marzia, my wife, because she is an extraordinary and patient life partner.

Giorgio Boa

Foreword by Miško Hevery

Why another framework?

The short answer is that the current approach to framework design is not producing fast sites, and this problem will not disappear by simply waiting for a faster CPU and network.

What is the core problem?

The core problem is that the industry has become very proficient at building complex applications to solve complex needs. The web applications of today are more capable than those of yesterday. Code bases are larger than ever. All of this is fantastic and a sign of the maturing industry.

However, tools (frameworks) still need to catch up with codebases. All frameworks need to know about the application before they can run. Frameworks need to know about component boundaries, state, and event listeners. However, how the frameworks obtain this information has a significant impact on how much code needs to be downloaded to the client on application startup.

Most frameworks use a technique called hydration. Hydration recovers all of the framework state (component boundaries, application state, event listeners) by re-executing all of the components on the page, starting with the root component and recursively visiting all components on the page. This is fine for small codebases, but as the applications become more complex, the amount of code needed to be executed by the browser on page startup becomes overwhelming, resulting in slow startup performance. This is the current O(n) approach to interactivity.

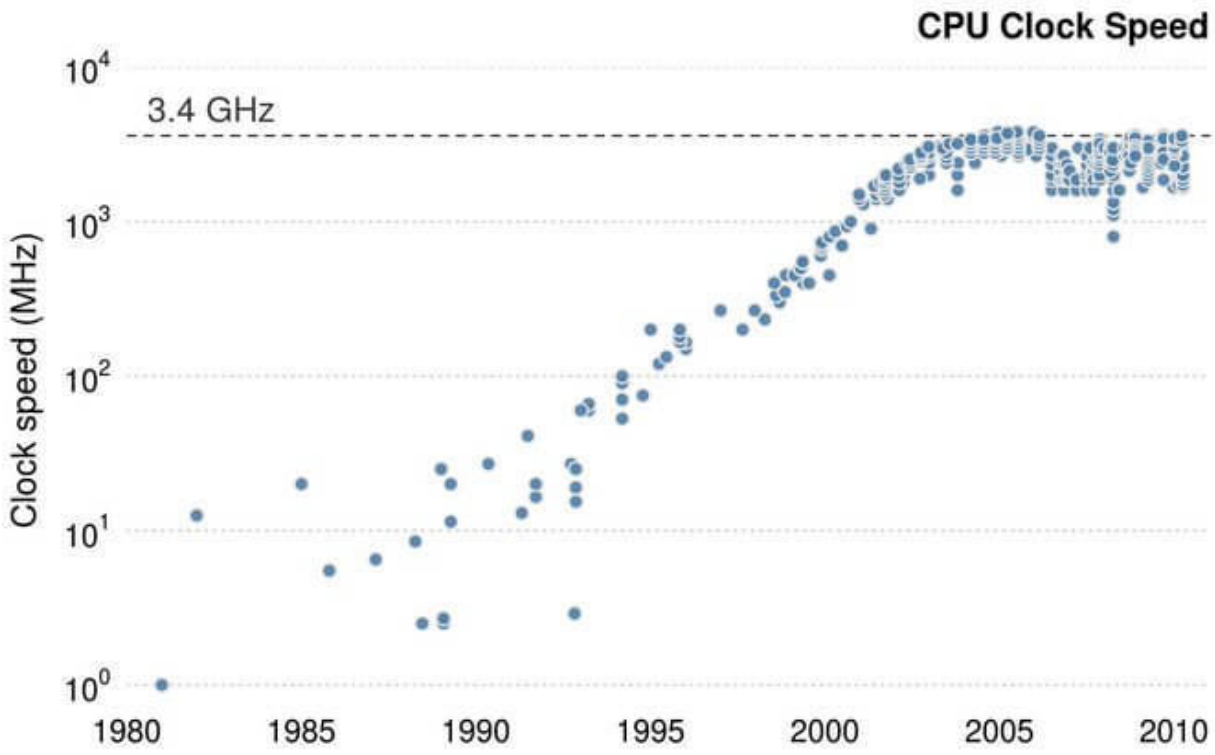
Qwik does not use hydration but instead employs resumability. Resumability takes advantage of the fact that the application has already been executed on the server as part of SSR/SSG, and the framework can serialize component boundaries, application state, and event listeners into the resulting HTML. On the client, the framework can deserialize the information instead of executing the code again. Resumability does not require running any code on application startup. This creates constant startup costs, regardless of the application complexity. This is the O(1) approach to interactivity.

The use of resumability opens up the possibility of lazy loading. Hydration requires eager execution of all components on the current page, which means that none of the components on the page can be lazy loaded. Only components not currently in the render tree can be lazy-loaded. Without hydration, loading all components on the page lazily is now possible. Now, the amount of code that the browser needs to execute is proportional to the interaction's complexity rather than the page's complexity.

But CPUs and networks are getting faster. Won't this problem go away?

In short, no. CPUs get faster in three ways:

1. By increasing the clock cycles
2. By increasing the number of instructions per clock cycle.
3. By adding more CPU cores and parallelizing the applications.



CPU Clock Speed

Image from Twitter[1]

[1] https://twitter.com/csgillespie/status/732532249325907968?lang=id

The limit on clock speed has been reached, and it is about 3.5 GHz. Why? At 3.5GHz, light can travel about 8.5cm, which is less than one order of magnitude to the size of a silicon die. Getting close to the theoretical limit of what Physics will allow for a signal to travel from one end of the chip to the other explains the stagnation of clock speeds.

Modern CPUs try to parallelize many instructions concurrently. But more parallel designs are more complex. In addition, code is sequential, so parallelizing instructions is difficult and relies on speculative execution. So, it is unlikely that instruction parallelism can ever be more than a handful of instructions at a time.

Almost all growth in performance in recent years has been due to multiple CPU cores running applications in parallel. It is common for phones to have half a dozen cores. But there is a catch: JavaScript is single-threaded, which means it can only run on one core at a time. So, websites do not benefit from multiple CPU cores.

If sites require ever more JavaScript, then it is unlikely that growth in processing speed will be able to fix the startup performance problem. The solution to JavaScript bloat must come in the form of spreading the cost over time rather than requiring all of the JavaScript to run on startup, as is currently the case.

Hydration alternatives

People have been trying to solve the hydration issue for a while, but none of the solutions have really caught on at scale because they all have trade-offs.

Hydration sabotages the lazy loading.[2]

- Islands - Extra work for the developer. Inter-island communication is a problem. Islands tend to grow over time.
- Server Components - Server components can't have state and interactivity. If app requirements change over time, component boundaries may need to be refactored, which may not be trivial.
- Progressive Hydration - Improves Time-To-Interactive (TTI) but does not improve the amount of code that needs to be executed.

The real issue is that hydration-based frameworks were designed as client-side rendering (CSR) frameworks, with server pre-rendering as an afterthought. This design choice forces the framework to re-run the application code on hydration. The framework only knows how to CSR the application. The fact that the CSR can reuse the DOM elements is incidental and does not lower the JavaScript requirements.

Why another framework?

Everyone stands on the shoulders of others. Qwik is not the first resumable framework, but it is the first open-source resumable framework. Google has Wiz that has been resumable for the past ten years; it runs Google Search, Photos, and many other properties. These apps are fast!

I really tried to integrate resumability into existing frameworks with a good Developer Experience (DX). I tried every rendering engine I could think of, but no luck; the assumptions that existing frameworks make and presumably requirements are not compatible. Through a process of elimination of existing solutions, a new solution was born.

[2] https://www.builder.io/blog/hydration-sabotages-lazy-loading

Easy is the same as performant

Many frameworks have great DX. They also have escape hatches. These escape hatch APIs are used once the codebase becomes large to "optimize" the application for speed. Ways of pruning the render tree to minimize the blast radius of framework rendering. The implication is that there are two ways of writing components. The DX-friendly/simple way and the performant way. It was ensured that Qwik only had one way. There will not be optimization APIs to "memoize" things to limit the framework rendering; instead, in Qwik, everything is memoized all the time. That is not to say that a slow application can't be written in Qwik; developers are a creative bunch, but rather that the happy path in Qwik is the performant path. There is no alternative. This philosophy permeates through Qwik. Here are a few things that don't need to be done in Qwik:

- Memoize state (all state is always memoized)
- Lazy-load components (all code is automatically broken down into functions and available for lazy loading)
- Prefetch code (Qwik service worker automatically prefetches code)
- Bundle optimization (Qwik-insights uses AI to optimize bundling, minimize waterfalls, and determine the order of prefetch.)

In places where Qwik can't ensure the right solution, it provides guide rails:

- Notify early of images that are missing size and cause CLS.
- Suggest image optimization by auto-fixing the code.
- Highlights which areas of the UI experience layout-shift

Not just another framework

There are a lot of frameworks to choose from! They differ on API. Do components need to be in a separate file? Are components markup with behavior or behavior with markup? But even though there are many choices for DX, it can be argued that if one zooms out, all of these frameworks work the same way. They are all CSR frameworks, with server rendering and hydration bolted on.

Qwik is very familiar with the DX and other popular frameworks. This is intentional to make developers feel at home. But don't be fooled by its familiarity. It works on fundamentally different principles. It is an SSR-first framework that focuses on startup performance by delivering nothing but HTML + CSS out of the box. This is in stark contrast to other approaches.

The hope is that Qwik will become successful, but no one can predict the future. However, it is certain that the next big framework will not rely on hydration.

Qwik loves the community

I express my sincere gratitude to the Qwik community. Qwik is more than just an open-source project; it is a venture designed to be operated and owned collectively by the community. The Qwik community

actively engages in supporting one another, responding to inquiries, prioritizing and resolving issues, crafting solutions, and determining the project's future trajectory. The existence and success of Qwik owe much to the dedicated involvement of the community.

Miško Hevery

Introduction

Introduction

Nowadays, the frontend developer has a well-defined role, but in the past, this was not the case because the applications were simpler both graphically and in terms of functionality, and once you had mastered HTML and CSS, you could already start creating something that was deemed adequate for most cases.

In the past, the first web applications were simple pages; they were static documents, and when the user visited a page, he initiated a client-side request, which was then managed by a server. The web server prepared the page, and during the page creation phase, it could also query a database, returning a result that was then displayed in the user's browser, so the browser contacted the web server with an HTTP call and then returned the response that contained the calculated results. Therefore, this process allowed the creation of interactive experiences for users who, for example, could navigate the application and modify data via links, buttons, and forms. This process was the standard for the time but limiting because the technologies used were not up to modern solutions and involved a roundtrip of information and manipulation that left room for improvements.

With the arrival of JavaScript in 1995, the possibility of adding dynamism to the web page was born, validating inputs and hiding or showing elements without performing the previously seen roundtrip between Client and Server with the advantage of a fast and pleasant user experience. From this moment on, evolution continued to make great strides forward, with different ways of making the web application interactive, but in 2005, a turning point came with the birth of AJAX (Asynchronous JavaScript + XML) because its "asynchronous" nature allowed to communicate with the server, exchange data and update the page without having to refresh the page, this made the application faster and more responsive to user actions.

The constant evolution of Web APIs that continues today has allowed the creation of increasingly interactive interfaces and the ability to communicate with external services very easily; this has given rise to increasingly interactive applications, graphically beautiful and useful because there is the ability to incorporate many more features. Now, you can create real web applications, mobile applications that often have user experiences similar to native ones, applications for smart TVs, IoT devices, and much more. Today, the limits imposed by browsers are light years away from what was once possible; now, the responsibility of creating good or bad applications depends on you because the means available are all there; you need to know how to identify them and apply them consistently to master them.

Running code examples

You can download code examples from the same place where you purchased this book.

If you have any trouble finding or downloading the code examples, email us at us@fullstack.io[3].

[3] mailto:us@fullstack.io

TypeScript

TypeScript merits a separate paragraph due to its arrival in 2012, revolutionizing front-end application development. The official website states: “TypeScript is JavaScript with type syntax.” This sentence alone provides significant insight. Developed by Microsoft and the same creator of C#, Anders Hejlsberg, TypeScript introduced types into an untyped language like Javascript to ensure code consistency.

JavaScript is a language interpreted by the browser, and any syntax or logic errors are discovered in the browser when the code is executed. This means that errors only surface when a feature is in use. For example, consider a button that saves a user profile. The saving code is only executed after pressing the “save user” button, and only then is the mistake discovered. With TypeScript, however, code verification shifts to the build phase because TypeScript files are compiled into JavaScript files during this phase. It’s important to remember that only JavaScript can be read by the browser, serving as the common language. Therefore, TypeScript essentially provides a compiled language. In the given example, any error on the “save user” button would be identified during the compilation phase, even before deploying the application. This approach significantly reduces the errors that reach the end user.

It’s worth noting that JavaScript purists didn’t immediately embrace TypeScript. Despite initial resistance, it has become an indispensable language with each new release. Its simplicity and efficiency have led to its incorporation into many modern tools, and the most popular frameworks now support TypeScript by default.

Additionally, it’s important to note that not all modern constructs offered by TypeScript are supported by browsers. However, TypeScript allows the freedom to write modern code, and during the build phase, it translates the code into fully functional JavaScript.

Frontend Architectures

Starting with a greenfield application often presents a fortunate opportunity because it allows for the imagination to run wild. It’s a chance to make decisions and consider what might be best for the application. Is this or that architecture better? How can the application be distributed to everyone quickly and easily? It’s easy to get caught up in the moment’s hype, but the best thing to do in these moments is to try to stay calm and put all choices on the table.

Listing the requirements to be met is certainly the first step to take. Who are the users? Will they use mobile or desktop? In what environmental conditions are they required to use the application? For example, suppose the typical platform user uses the app in an area with zero connectivity. In that case, this difficulty must be considered and, therefore, cannot be forgotten.

Once all these requirements are clear, the next step is to make the list of frontend architectures that allow for achieving the desired results within the deadlines imposed by the project. But what are the different architectures that can be chosen? The most important ones will be explained. This list will be very useful for aligning some terms and considerations that will be used throughout this book. The book’s main topic is Qwik, but before learning to run, it’s important to know how to walk.

Client Side Rendering (CSR)

Client Side Rendering is the process of sending to the browser a series of HTML, CSS and JS files with our logic and these files are the result of the compiling code we have written and the libraries we have used. Single-Page applications (SPA) use this process and have been in vogue for many years in the frontend world starting with AngularJS way back in 2010 and then continuing with Angular and React and now many other frameworks. AngularJS was a game changer for frontend development because it replaced a way of developing applications where developer experience was sorely lacking.

> Developer Experience (DX): We have already heard and digested the term user experience (UX) and it is the experience you have when using a product, service, or any system. This experience is based on the ease of use, its usefulness, and the perception we have of its use. If these users are developers then we can talk about developer experience (DX). Organizations that offer products, tools, or services for developers aim to improve DX. This particular attention leads to happy and loyal developers who will continue to use the products, tools, or services. This will also bring a further benefit because it will create word of mouth which will positively benefit and increase its use. A clear example is the presence of good documentation, easy to understand and resolves doubts very quickly.

At the time it was very difficult to write and maintain applications because vanilla JavaScript was used versus a more modern TypeScript used today. Using JavaScript in large applications used to be the norm, but thinking back to the effort required to write such applications at the time and how easily bugs could be introduced without obviously knowing makes me smile. But let's not waste time talking and get back to the topic of this paragraph, Client Side Rendering. The more features we add to our Single-page application, the larger the JS files will be and will affect the loading speed. There is a clear direct correlation between functionality and required JavaScript, the bigger your app gets, the more JavaScript it will need at the expense of performance.

> Libraries can be included to take advantage of functions written by third parties. e.g., If I want to be able to easily manipulate dates and time zones, instead of writing all the logic to handle the various cases, I can download and use the DayJS library distributed via NPM, which is the repository of all the JavaScript libraries and more.

However, it's useful to know that the more dependencies included in the project, the more the size of the application bundle increases because the functionality is being added to the code base. Therefore, one of the best practices to apply is to keep unnecessary dependencies to a minimum. The compilation process will allow for optimizing and packaging the application code of the SPA and obtaining files that can be loaded when the application starts.

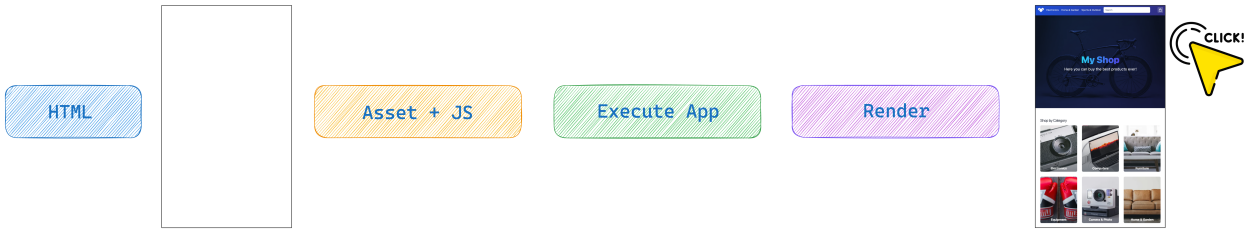
A usually very concise HTML file (index.html) will be produced, which, via script tags, will include all the JS and CSS necessary to display and make the application interactive.

This is an example of a real-world SPA index.html file. It can be noticed that it will appear as a blank page because there are no things to display.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>...</title>
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalab\
le=no, viewport-fit=cover"
    />
    <link
      rel="icon"
      type="image/x-icon"
      href="favicon.ico"
    />
    <meta name="description" content="..." />
    <link rel="stylesheet" href="theme.2780d3.css" />
    <link rel="stylesheet" href="styles.4e229bd3f.css" />
  </head>
  <body>
    <script
      type="text/javascript"
      src="runtime.7c545.js"
    ></script>
    <script
      type="text/javascript"
      src="polyfills.be8befaa.js"
    ></script>
    <script
      type="text/javascript"
      src="scripts.859e944f8beb.js"
    ></script>
    <script
      type="text/javascript"
      src="main.29d35dbd.js"
    ></script>
  </body>
</html>
```

An SPA application's build produces static files; therefore, our application can be easily served with CDN-based solutions.

Here, the CSR rendering process can be seen.



CSR

Steps:

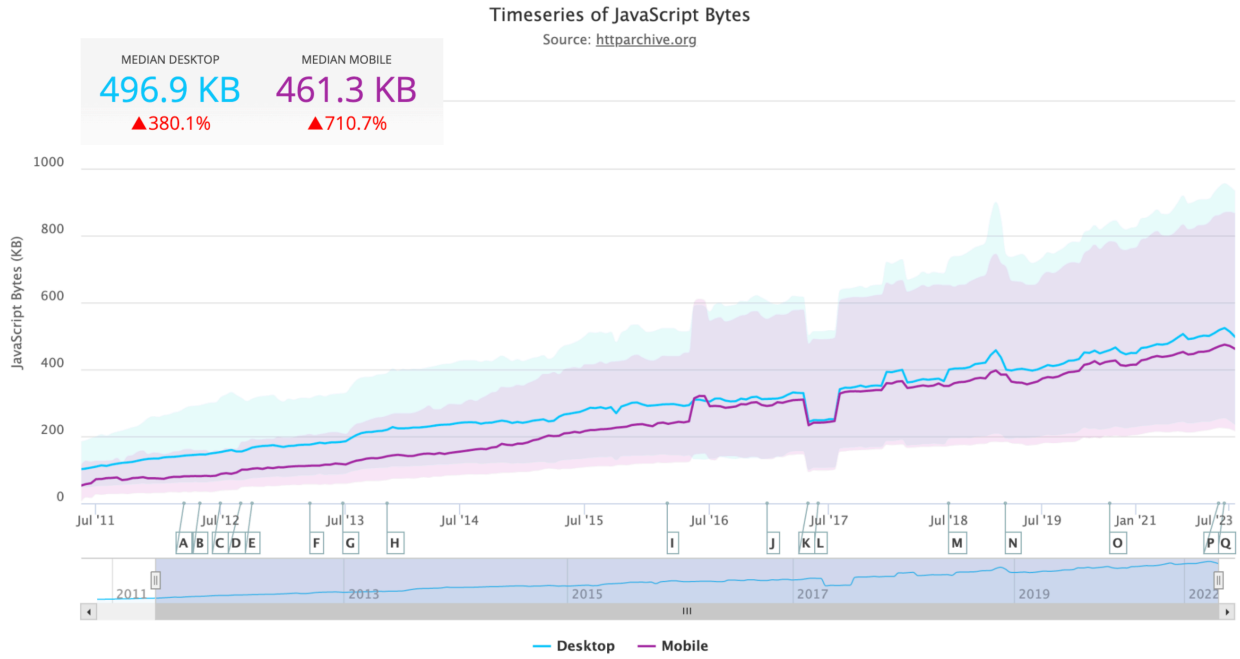
- Download the HTML
- Via the links in the HTML, download the assets and the JavaScript
- Run JavaScript
- Render the application
- Finally, the user can click

With this approach, the application will immediately download everything it needs, and all the logic will be available in advance, but this type of process has limitations that must be taken into account. The first limit is related to performance because the application will be downloaded entirely into the browser's memory, and therefore, the initial waiting times, before being able to interact with the platform, will require a few seconds of waiting for larger applications. When downloading the app, the user will only see a blank page until it is fully loaded, not the best user experience. Frameworks have introduced a lazy loading feature over the years to avoid this problem. Let's imagine an application needs to be optimized as a bundle because there are a lot of features and js to load. As developers, the code can be modified to isolate a certain part at the module level; for example, the user profile section can be isolated to load it only if the user wants to access that section.

> This optimization is left to the developer, but at least it allows us to limit a known CSR problem.

On the other hand, having all the application code in memory, however, ensures that navigation through the application paths can be done very quickly and fluidly. For example, a global state of the application, with Redux[4] can also always be kept in memory because the page is never reloaded. After all, routing is client-side.

[4] https://redux.js.org/



timeseries_js

This graph shows the average amount of Javascript present on the page over time, and it can be seen that in a few years, the growth has been considerable. The biggest problem is the devices and conditions of the users. They generally do not keep up with this trend. Every year, there are new and more powerful devices, but the dominant trend for devices worldwide is to be cheaper, not better performing. Precisely for this reason, there has been a paradigm shift in developing medium/large-sized applications to the `Multi page application` approach.

Another factor to consider is whether or not the site needs to be indexed by search engines. By their nature, SPAs are not easily indexable because, as mentioned above, they have very basic initial HTML, and all the logic is in the JavaScript bundle. The crawler, usually in a hurry (has a time limit for each indexing), will have to execute the JavaScript to render the application. Since this is an expensive operation in terms of time and bandwidth, an optimal result will not be achieved. To overcome this problem, many companies provide different content based on the user agent. If it is recognized that the request comes from the crawler, the content is returned with static HTML; on the other hand, if it's a normal request, the standard process is followed. Therefore, if the goal is to create an easily indexable web application, the CSR approach is to be avoided.

Multi Page Application (MPA)

The multi-page application architecture can be used to solve the problems of SEO and slow loading with white pages. In fact, with this approach, it's a return to the beginnings of web applications where both Client and Server are constantly communicating to deliver pages as quickly as possible. Respond with HTML pages rich in SEO content and eliminate that annoying blank page that characterizes the CSR approach when the entire application is loaded into memory. The code executed by the server and the code executed on the client side can also be split. The database can then be queried securely on the server side using secrets such

as connecting to external databases or services. The frameworks that allow for the use of the MPA approach guarantee that the server-side code will never be sent to the browser to guarantee its secrecy.

- Static Site Generation (SSG)

In this type of approach, the application is developed normally, but the output of the build procedure produces static files that can be statically hosted because each time an app is built, a series of pages are created. These files are then served to users, meaning the server does not have to do additional work when a user visits the website. The pages can then be served via CDN and have further control over the entire flow.

> A content delivery network (CDN) is a group of servers distributed across multiple geographic areas that accelerates the delivery of web content closer to users' geographic locations. Data centers around the world use caching, a process that temporarily stores copies of files to help you access content on the Internet more quickly from a web-enabled device or browser via a server near you. CDNs cache content close to your physical location. This content is static and can be images, videos or, more generally, any type of file. Imagine that a CDN is like a supermarket; having a supermarket on every corner makes shopping faster and more efficient and also reduces the possibility of queuing at the checkout.

This approach is great for optimizing the response time of the application. It is easy to host because the application is a set of static files, but there is a disadvantage regarding the data used. Let's imagine there is a page with a list of users. If static generation is performed, the users present in the list will be those available during compilation. Once the page's HTML has been produced and written to a static file, it is immutable. So, if two more users are added to the database, they will not be present unless a new compilation is launched. A very frequent use case of this architecture is blogs. Once the page with the article has been produced, it will hardly change over time. Therefore, there is extreme ease of implementation and excellent scalability with CDN. However, it is known that if the data changes quickly over time, a new build would have to be launched to produce the new updated files. In this case, it is not the best; there is certainly something better.

- Server-side rendering (SSR)

Server-side rendering involves generating HTML for each page on the server when a user requests it. Also, with this way of writing applications, the server and client are normally developed by dividing what the server is and what the client is. Rendering happens at the time of the request, and everything happens on the server side and never runs in the browser. Therefore, unlike SSG, where the page is already rendered on the server waiting to be served to the client, SSR renders the page on the server after receiving a request. SSR is ideal for websites with dynamic or personalized content that changes frequently, such as e-commerce websites or social media platforms. So with SSR, there is an alternative to SSG, and rapidly changing data is resolved very well. In terms of costs, an up-and-running server is needed, or Lambda functions need to be used, so compared to the static approach, higher costs than a static solution will have to be incurred.

> Lambda Function is a cloud computing service that, without the need to configure servers or network components, allows you to execute code in response to events which, in the case of web applications, are HTTP requests. In practice, every time we request a page with an HTTP request, a Lambda function will be called which will return the requested page.

Thanks to this approach we can have several advantages:

- reduced costs: because payment will only be for the actual execution time of the code and only when it is executed
- serverless: it is not necessary to have an always active server because everything will be based on cloud infrastructure
- autoscaling: automatic instantiation of resources based on requests received

With the MPA approach, it should be known that the pages that are sent to the browser are rich in content, faithfully reflect the page of the application, and can contain the necessary metadata to be indexed correctly without any difficulty. However, these pages are HTML code with no interactivity, and therefore, a hydration process is required to make them responsive to user events (e.g., clicking a button). The `Hydration` process will be looked at in a while.

- Incremental Static Regeneration (ISR)

This technique was created to overcome the slowness of the real-time generation of the pages in SSR mode because it allows for defining a revalidation period for each page, indicating the frequency with which the page must be regenerated in the background. A page is then generated, which will be used to respond to requests quickly; behind the scenes, the server will prepare the updated version, which will then replace the previous one in the cache. The advantages of this approach are the possibility of updating pages dynamically by finding the right balance between response times and dynamic data.

SEO also benefits from this because the pages are easily indexable by search engines so excellent indexing can be guaranteed. The developer's experience is good because it is usually to carry out this type of configuration. Specific APIs facilitate this type of setting so that the focus can be on the functions and not on the configuration. The server management costs also benefit because the resources used for each request are optimized.

As in all choices, there are compromises; in fact, invalidating the cache requires correct management, and whoever develops the application must carefully evaluate how to invalidate the cache if necessary because otherwise, there is a risk of providing content that is not updated correctly. Another thing to take into consideration is the load of the server; if many pages require frequent regeneration, then there could be an overload of resources on the server side; at this point, it would be better to opt for a classic SSR instead of the ISR which as it has been understood is a cross between SSG and SSR.

Hydration

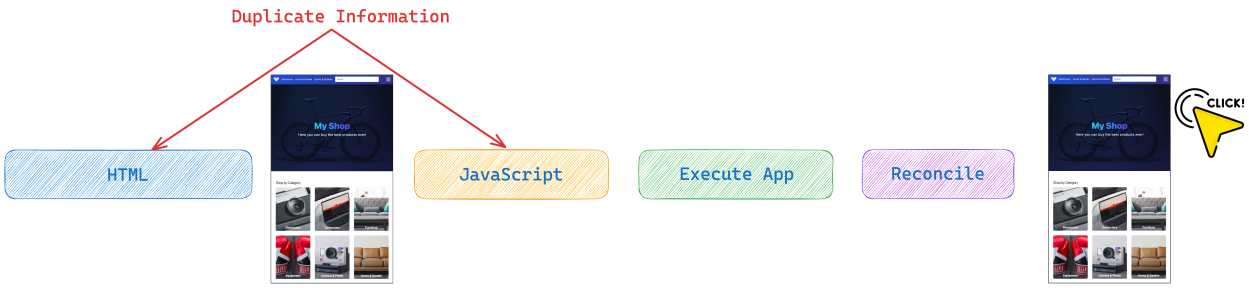
Hydration is a process that occurs on the client side and is linked to the generation of pages on the server side. It is used to recreate the state of the application (e.g., the state of the open/closed menu) and the interactivity of the HTML rendered by the server (e.g., the listener of events on the buttons to make them interactive). The server generates the page server-side to support SEO and sends this page in HTML and CSS to the browser. Then, an additional process, Hydration, is needed to add the application state.

To carry out the Hydration process, all input information is needed, such as the data displayed on the page rather than the number of open orders that need to be processed. This type of behavior is easy to notice

because, in the Network tab of the browser, data being sent from the server to the client can be seen to ensure this mechanism. This creates network overhead just to perform this state reconciliation process, which is a problem and a waste of resources.

Imagine a header with the title of the page and a button that opens and closes the side menu. The page sent from the server to the browser will contain the page's title and the menu's state, which will be used by the application to show whether the menu is open or closed. This gives a significant advantage for SEO. However, once the page is rendered, it will begin the hydration process, which will calculate the state of the menu and attach the listener to the menu button to allow the application to become interactive.

Here, the hydration rendering process can be seen.



Hydration

Steps:

- Immediately download the HTML with the information to show a snapshot of the application. It is larger than the one previously seen in the Single Page Application.
- Then, download the JavaScript needed to perform the hydration process.
- Run the application.
- There is the Reconciliation phase, which recreates the state.
- Finally, the application is interactive, and it can be clicked.

Note that JavaScript is used twice, once to create the HTML and the other to perform the Hydration. If the word "My Shop" is searched within the files produced by the build, this word will be found twice in the HTML and JavaScript sent to the browser because all the information is used to recreate the state. So, the process seems faster than a single-page application because there is faster feedback, but more time passes before interaction with the app is possible, and therefore, in the end, it's slower.

However, the end users of the application are now able to see and read the contents even if they are not completely interactive because, as mentioned, server-side rendering eliminates the problem of the blank page at startup (typical of the CSR approach) and this is all to the advantage of search engines and social media crawlers. It is possible to add some extra information to the static page sent by the server, which allows the links to preview the contents. So, imagine sending a link to an article that has been read to an acquaintance; this link, referring to a page containing this extra information, will be displayed with a preview of the article itself to offer a superior user experience.

link_with_preview

Island architecture

To overcome the costly waste of resources that hydration causes, a solution was sought by splitting and postponing the hydration process; this architecture is also called Partial hydration. The central point of this architecture is the concept of an island because the HTML pages will be rendered on the server, and only in the points where there is dynamism will an island be defined.

There can be multiple islands within the same page; these islands are like placeholders, which will then be hydrated on the client side independently, thanks to the HTML and the state that the server provides. So, in

fact, the Hydration problem is being circumvented only by splitting the process into multiple independent operations.

In general, the applications that benefit from this approach are mostly static, with some points where there is dynamism; blogs, news sites, and showcase sites are the ideal use cases for this architecture because, all in all, it does not bring disadvantages. By doing this, the static content does not require rework because it does not support events, but for example, the buttons, search bar, and menu must receive special treatment to become interactive and require the input state provided by the server for rehydration and event management.

Source: Islands Architecture: Jason Miller - https://jasonformat.com/islands-architecture/

island_architecture

This is just an example, but to get good results we need to think about how to split our app. Here, through the appropriate APIs, some frameworks allow us to render our HTML page without interactivity and to intelligently hydrate only parts of our page. Let's imagine we have a part of the page outside the initial viewport, for this section, it is possible to define an island and postpone the hydration process until it is

visible. Astro[5] for example, one of the frameworks that supports the Island architecture, offers us various directives to command Hydration at a time we see fit.

Example:

```
<BuyButton client:load />
```

- **client:load**: Based on priority, as soon as possible.
- **client:idle**: According to priority, calmly.
- **client:visible**: Based on whether they are visible or not.
- **client:media**: Based on screen size.
- **client:only**: Skip HTML server rendering and render only on the client.

> `client:only` is similar to `client:load` in that it loads, renders, and hydrates the component immediately on page load.

In the previous graph, we can see that the islands are independent silos and the borderline is clear and well delineated, but once we have all these islands on our page we will also have to share some information between islands. For example, let's think about wanting to select one or more orders from a list, and in the header we want to display the sum of all the selected orders. If to optimize the rendering we have separated these two components into islands we must find a way to make them communicate. If we want to co-share the state between islands, the solution that is used as a standard is to share the information via a state external to the islands. In this way, the external source will act as a source of truth to keep the information, but in my opinion, it is not an optimal solution. The first problem I see is that we have to use an external library to manage the state, as a second thing we are adding a layer and further boilerplate code for such a simple and certainly necessary operation in most applications. So in my opinion this architecture tries to solve the hydration problem but introduces other limitations which then fall on the shoulders of the developers who have to fight these problems.

React Server Component

An evolution of the island architecture is the introduction of React server components. This approach is specific to React and aims to limit and further optimize the hydration process by defining server and client components.

Server components

There are some advantages to using Server components because we can for example perform data retrieval securely on the server side, and communicate with tokens and API keys, without the risk of exposing them to the client. Being able to read the information on the server side certainly also guarantees us greater speed and also allows us to make server-to-server calls which overcome a whole series of problems that could be

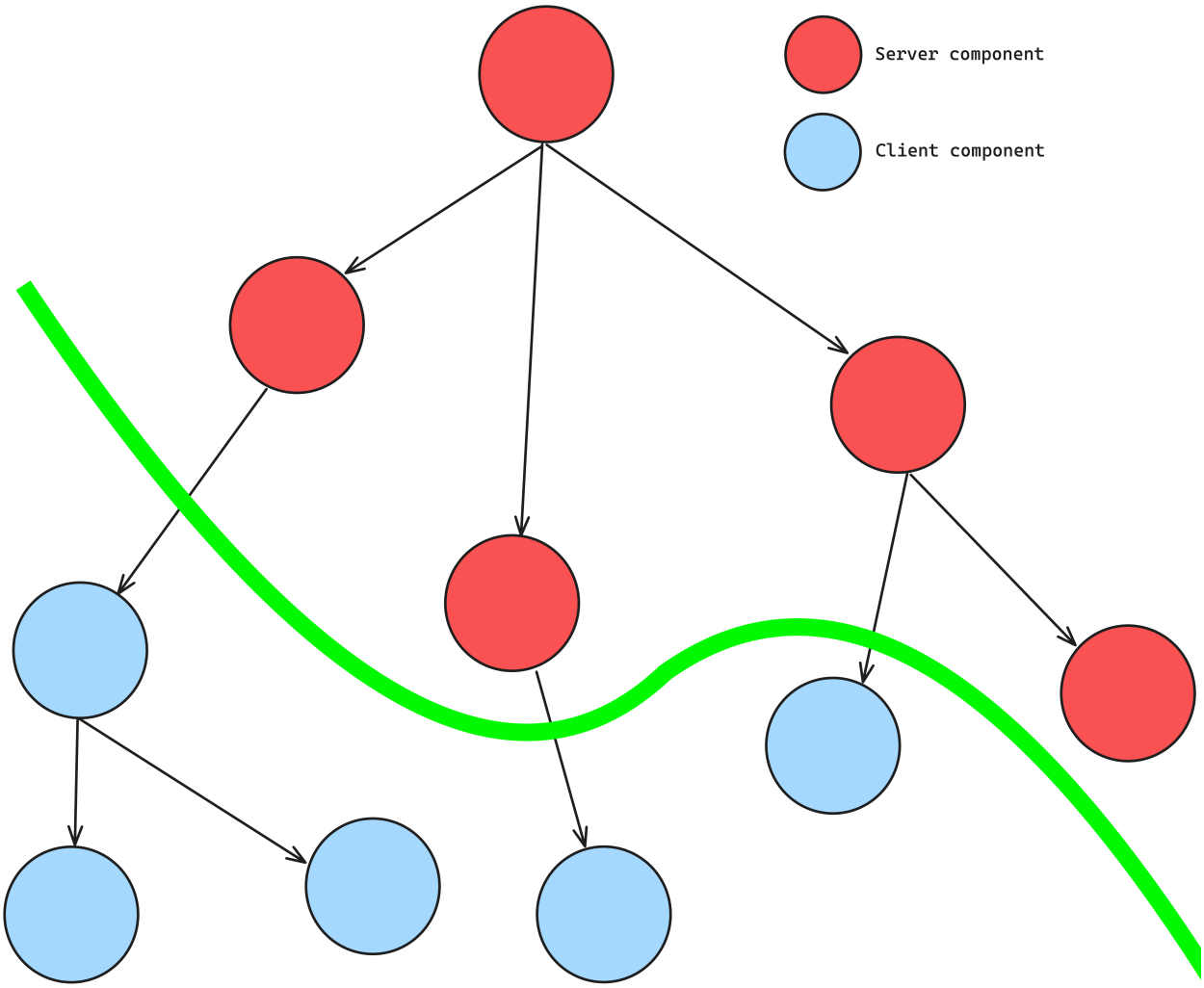
[5] https://astro.build/

encountered by making the call on the client side. Furthermore, our business logic can remain on the server without exposing it to the client, particular algorithms or discount calculations for certain customer groups can remain safe from prying eyes. The size of the Javascript bundle that the server and client exchange also benefits from this approach and all this benefits users with less powerful devices or slower internet since the client does not have to download, parse and execute any JavaScript for the server components. We are still talking about server-side rendering so our application will be optimized for search engines and will be highly shareable on social networks. As seen previously, bots can analyze the page to generate social card previews for our pages. It is also possible to perform `streaming` of our component, in fact with this technique it is possible to divide the transmission to the client of the component into blocks and transmit them as soon as they are ready. This technique benefits the user who sees the page in advance without having to wait for the entire page to be rendered on the server. For example, if a section of our page requires a database query that we know is slower, we can postpone sending the section just mentioned and render the rest of the information on the client so the end-user has visibility of the page. Next.js[6] is one of the frameworks that has implemented this architecture and has chosen to support server components by default so any new component created will be a Server component.

Client components

Compared to server components, these components are interactive and have the advantage that they can use state, effects and events because the hydration process gives them interactivity. These features allow us to update the user interface and provide immediate feedback on our applications. Another advantage is that of being able to exploit the libraries that require the browser's API, the `document`, `window` and other objects are only present in the "client world" so if the libraries we use to make use of them, client component is the choice to focus on. Furthermore, if we have to use the browser API to read/write on localstorage or use more advanced APIs such as the camera, geolocation and more, we are forced to use the client component for the reasons seen previously. Next.js[7] is one of the frameworks that has implemented this architecture and to declare Client components it is necessary to add the `use client` directive of React in the first line of each component, to instruct the framework of our intention. `use client` creates a boundary between server and client, this means that once the boundary is crossed by declaring `use client` all modules imported from it, including child components, are considered part of the client package. So once we have moved to the client side we can no longer import server components, we are forced to continue on the client side.

[6] https://nextjs.org/
[7] https://nextjs.org/



RSC

So, as developers, decisions need to be made about which components are servers and which are clients. With this type of architecture, it can be seen that the boundary between the Server Component and the Client Component is not a straight line but a curved line that can be shaped at will according to the commands. If a server component requires dynamism one day, it will be necessary to go and refactor the architecture. For example, the Footer component, which is initially static but then requires dynamism with the various evolutions because the subscription to the newsletter or the “Contact Us” section is inserted.

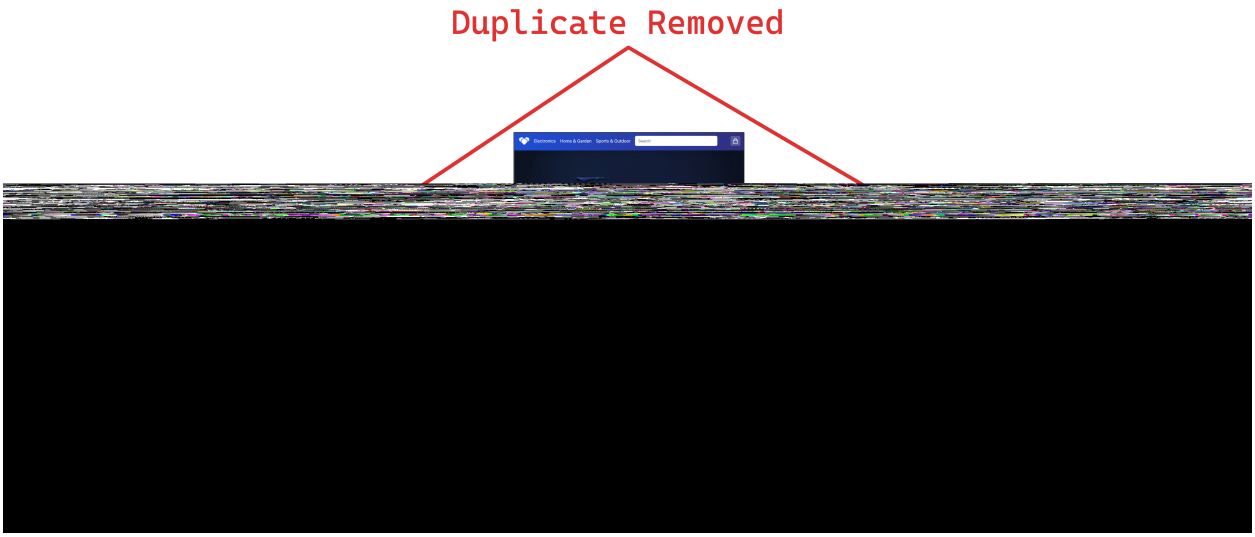
Resumability, the Qwik way

Different approaches and mechanisms have been observed to manage the rendering of applications. However, if attention has been paid to the various solutions, it will certainly have been noticed that none of them solve all the problems. There is the CSR, which has the problem of an initial bundle to download before being able to interact with the application. On the other hand, server-side rendering with all its facets: SSG, SSR, and ISR is also present. Not to mention the hydration problem common to almost all frameworks available

in the frontend landscape, with some trying to optimize it because it has been realized that it is a process greedy for resources and overheads. It was deliberately said “almost all” because Qwik was born to solve this problem once and for all with a new mental model, Resumability. This innovative approach reverses the way of thinking, starting from the events triggered by the user and not from the continuous search to recreate the application state.

Qwik is an SSR framework, but the CSR approach can also be used. In the SSR mode, the rendering always starts with a static HTML page served by the server with inline CSS and only 1KB of JavaScript injected at the last part of the HTML to ensure maximum performance. This small JavaScript code will intercept any events coming from outside (user interactions) and react accordingly, so a global event listener is attached to that.

Here, the Resumability rendering process can be seen.



Resumability

Steps:

- The HTML with the information to show a snapshot of the application is immediately downloaded. It is larger than the one previously seen in the Single Page Application. This HTML also contains some extra information, including the serialized application state.
- The page is interactive straight away, waiting for events triggered by the user. So, on click, thanks to the fact that serialized information is sent in the HTML, the framework can execute only the JavaScript necessary for that specific action to avoid the massive download of the entire JavaScript bundle upfront. There are no duplicate calculations, but better results with a cleaner process.

An excellent result can be observed! This approach will be clarified in the next chapter because this innovative way makes Qwik the framework for future generations.

Summary

This first chapter touched upon many intriguing points, and a wealth of information was shared. Attempting to summarize the shared information in these paragraphs is the next step.

The discussion began with the history of the frontend world. Evolution has been rapid in recent years; fortunately, things have evolved for the better. In fact, configuring projects has become simpler than in previous years, thanks to the advent of CLIs, command-line tools that allow for quick scaffolding of projects. Now, any modern framework with a basic application can start with one or two commands. A nostalgic recall of when each project had its personalized configuration, each process, and each task was the result of study and attempts to find the ideal solution for the infrastructure.

Moreover, transitioning from one bundle tool to another was complex because the migration guides were there, but today, many tools allow automatic updating. Significant strides have been made to make the contents easily usable on the documentation side. Consider, for example, the Qwik documentation. The search bar allows for searching within all the documentation automatically because there are tools that index the contents behind the scenes and enable this type of user experience.

But that's not all. Through AI, a question can be asked, and an answer can be waited for a few seconds, which is most of the time accurate and immediately provides the necessary solution to solve the requested problems. These AI tools are phenomenal because they learn from themselves. Every time negative feedback is provided concerning a result, they manage to learn and improve day after day.

The various frontend architectures and the types of rendering were analyzed continuing in the chapter. Each choice has its pros and cons because, as with all choices, the perfect solution does not exist. It is always a compromise between pros and cons. All the approaches examined together leave a lot of responsibility to the developer, who will have to decide which parts of the application to lazy load in the case of SPA.

The Hydration process was analyzed, and it was observed that different approaches were born to limit this poorly performing process over time. If it had been optimal, other ways to perform it better would not have arisen. It was seen that Island Architecture tries to find a solution while leaving the decision to the developers on how to divide the application to limit the Hydration problem.

The same thing can be considered regarding the React Server Components. The developer has the burden of deciding upfront which are the server and client components. These are important decisions that changing during construction could be costly regarding refactoring.

At the end of the chapter, it was mentioned that things are different with Qwik. The framework optimally solves these problems. Furthermore, a preview of the next chapter can be given, and it can be said that the Developer Experience also takes the weight off the developer's shoulders on making the reasoning seen previously. Therefore, it removes the headaches of having to optimize the application because it is done by default for them by the framework.

As a premise, it can be said that it doesn't seem bad at all. So, there is no point in waiting. Let's continue straight to the next chapter!

Qwik, the framework built on top of closure-extraction

Qwik, the Framework Built on Top of Closure-Extraction

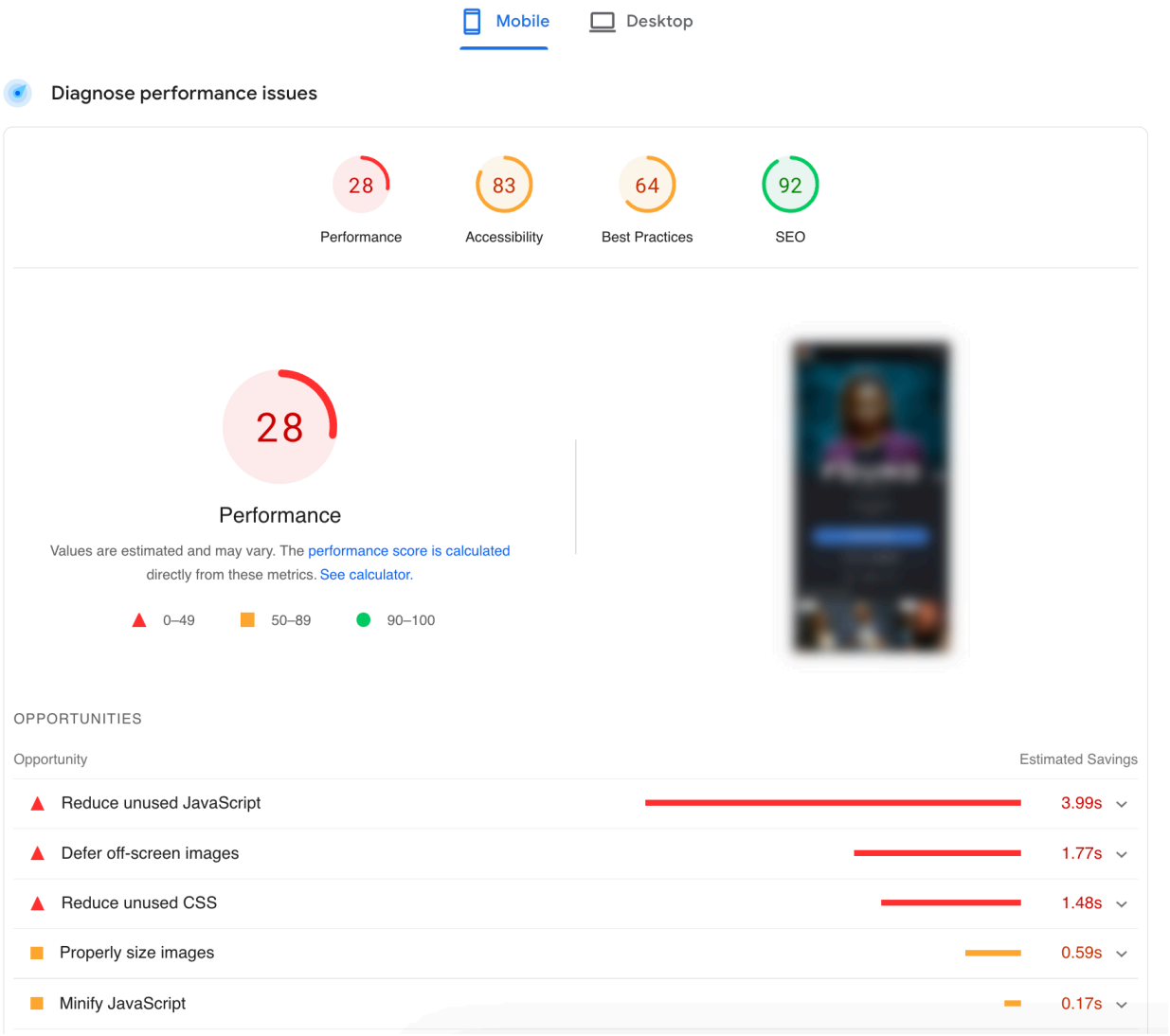
In the previous chapter, it was learned that Qwik is unique compared to other frameworks. This framework can send the browser a static page consisting only of HTML and CSS and 1KB of JavaScript. This almost magically makes the application interactive immediately upon startup. However, it's worth delving a little deeper into this because, behind every magic trick, there's a trick. With modern connections, a page containing HTML and CSS is undoubtedly efficient and speedy. It would be the ideal situation, but unfortunately, interactivity is needed, which requires JavaScript.

JavaScript was pointed out as it seems to be the weak point in creating quality applications. This can be analyzed using pagespeed[8], a reference site offered by Google to analyze web applications. Google has been actively working for years to raise awareness among developers about optimizing web applications to improve the ecosystem and make the web a better place. Google, also the developer of the Chrome browser, has created a series of tools and DevTools to help developers improve their applications and code.

The results of a completely random site can be analyzed by focusing on pagespeed. A screenshot of a site chosen at random will be provided, but any favorite site can be tested and the results compared. Just insert the site in this URL: https://pagespeed.web.dev/analysis?url=‘your-website-here‘ to start the analysis. After a few seconds, the result will be available.

Here is the report:

[8] https://pagespeed.web.dev/



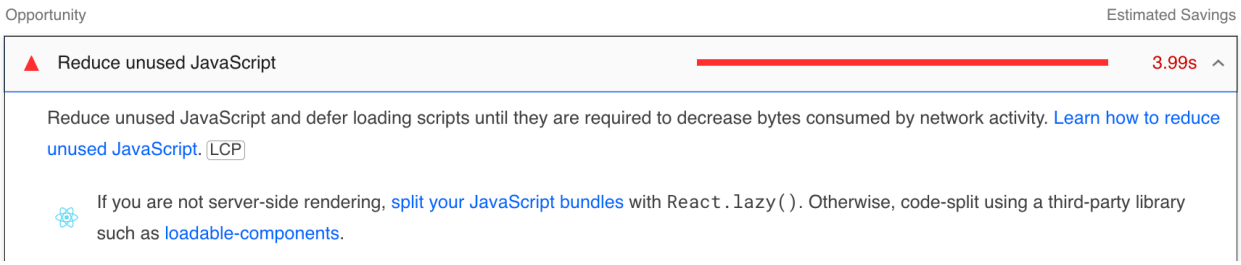
page-speed-results

The analyses for mobile devices, which usually have lower results than desktop ones, are being looked at. All references to the site have been deliberately removed, and the image showing the site preview has been blurred to avoid pointing fingers. However, it can be revealed that the site belongs to a very famous American television company. The results are likely no better unless the check was run on a site where Qwik is present, for example, qwik.dev[9]. It is suggested that this test be run to see the differences.

In the first part of the results, there are several things regarding the Core Web Vitals that were deliberately not included in the image because these metrics will be analyzed in the following chapters. For now, the focus is on the advice that the analysis offers. Specifically, what seems to be the biggest problem: “Reduce unused JavaScript”, was wanted to be discussed.

The introductory theme of this chapter, dear JavaScript, which creates slowness in applications, is back. If the advice is clicked on to read further details, this description can be seen.

[9] https://pagespeed.web.dev/analysis?url=https://qwik.dev/



reduce-javascript

According to this advice, the JavaScript must be reduced and, if possible, use `React.lazy()`. The tool has analyzed the page, understood the technology used, and is giving targeted advice on the framework. But how can the JavaScript sent be limited if the framework itself requires it to run Hydration or the processes seen in the previous chapter? How can developers be blamed for the inefficiency caused by frameworks?

Methods can be invented to limit the problem, but don't worry; the problem is not with the developers. It is hoped that this comparison has provided reassurance in this regard.

Resumability

As previously mentioned, Qwik does not send JavaScript at startup; it only sends HTML and CSS. Upon launching the analysis on the site qwik.dev[10], the enormous differences with other approaches to the rendering process become apparent. To be honest, as has been previously stated, only 1KB of Javascript is sent. This script is inserted at the end of the page because the browser parses the HTML top-down and tries to resolve the various links, scripts, rendering of the HTML, and all the other tasks that the browser is asked to perform. This script is called qwikloader[11] and is recognizable because, by inspecting the page, it can be seen.

Qwik Loader

Upon analyzing the page of a Qwik application (Chrome Dev Tools –> Element Tab), the source code can be seen, and scrolling to the bottom of the page reveals this script.

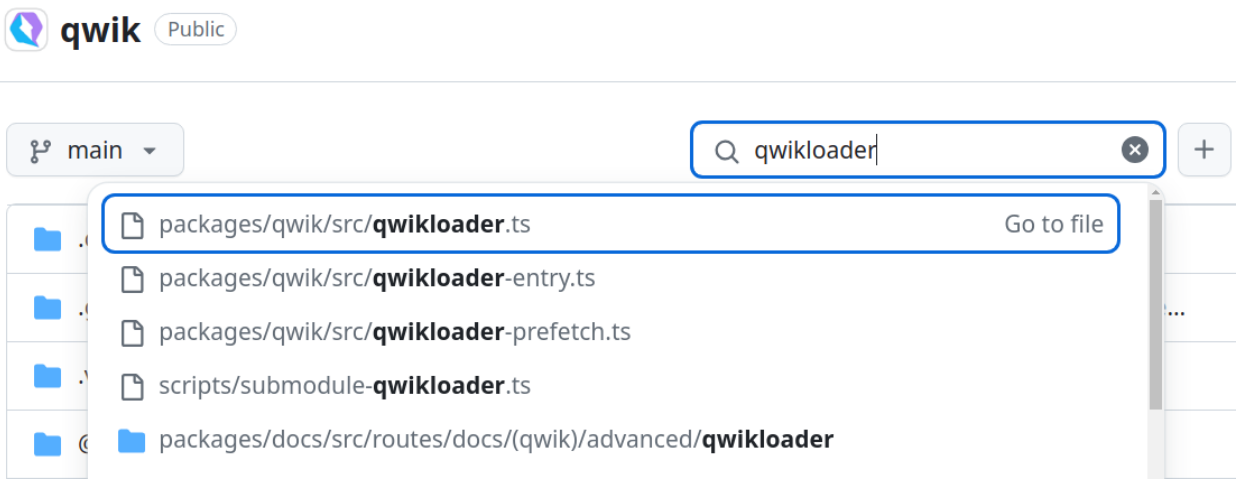
```
<script id="qwikloader">
((doc) => {
  [...]
})(document);
</script>
```

Inside this script, there is vanilla JavaScript code that contains a function IIFE[12], which, by its nature, is immediately invoked. It is used to make the application immediately ready to be interactive, and it knows how to download the rest of the application on an as-needed basis. Here's the magic trick that was being sought, but this is just the tip of the iceberg. Let's go deeper.

[10] https://pagespeed.web.dev/analysis?url=https://qwik.dev/
[11] https://qwik.dev/docs/advanced/qwikloader/
[12] https://developer.mozilla.org/en-US/docs/Glossary/IIFE

> IIFE (Immediately Invoked Function Expression): It is a design pattern that is also known as a Self-Executing Anonymous Function and contains two major parts:
>
> 1. The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
> 2. The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.
>
> Source: MDN Web Docs Glossary[13]

Ok, the code is minified, and it's not clear what it does, but by analyzing this code, it will be possible to fully understand the magic trick that Qwik is capable of doing. Well, there are at least two paths that can be taken. The simplest is to go to the Qwik repository and search for the file that declares this famous script called `qwikloader` within the framework. The project is open-source, so all the material needed to search and find this file is available. Starting from the root of the project, it's possible to simply try luck and search for a file called qwikloader; let's see if today is a lucky day.

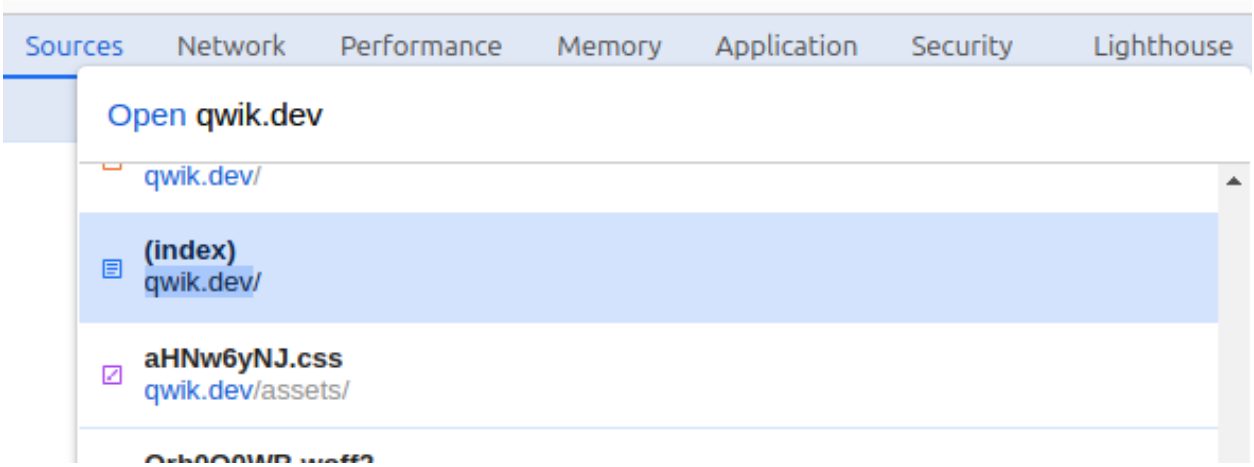


qwikloader

Fortunately, now and then things go in the right direction and in fact, the first file is exactly what we need. We can read the code and try to understand what is being executed, we have more readable variables because the code here is not minified, we also have TypeScript which helps us understand the code better, but we cannot see the code in action and therefore it's difficult to follow the flow of the code and try to imagine how it will work in the browser. So we can try the other method that immediately comes to mind, let's try to insert a breakpoint in our browser and with the Chrome DevTools we go forward step by step to try to understand in broad terms what this piece of code does.

The same can be done to actively follow the debugging in the Qwik documentation site[14]. First, it's necessary to open the DevTools; it's possible to search for `qwik.dev` and look for the element with `(index)` using the "Open file" function.

[13] https://developer.mozilla.org/en-US/docs/Glossary/IIFE
[14] https://qwik.dev/



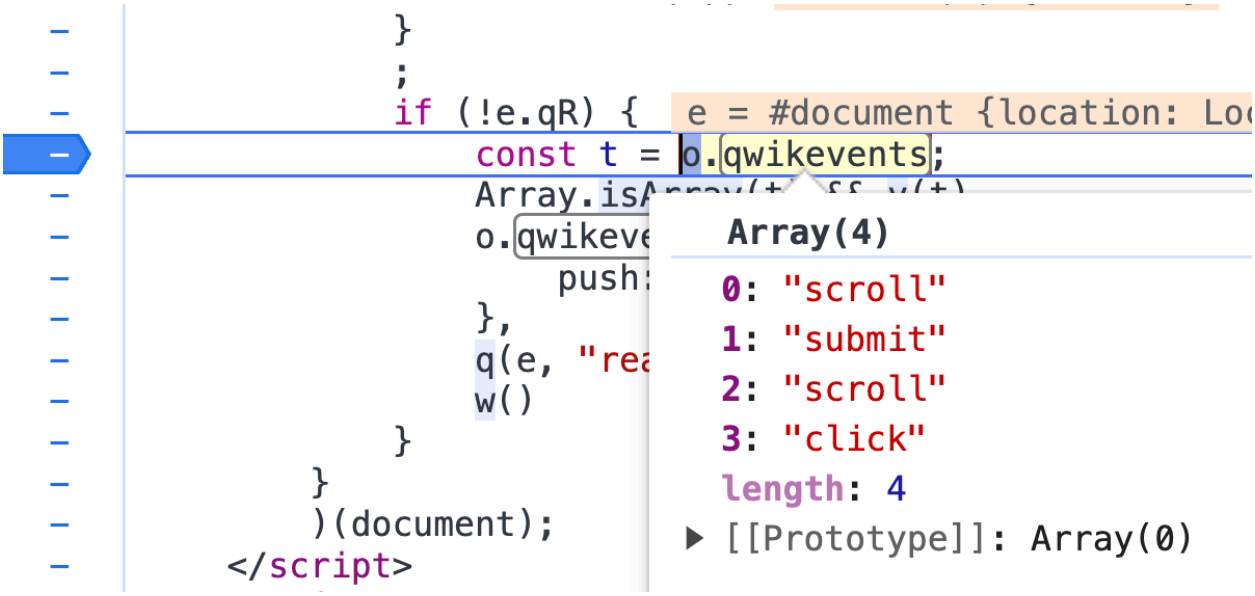
1_devtool

Once the element is found, it can be selected, and now all that's left is to search for the string `id="qwikloader"` to finally be able to insert a breakpoint on the code so live debugging can be performed.

2_devtool

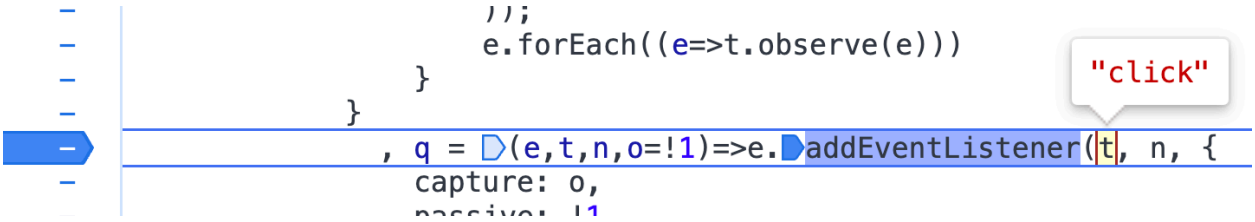
Let's refresh the page. Now, everything is ready to investigate the code. By inspecting the code, this point of code can be debugged.

```
1  const t = o.qwikevents;
```



3_devtool

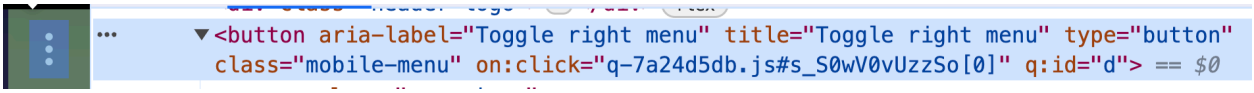
Here, it can be seen that Qwik knows in advance which events can be executed within the page because some logic has been executed on the server side to create the page. It then takes advantage of the work done on the server side and passes the information into the browser to create a list of events. This list is then used to create event listeners for both the `window` and `document` objects.



4_devtool

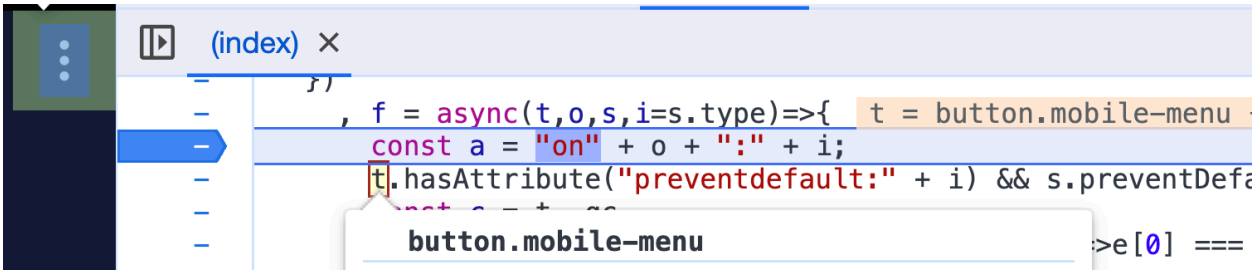
Excellent, so it's possible to capture the events that happen on the page. So now, any user input is being waited for. But how do you understand where it was clicked? If the HTML is analyzed, it will be possible to find special attributes that Qwik adds to the HTML during the build phase. It might be thought that it will weigh everything down, but the weight is negligible compared to other traditional methods that download all the JavaScript upfront.

It can be seen that in the "Menu" button, there is a non-standard attribute `on:click="....."`.



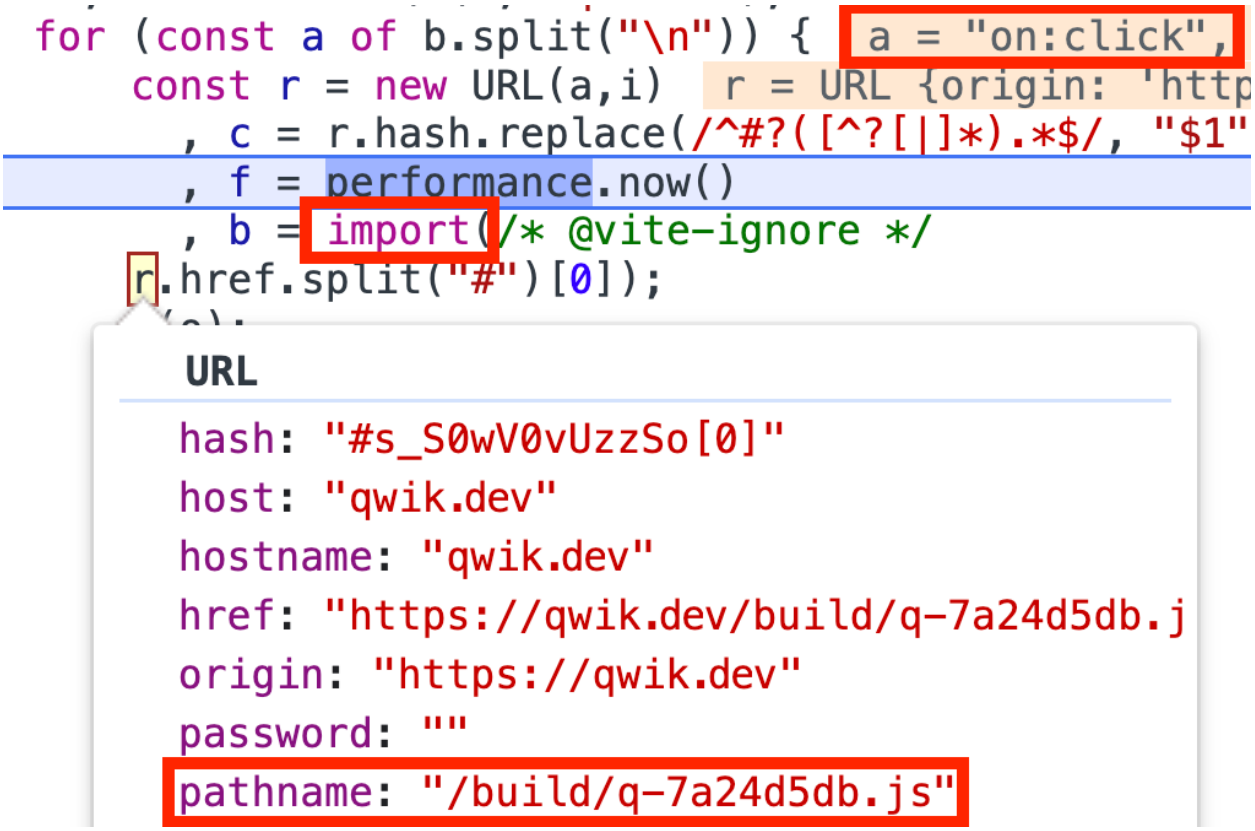
on_click

Let's place an additional breakpoint on this line of code `const a = "on" + o + ":" + i;` and click on the "Menu" button. Here, it can be seen that the variable `t` is the target of the event. So, the user event was captured via the global event listener, and now some logic is being executed managing to reference the element that received the user's action.



on_event

If the debugging is continued, it can be seen that the value of the on:click attribute will be read. The value of this attribute is the JavaScript file to be executed containing the logic that will animate the application.



file_import

It was purposely said to execute because the import (see image) will perform an HTTP call to obtain that specific file path, but before downloading the file from the server, it will pass through a service worker that is instantiated at application startup to preload all possible JavaScript files that the application can use. To leave the browser's main thread free to render the application, behind the scenes, a service worker preloads the Javascript files to have them immediately available. If the network tab is looked at, confirmation of what was said can be found.

Name	Status	Type	Initiator	Size	Time	Waterfall
q-7a24d5db.js	200	script	(index):1065	(ServiceWorker)	3 ms	

service_worker

In the `size` column (ServiceWorker), it can be seen, and as execution times of the HTTP call 3ms can be seen, it is clear that it is a resource already available. The file will be executed, and the code will continue with its reasoning. It is very important to note that in that file, there is only the logic necessary to complete the execution of the click made by the user.

But who produces this division into small independent files? Qwik performs this division into files during the build phase of the application by the `Optimizer`, which will be explored in the next chapter. In fact, as developers, there's no need to worry about anything; the application is written normally like any other framework. It is necessary to take a step back and understand how one can differentiate the code that must be executed on the server and what will be executed on the client side to understand in detail how the optimizer can perform this magic.

Code Extraction

The client and server represent two very different worlds. In the code executed by the server, access to the browser APIs is not possible, and in the client one, reading files, for example, is not feasible. Therefore, a method to distinguish the two codebases and produce bundles for these two very different worlds is necessary.

> The process of separating your code and packaging the server and client parts is called code extraction.

When applications that take advantage of server-side rendering are written, server code and client code are usually mixed in the same file. This method helps to keep track of the work, prevents the proliferation of imports in the files, and guarantees a better developer experience. Therefore, the framework defines naming conventions to instruct the server on the code that is reserved for the client and what is instead for the server.

```
1  export default function Page({ data }) {
2    // Render data
3  }
4
5  // Server only code
6  export async function getServerSideProps() {
7    // Pass data to the page component via props
8    return { props: { data } };
9  }
```

A clear example of a naming convention is the code typically written with NextJS[15]. Most of the frameworks offering SSR/SSG have a naming convention mechanism, and NextJS is just one of many. The

[15] https://nextjs.org/

`getServerSideProps` function runs only on the server, while the default exported component runs on the client and in the server for the first render. Three different strategies will be presented, starting with the most basic to the most advanced one.

Export extraction

Export extraction is a way to divide server code from the client bundle by relying on the bundle tree-shaker behavior.

> `bundle` `tree-shaking` is a term commonly used in the JavaScript context for dead-code elimination. It relies on the static structure of ES2015 module syntax, e.g., import and export.

```
export default function Page() {
  // Client code
  // Is imported from other locations in the codebase
}

export const wellKnownServerFunction = () => {
  // Server code
  // There is NO reference to `wellKnownServerFunction` in the client code
};
```

The bundling process traverses the references starting at a root method recursively, and the tree-shaking behavior will do the heavy lifting work. If the code is not reachable, it is removed, and only the used code is placed in the bundle. The Page component is used in the client and will be present in the client bundle. `myServerCode` is removed instead because there is no reference to it in the client code. Assuming that `wellKnownServerFunction` is a name that the framework expects, this function can be called using reflection into the server code. So, tree-shaking helps to divide the codebase. If this behavior is ON, the result is the client bundle; otherwise, it is (OFF), and all the code that is used on the server will be present.

Export extraction is nice, but there are some drawbacks:

- `wellKnownServerFunction` is a special code reserved only for the framework; it can't be called from the client code
- only one server function per file can be included
- this approach is not type-safe; look at the example; nothing prevents this

```
export default function Page({data}: WRONG_TYPE_HERE) => {
  return <span>{data}<span>
}

export const wellKnownServerFunction = () => {
  return someData;
}
```

Function extraction

Indeed, the example just seen could be written in this way to guarantee the correctness of the types.

```
export const myDataLoader = () => {
  // SERVER CODE
  return readDataFromDatabase();
}

export default function Page() => {
  // No need for manual flow typing. This can't be incorrect.
  const data = myDataLoader();
  return <span>{data}<span>
}
```

Unfortunately, this code breaks the tree-shaking behavior. In the client code, there is a server function, and it can't be called because this code will blow up. The export extraction approach could be used, but a way to separate the server and client code is needed to solve this problem. A marker function can be used!

> A marker function is a way to mark up a piece of code for future transformations.

The previous example can be modified with a marker function called `SERVER()`

```
export const myDataLoader = SERVER(() => {
  // Server code
  return readDataFromDatabase();
});

export default function Page() => {
  // No need for manual flow typing. This can't be incorrect.
  const data = myDataLoader();
  return <span>{data}<span>
}
```

Now the SERVER() function wraps the server code, an AST[16] transform that looks for SERVER() functions and changes the code can be used. The example will be translated like this:

```
/*#__PURE__*/ SERVER_REGISTER('UNIQUE_ID', () => {
  return readDataFromDatabase();
});

export const myDataLoader = SERVER_PROXY('UNIQUE_ID');

export default function Page() => {
  const data = myDataLoader();
  return <span>{data}<span>
}
```

The AST transformation changed these things:

- SERVER() code is moved into a new top-level location.
</br> With this move, the code is tree-shakable because there is no direct call to it.
- SERVER_REGISTER() is used to wrap the SERVER() code.
</br> In this way, the framework links the moved function with SERVER_REGISTER() and its UNIQUE_ID.
- AST added a /#***PURE***/ annotation to SERVER_REGISTER().
</br> /*#**PURE**_/ annotation is essential because the bundler doesn't include this code. In this way, server code can be removed from the client bundle.
- The call to the SERVER() function is transformed to SERVER_PROXY() with its UNIQUE_ID.
</br> SERVER_PROXY() acts as a bridge between server and client worlds.

So far, so good. The server code can now be called from the client, and a type-safe flow is achieved.

> Our AST transform to only recognize SERVER(), but no one forbids us to say: "Please AST do the code transformation for me whenever you find a function suffixed with $ (as in___$())"

Lazy load code

If a function that lazy loads code is desired, this solution needs to be improved.

Let's take this example:

[16] https://en.wikipedia.org/wiki/Abstract_syntax_tree

```
import {lazyLoad$} from 'my-cool-framework';
import {invokeLazyCode} from 'someplace';

export function() {
  return (
    <button onClick={async () => lazyLoad$(() => invokeLazyCode())}>
     click
   </button>
  );
}
```

The tree-shaker, after the AST transformation, will remove the code inside `lazyLoad$()`; a few things in the approach need to be changed. The code can be moved to a separate file instead of transforming it as /#***PURE***/ for tree-shaking.

FILE: `file_with_unique_id.js`

```
import { invokeLazyCode } from "someplace";

export const id123 = () => invokeLazyCode();
```

FILE: original file

```
import {lazyProxy} from 'my-cool-framework';

export function() {
  return (
    <button onClick={async () => lazyProxy('./file_with_unique_id.js', 'id123')}>
     click
   </button>
  );
}
```

With this approach, /#***PURE***/ annotation is not needed at all, and the `lazyProxy()` function can load the code because it's in a different file. Now, the runtime can load the function on the server if needed, so the `__Register()` function transformation can be removed.

Closure extraction

Marker functions are awesome, but there is a limitation; let's take this example.

```
import {lazyLoad$} from 'my-cool-framework';
import {invokeLazyCode} from 'someplace';

export function() {
  // useStore declares a scoped state
  const [state] = useStore();
  return (
    <button onClick={async () => lazyLoad$(() => invokeLazyCode(state))}>
     click
   </button>
  );
}
```

The problem is that the `lazy$(() => invokeLazyCode(state))` closes over the state. So when it gets extracted into a new file, it creates an unresolved reference.

```
import {inovkeLayzCode} from 'someplace';

export id123 = () => invokeLazyCode(state); // ERROR: `state` undefined
```

Hold on, this problem can be solved! Let's have a look at this AST transformation.

FILE: `file_with_unique_id.js`

```
import { invokeLazyCode } from "someplace";
import { lazyLexicalScope } from "my-cool-framework";

export const id123 = () => {
  const [state] = lazyLexicalScope(); // <==== IMPORTANT BIT
  invokeLazyCode(state);
};
```

FILE: original file

```
import {lazyProxy} from 'my-cool-framework';

export function() {
  return (
    <button onClick={async () => lazyProxy('./file_with_unique_id.js', 'id123', [sta\
te])}>
     click
   </button>
  );
}
```

The compiler extracts the closure and takes note of the parameters; the `lazyProxy()` call has missing variables in the same order. e.g., [state] On the other side, `lazyLexicalScope()` restores the variables needed from the closure.

> Thanks to the previous changes, we can refer to variables from a different location, and we can now lazy load closures!

Server/client code is already being mixed into a single file in existing technologies using the export extraction model, but the solutions are limited. The new frontier of technologies allows the code to be mixed even more through function extraction and closure extraction.

The Qwik solution

So far, so good; lots of new concepts have been seen, and it's likely that the code will need to be reread more than a few times to understand all the steps. Well, then, let's take the last example regarding "Closure extraction". Qwik takes this concept to the extreme and can perform lazy loading, lazy execution, and server/client code-mixing with a truly sensational developer experience.

Let's look at this example:

```
export const useMyData = routeLoader$(() => {
  // ALWAYS RUNS ON SERVER
  console.log("SERVER", "fetch data");
  return { msg: "hello world" };
});

export default component$(() => {
  // RUNS ON SERVER OR CLIENT AS NEEDED
  const data = useMyData();
  return (
    <>
      <div>{data.value.msg}</div>
      <button
        onClick$={async () => {
          // Always RUNS ALWAYS ON CLIENT
          const timestamp = Date.now();
          const value = await server$(() => {
            // ALWAYS RUNS ON SERVER
            console.log("SERVER", timestamp);
            return "OK";
          });
          console.log("CLIENT", value);
        }}
```

```
      >
        click
      </button>
    </>
  );
});
```

Here is a real Qwik component, `component$`, `server$`, `routeLoader$`, and `onClick$` can be seen; these are just some of the APIs made available by the framework which, in fact, with the `$` symbol is a marker to be able to extract the various functions during the build phase and benefit from code extraction. The various APIs will be detailed in the next chapters, although thanks to the comments, it's clear what the various functions do. Obviously, behind the scenes, the framework makes sure that the server-related code remains protected to safeguard API keys, secrets, and everything that is executed on the server side. It should be noted that if desired, it is also possible to separate the server functions from the client ones to keep the two worlds separate.

Why Qwik Uses JSX Syntax

If attention had been paid to the previous example, it would have been noticed that the procedure for writing a Qwik component is not at all different from that used to write a React, SolidJS, or Preact component. All these frameworks mentioned use JSX, and the Qwik team's choice to use the same system is driven by the desire to lower the learning curve of the framework and attract as many developers as possible.

This choice was made not only because most front-end developers know it but also because there are many tools and an ecosystem already available for this syntax. For example, consider the various linters for the code, the most famous of which is Prettier[17], which already offers support for JSX. There was no need to adapt or create extra features to support Qwik; it already worked by default because the syntax is the same as other frameworks that have already been widely adopted. Hence, if this syntax is already known, it will not be difficult to start writing the first component.

But what is JSX?

JSX is an HTML-like syntax that was seen in the example of a real Qwik component, and it is very useful because it adds dynamic expressions that allow referencing variables and functions within the HTML using the `{ }` syntax. This allows writing HTML-like code inside JavaScript.

Here's an example:

[17] https://prettier.io/

```
export default component$(() => {
  // Logic and style should be implemented here

  return (
    <button
      class="..."
      onClick$={() => {
        console.log("Hi!");
      }}
    >
      Greetings
    </button>
  );
});
```

In this example we have written HTML, we see the `<button>` element, and it is easy to understand that with this syntax we can create our Qwik application in a very similar way to what we could do by writing Standard HTML. Like normal HTML elements, it is also possible to use standard HTML attributes in JSX, so we can use `class` to style our application, `aria-label` for accessibility or just to give another example we can use `disabled` to disable a particular button that we don't want the user to click. But not all frameworks use JSX there are other templating systems in the frontend ecosystem and to make you understand what I'm talking about I want to give you some examples, React with JSX, Vue.js, and Angular with a different templating system. Let's try to compare the same component seen before with Qwik but with the syntax of React, Vue.js, and Angular.

- React

Here is the example with React:

```
export function App() {
  // your logic and style here

  return (
    <button
      className="..."
      onClick={() => {
        console.log("Hi!");
      }}
    >
      Greetings
    </button>
  );
}
```

Here is the component written in React. No, there was no mistake with the copy/paste; it's so similar to a Qwik component. The only differences that may be noticed are `className` instead of `class`, `onClick` instead of `onClick$`, and in the case of Qwik, there is the addition of `component$`. Having seen before that the `$` is the famous marker that makes it possible to divide the application into bundles, all this makes a lot of sense. It can be seen that refactoring from React to Qwik is very easy. It will take some practice, but all the React or Next.js APIs are also present in Qwik, with a different mental model and performance. So once it is understood which Qwik APIs to use to replace the existing ones, the game is done; all that remains is to have fun and appreciate the benefits that Qwik will provide.

- Vue.js

Here is the Vue.js example:

```
<script setup >
  // Implement logic here
<script>

<template>
  <button class="..." @click="()=> console.log('Hi!')">
    Greetings
  </button>
</template>

<style>
  // Implement style here
</style>
```

This example recreates the same behavior, a button that prints `Hi!` in the browser console. In Vue.js, it is possible to declare the logic (script section), the template, the "HTML", and the style (style section) in the same file. As can be seen in this example, which is deliberately basic to only focus on the template differences, there is a more verbose file compared to the JSX syntax, but in any case, it can benefit from colocation because all the code relating to the component is in the same position. This helps the developer because there is no need to jump from one file to another to implement a feature or simply read the code. But if this type of approach is not preferred, in Vue.js there is the possibility of splitting the various sections seen previously into multiple files, and this is possible thanks to this system.

```
<template src="./template.html"></template>
<style src="./style.css"></style>
<script src="./script.js"></script>
```

So, those who prefer to have a clear separation between files can be satisfied. This approach, however, is not what the documentation proposes as the ideal choice, although all the functionality of the framework is guaranteed.

- Angular

Let's move on to Angular and see an example:

FILE: `app.component.ts`

```
@Component({
  [...]
  templateUrl: "./app.component.html",
  // your style in this file
  styleUrls: ["./app.component.css"],
})
export class AppComponent {
  onClick() {
    console.log("Hi!");
  }
}
```

FILE: `app.component.html`

```
<button (click)="onClick()">Greetings</button>
```

Here, too, the button that prints `Hi!` in the console was created. Unlike Qwik and Vue.js, in Angular, the approach that the official documentation sponsors is to have more files for greater separation of responsibilities and less code coupling. This means that it is not possible to insert the console.log into the template part directly; a function had to be created to perform this simple action to accommodate the separation of responsibility between UI and business logic that the framework imposes. Of course, being able to declare an inline function inside the HTML would have been useful in this case. But if it is looked at with a broader focus, it can be said that no Angular developer would ever expect to have an isolated function in the template but should immediately check the file that is responsible for the business logic. It can be seen that there is the logic file and the one relating to HTML; the snippet of the style file was omitted because it is not needed for reasoning purposes. If the template and style are preferred to be in the same file, the framework-specific syntax can be used to take advantage of colocation. Honestly, having all the code for a component inside the same file is preferred because it is remembered that in the past, one of the most expensive things during development was having to jump from one file to another to implement even the simplest of changes.

Summary

This chapter is packed with new concepts. It begins by discussing a problem that plagues many web applications. Upon inspecting the results, it becomes apparent that JavaScript is the root of the issues. Unfortunately, it's not possible to simply do away with it. The chapter then proceeds to analyze, step by step, how Qwik manages to outperform other frameworks without resorting to unusual mechanisms. Instead,

it uses a well-thought-out process to address the issues that other frameworks struggle with. The chapter delves into step-by-step debugging, and it's hoped that navigating the code broadens the understanding and provides a more detailed insight into how the framework operates behind the scenes. Some highly advanced mechanisms for code extraction are explored to comprehend how to segment code during the build phase. The solution continues to be refined, delving deeper and deeper until the optimal solution used by Qwik is approached.

A Qwik component is examined in detail, and how it's possible to write code on both the client and server-side in a seamless and enjoyable manner is discussed. In the final part of the chapter, different template systems are compared, focusing on the same example but implemented in different frameworks. React mirrors Qwik, while Vue.js begins to show a more pronounced difference, largely because its framework template involves using layers to separate logic, template, and style. Angular, in contrast, ensures a clear separation of file responsibility by default, using a different file for each layer (logic, template, and style). This highlights that not all frameworks are created equal.

In the next chapter, an exploration of how to start developing with Qwik will be undertaken. The project configuration will be examined, and the unique features of the framework will be delved into.

Links

- Code extraction article[18]

[18] https://www.builder.io/blog/wtf-is-code-extraction#export-extraction

Getting started with Qwik

Getting started with Qwik

Before beginning the process of building a Qwik application and analyzing its configuration in detail, several concepts need to be introduced to ensure everyone is on the same page. It's important to understand how the process of creating an application works behind the scenes. For developers, it involves writing a normal application, but the heavy lifting is done in the build phase by looking for the $ markers and extracting the code present in the application to isolate it into small bundles. Modern code is written with the best possible language and updated tools that allow for avoiding writing raw JavaScript code. Even for the style part, raw CSS is no longer used, but always try to find a way to speed up and make writing code enjoyable. Therefore, on the one hand, TypeScript is used for the code part and other modes for the style part, e.g., SCSS or CSS in JS.

Furthermore, when the code is stripped, the application is divided into multiple files for readability and good practice reasons. Importing from external libraries is also done to take advantage of the functionality they provide without having to reimplement the functionality from scratch. Imagine having to create an application that shows the stock market performance of the Dow Jones index in real-time with interactive graphs. A library is needed that allows for creating these graphs without much effort. If it didn't exist, it would be necessary to sit down and think about how to create these components and how to handle real-time updating with measurable variability per second. It's certainly much easier to choose the right library for your needs and take advantage of the graphics that are made available.

As good front-end developers, it is known that the browser only digests three ingredients: HTML, CSS, and JavaScript. The recipe is quite simple, but who is the chef who transforms modern and complex code into those three basic ingredients that the browser recognizes? The chef in question is called a "Bundler".

webpack

Webpack has been the reference `bundler` for front-end applications for years. Indeed, when it came out with a stable version, it was immediately adopted by many because it solved a series of configuration problems. Back then, setting up a development environment was a pain. Gulp, Grunt, and Browserify are just some of the tools that were used simultaneously in the project pipelines. If these names are recognized because they, too, were fought with, sympathy is extended. But if it's the first time these names have been heard, envy is extended because it means being of the new generation.

The role of the bundler is to take all of the code and combine it into one large file, which the browser can then use to load the application. This file contains the code plus any third-party dependencies that have been imported, which sometimes might have addictions and so on. So, at the end of the process, there are static files: HTML, CSS, and Javascript. But the big problem is that this process slows down more and more as the application grows in size. This slowness is not only noticeable during the production build phase but also on a day-to-day basis. Therefore, every time a file is saved and even the smallest of changes is made, a build will be

started, which will take time, and therefore there will be a wait. Working this way is very frustrating because, on the one hand, there is a need to produce code to complete tasks, but it can't be done because of these constant slowdowns. Luckily, in 2020, Evan You, the creator of Vue.js, created Vite, a bundler that, thanks to a new way of processing files, solves the problems that Webpack brought with it. Now, all frameworks are based on Vite, and Qwik is no different. Vite was chosen for its speed, but how is it different from Webpack?

Vite

As mentioned, Vite was invented by the creator of Vue.js and became popular because it allows for an optimal experience for developers due to its way of processing files. In fact, in 2020, browsers offered broad support for modules, and ES modules were introduced. This feature allows developers to import and export code from different files in the browser. Vite uses native modules in the browser to load and process only the code needed to request resources. Part of this process is delegated behind the scenes to Rollup.

This means there's no need to reload the entire application when a change is made, and there is no interruption to the development cycle. Thanks to Hot Module Replacement (HMR), modified modules can be replaced. While some bundlers support HMR, Vite's approach is faster, using native ESMs, so only a limited part of the dependency chain is invalidated. This means that HMR times do not increase as the complexity of the application increases.

Additionally, Vite automatically divides the pieces, determining what needs to be loaded and when. Let's imagine this code and see what Vite allows to be done. Normally, a module is imported at the beginning of the file, like the code in the following example.

```
import myFunction from "...";

const myEvent = async () => {
  myFunction();
};
```

But if there's a desire to do lazy loading of this module, a dynamic import can be performed (see example below), and Vite will create a separate file that will be downloaded when and if the `myEvent` event is executed.

```
const myEvent = async () => {
  const myFunction = await import("...");
};
```

So, thanks to this approach, it can be said to import a series of files only if the user is logged in; otherwise, don't import anything. Thanks to these tools, it is possible to optimize the code.

Vite Plugins

Vite offers an integration system to extend the build process with extra operations that are wanted to be performed. Creating a plugin for Vite is very simple. First, Vite needs to be configured and the plugin added

to the plugin property. A plugin is a simple function that can receive various options as parameters and perform operations. It can attach to different Hooks. They won't all be listed because the Vite documentation can be referred to, which is well done. A basic plugin will be created to understand how it works.

```
import { PluginOption, defineConfig } from "vite";

const myPlugin = () => {
  const screwPlugin: PluginOption = {
    name: "vite-plugin-base",
    apply: "build",
    buildStart: () => {
      console.log("buildStart: do things");
    },
    closeBundle: () => {
      console.log("closeBundle: do things");
    },
  };
  return vitePlugin;
};

export default defineConfig(() => {
  return {
    plugins: [myPlugin()],
  };
});
```

In this example, a synchronous function has been created, but nothing prevents also being able to declare something asynchronous. With this function, the `buildStart` and `closeBundle` events are listened to, and something is printed to the console. With a custom Vite plugin, everything is possible: creating files, making HTTP calls, logic during the creation phase, and more. The most frequent use case, however, is the transformation of the files of the project. For example, a particular extension can be intercepted to add a comment in the first part of the file or, more simply, modify the imports and the structure of the file to create dynamic imports on the fly and tell Vite to generate independent files. This use case seems to fit Qwik and its mental model.

Qwik compilation Process

You have seen how to create Vite plugins, and in fact, behind the scenes, at the time of compilation, optimization takes place and prepares our code to be used in the Resumability architecture. To perform this optimization, the Qwik team built `Qwik Optimizer` in Rust using SWC[19] (stands for Speedy Web Compiler), the same technology used by TurboBuild[20], to perform advanced optimizations without sacrificing build times.

[19] https://swc.rs
[20] https://turbo.build/

SWC accepts any JavaScript or TypeScript files using modern JavaScript features and outputs valid code that is supported by all major browsers; it's fast because it is Rust-based. If you were to do the same computations with JavaScript, you would run into the problem that it is single-threaded and is not a good place to do heavy computation. Additionally, the Rust and SWC communities are growing, and you can easily find support, so it's a plus. So, `Qwik Optimizer` looks for `$` and applies a transformation that extracts the expression into a loadable and importable symbol.

Is there a way to see how this transformation occurs in the Vite process? You can install a special plugin, `vite-plugin-inspect`, to be able to analyze what Vite does behind the scenes; this is also very useful when you are developing a new plugin because it is difficult to understand the build process without being able to see step by step what happens behind the scenes. To activate the plugin, it is very simple; as you saw before, just add it to the configuration as follows.

```
import Inspect from 'vite-plugin-inspect';

export default defineConfig(() => {
  return {
    plugins: [..., Inspect()],
  };
});
```

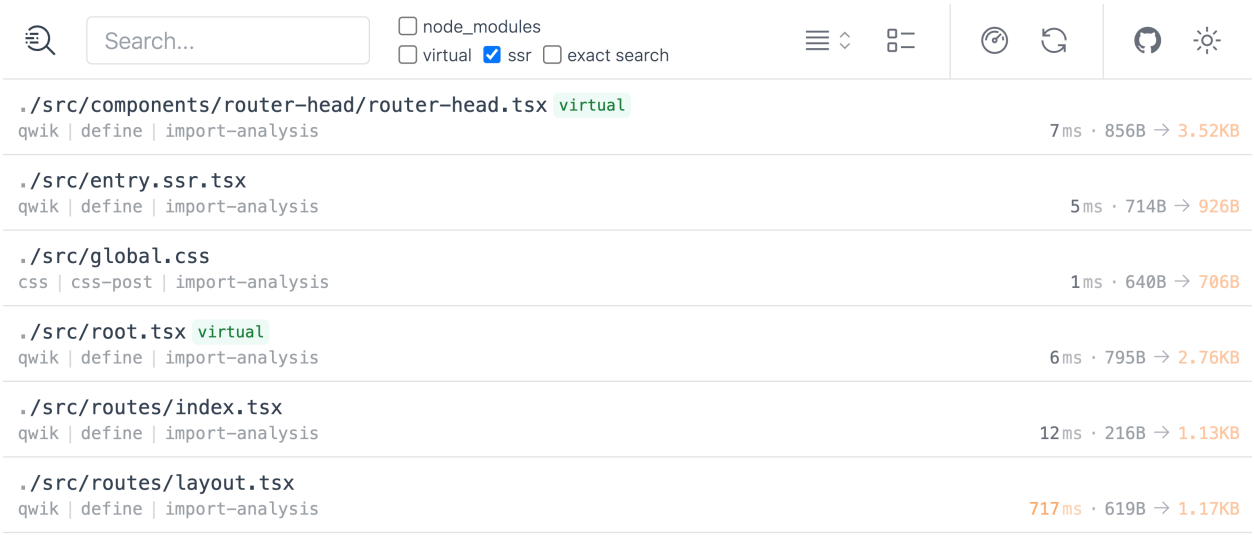
So, activate it by adding `Inspect()` inside the dependencies array. This plugin, if not configured differently, will start when you launch the `dev` process, and in addition to the application, an interactive page will also be served, usually at the address `localhost:5173/__inspect/` to inspect the modules.

Let's see the component below using this plugin and try to analyze the information it provides us.

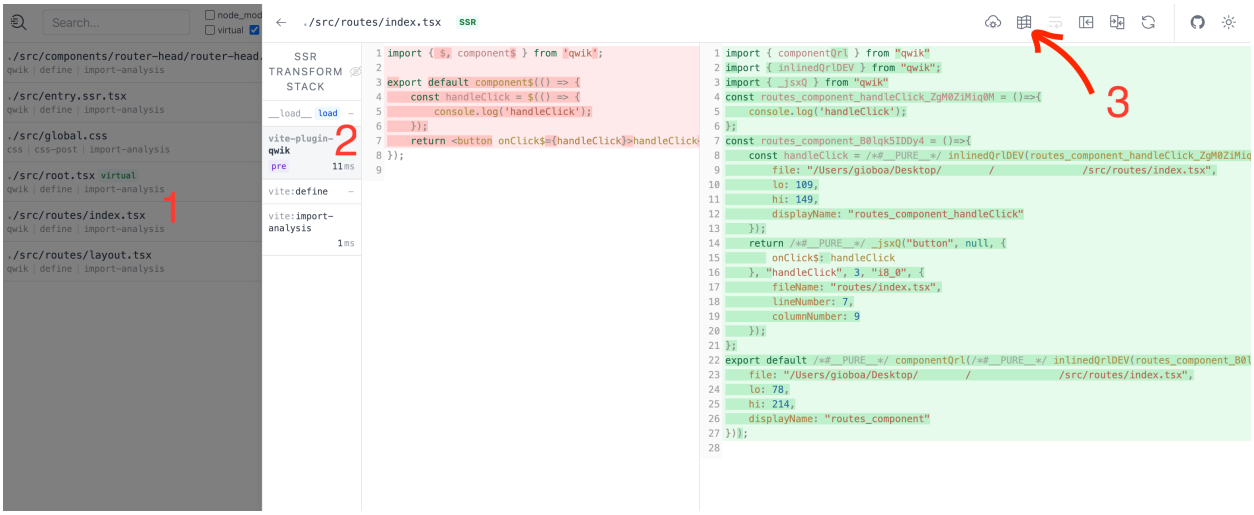
FILE: `./src/routes/index.tsx`

```
export default component$(() => {
  const handleClick = $(() => {
    console.log("handleClick");
  });
  return (
    <button onClick$={handleClick}>handleClick</button>
  );
});
```

Here are the screens of the build steps.



1_vite-plugin-inspect

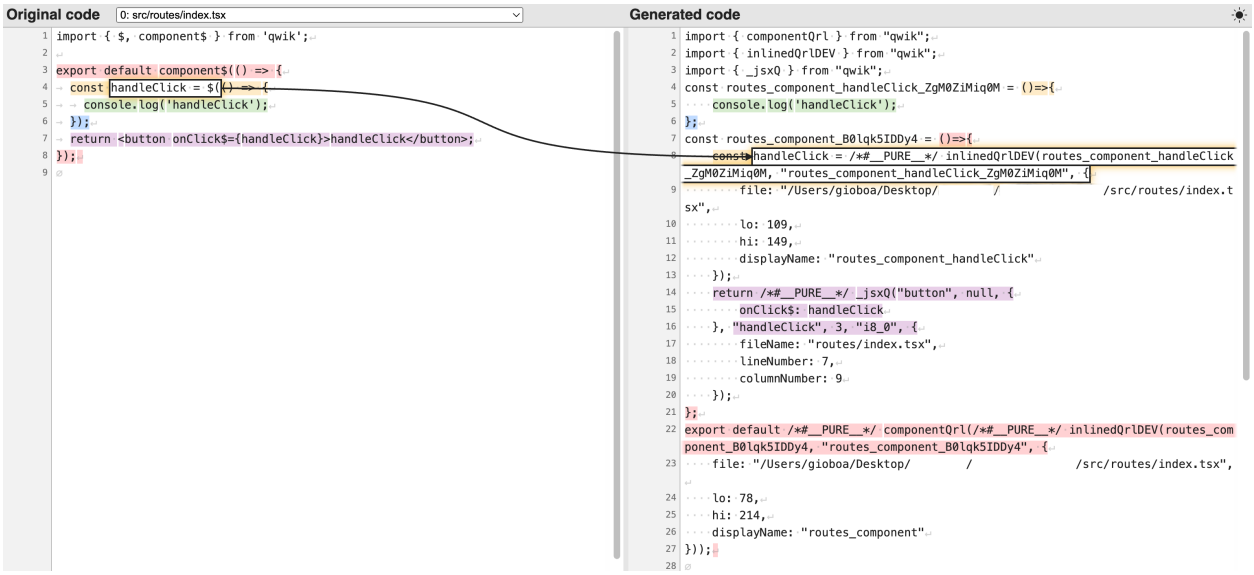


2_vite-plugin-inspect

Step 1: Clicking on the file `./src/routes/index.tsx` opens a panel displaying the transformation steps that have been performed.

Step 2: One of the steps is `vite-plugin-qwik`, which is responsible for the optimizations performed by the Qwik plugin, the `Optimizer`.

Step 3: Clicking on the icon provides a detailed analysis of how the file is optimized.



3_vite-plugin-inspect

Here, you can see that our `handleClick` function has been extracted on an isolated symbol.

> `Symbols` are single lazy-loadable pieces in Qwik; as you saw earlier, when `$` is encountered, the optimizer creates a symbol. `Bundles` are JavaScript packages that can contain one or more symbols. These packages are files loaded by the service worker under the hood and imported when needed.

This is a simple example, but let's complicate things to see how Qwik groups `Symbols` into the `Bundles`. This serves to optimize the entire Resumability process and obtain maximum performance.

Let's take this slightly more complex example.

FILE: `./src/routes/index.tsx`

```
export default component$(() => {
  const signalOne = useSignal(0);
  const signalTwo = useComputed$(() => {
    console.log("useComputed$");
    return signalOne.value * 2;
  });
  const handleClick = $(() => {
    console.log("handleClick");
    signalOne.value++;
  });
  return (
    <button onClick$={handleClick}>
      {signalOne.value} - {signalTwo.value}
    </button>
```

```
15    );
16  });
```

In this example, you can see `signalOne` and `signalOne`, which are two reactive variables. Let's go into more detail about the `useSignal` and `useComputed$` API, but for now, you just need to know that every time `signalOne` changes, `signalTwo` changes automatically too. Every time you click on the button, `signalOne` will be changed, and, as previously explained, `signalTwo` will also be changed. These transformations and optimizations only take place in the build phase for production, and the code that is produced is minified, which is why two console.log were added to better understand how the functions are divided in the screenshots. This way, you will be able to trace the complete tour without too much difficulty.

Let's start by analyzing the two symbols using the `vite-plugin-inspect` tool seen previously.

symbol_1

symbol_2

The build process creates two constants, which are the example functions. In a previous chapter, this behavior was discussed in detail. The functions contain `useLexicalScope`, which is a function that restores the necessary variables from the closure.

The build process generates a `q-manifest.json` file that provides a detailed graph of the bundles and symbols. This data is used by the service worker to cache network requests for known bundles.

The output of the build contains our functions.

dist > {..} q-manifest.json > { } symbols > { } s_EJVFv2QSBxo

```
214     "s_EJVFv2QSBxo": {
215       "origin": "routes/index.tsx",
216       "displayName": "routes_component_signalTwo_useComputed",
217       "canonicalFilename": "s_ejvfv2qsbxo",
218       "hash": "EJVFv2QSBxo",
219       "ctxKind": "function",
220       "ctxName": "useComputed$",
221       "captures": true,
222       "parent": "s_B0lqk5IDDy4",
223       "loc": [
224         177,
225         248
226       ]
227     },
```

symbol_1_manifest

dist > {..} q-manifest.json > { } symbols > { } s_ZgM0ZiMiq0M

```
228     "s_ZgM0ZiMiq0M": {
229       "origin": "routes/index.tsx",
230       "displayName": "routes_component_handleClick",
231       "canonicalFilename": "s_zgm0zimiq0m",
232       "hash": "ZgM0ZiMiq0M",
233       "ctxKind": "function",
234       "ctxName": "$",
235       "captures": true,
236       "parent": "s_B0lqk5IDDy4",
237       "loc": [
238         274,
239         335
240       ]
241     },
```

symbol_2_manifest

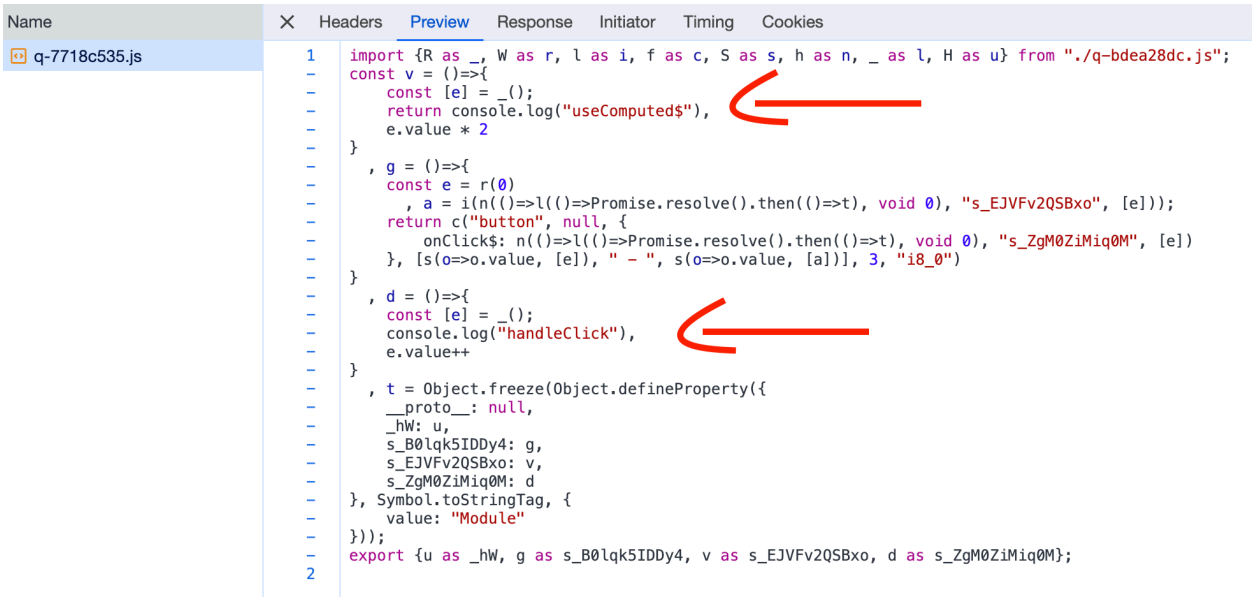
In fact, in this file, you will find `"displayName": "routes_component_signalTwo_useComputed"` and

`"canonicalFilename": "s_ejvfv2qsbxo"` and you will therefore find confirmation of the name of the symbol and the name of the API and the variable in the `displayName` property.

If you analyze the `q-manifest.json` file more deeply, you will see the description of the `q-7718c535.js` bundle, which contains previously seen symbols within the `symbols` property.

bundle

When launching the application and inspecting the network requests in the browser's DevTools, you can observe that the specific file containing the two `symbols` is requested. You can also see the two `console.log` statements.



bundle_network

Although the explanation involved several steps, let's hope this detailed exploration has clarified the process. Thanks to the `Qwik Optimizer` developed by the Qwik team, you can write modern and readable code while the framework handles the optimization. It's pretty cool!

Service Worker

The resumability technique relies on executing code related to user interactions. However, for the JavaScript code to be executed, it needs to be downloaded. This is where the Service Worker comes into play.

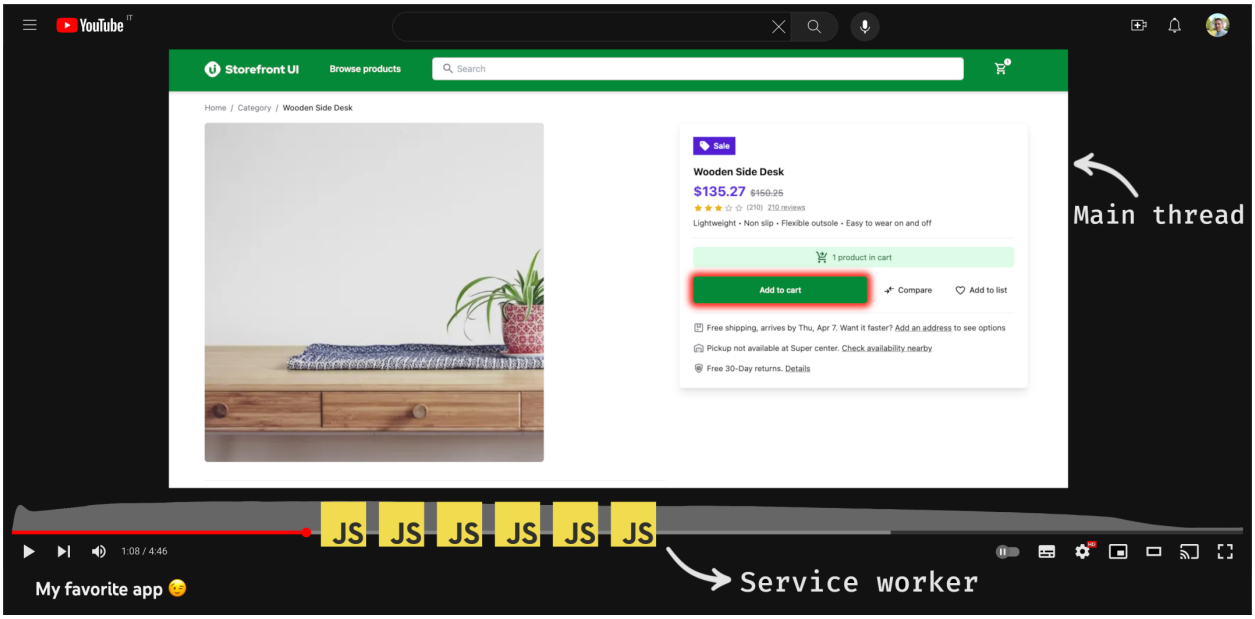
The Service Worker operates in a separate thread, allowing background actions to be performed without affecting the smoothness of the application. This is particularly useful because browsers are single-threaded.

In a Qwik app, the Service Worker's task is to preload all possible chunks that the user may request through their actions. It does this silently in the background. The code is already available in the browser's cache when the user interacts with the application. Instead of requesting it from a remote server, the code is ready to be executed.

For example, when the user clicks on a button for the first time, the request goes to the Service Worker, which returns the requested bundle (if the code is not in the cache, the request is sent to the network). The JavaScript bundle is then executed, and the application performs its logic.

Once a specific JavaScript related to an action has been executed, the code remains in memory and can be reused.

This process is similar to watching a YouTube video. When you watch a video, the rest of the video is downloaded in the background, ready to be played when needed. The Service Worker in the Resumability mental model performs a similar task. It prepares everything in the background to ensure optimal performance if further iterations are required.



ServiceWorker_YouTube

Another task of the Service Worker is to minimize the number of requests by preloading bundles. It uses the `q-manifest.json` file mentioned earlier. The `manifest` represents a complete graph of symbols and corresponding blocks, including the correlation between them. When a symbol is preloaded, the Service Worker also preloads all other related symbols.

Bundler configuration

One extremely powerful thing is to be able to change the way the framework builds our app via configuration. You usually have control over the application splitting because you use lazy loading, but this requires the developer to write dynamic imports and refactor the code. With Qwik, if you want, you have full control of which symbol goes into which chunk, and no other framework can offer this granular possibility. In fact, all $ are potential lazy load locations; this $ marker is used to inform the bundler on how to group the symbols. Thanks to the fact that the Qwik team also develops the bundling and optimization part, different strategies can be set up to load our bundles.

entryStrategy config

> Qwik's optimization and bundling strategy is already very good by default, but if you want full control over this process, you have the tools to do it.

In the Vite plugin `qwikVite()`, there is the `entryStrategy` parameter which can be changed if you wish to vary the optimization of the bundle. The first thing you can think of is to go and set it to create a single file. On the other hand, you have always been used to running our entire app at a single time, so in Qwik, it would be a single bundle with all the symbols. The problem with this approach is that you are loading more symbols than the client could request, you are effectively wasting bandwidth, and until the bundle has been loaded, you cannot execute any symbols.

> It should be noted that this approach might come in handy if you are in a no connectivity area because by loading the whole application, I'm able to use it, calls to external services permitting. Basically, I can create differently optimized applications just with a setting; for users with no connection, I will set the creation of a single file; for the others, instead, I will be able to benefit from the Resumability.

The opposite approach to the one just seen is to create a bundle for each symbol, which is the behavior that Qwik uses in development mode. With this high fragmentation, however, the client has to make many more requests to load all the bundles, and this often leads to undesirable waterfall request behavior.

By default for production, the `"entryStrategy"` parameter is set to `smart` which allows Qwik to make heuristics about how symbols should be loaded lazily, but you can override this logic by providing a `manual` configuration like this one. FILE: `vite.config.ts`

```
export default defineConfig(() => {
  return {
    plugins: [
      qwikVite({
        entryStrategy: {
          type: "smart",
          manual: {
            ...{
              i1Cv0pYJNR0: "bundleA",
              "0vphQYqOdZI": "bundleA",
            },
            ...{
              vXb90XKAnjE: "bundleB",
              hYpp40gCb60: "bundleB",
            },
          },
        },
      }),
      //...
    ],
    // ...
  };
});
```

In this example, the configuration specifies which symbols go into `bundleA` and `bundleB`. However, manually configuring the bundles can be a tedious task. The optimal chunks depend on user behavior, which can only be observed over time. Therefore, Qwik Insight was created to collect anonymous usage information and refine the bundling strategy based on real user data.

Qwik Insights

Qwik Insight is a feature that collects information about requested symbols, their priority, frequency, and associations. This data can be used to optimize the bundling strategy in subsequent builds. Qwik Insight can also track bundle priorities for each route, instructing the service worker on which files to download immediately.

The application needs to be wrapped with a special component that anonymously aggregates and collects the symbols that are used to activate Qwik Insights. The data is sent to a remote database for storage. In the build process, the Qwik Vite plugin can be configured to use the real user usage information.

FILE: `vite.config.ts`.

```
export default defineConfig(async () => {
  return {
    plugins: [
      qwikInsights({
        publicApiKey: loadEnv("", ".", "")
          .PUBLIC_QWIK_INSIGHTS_KEY,
      }),
      //...
    ],
    // ...
  };
});
```

Qwik Insights can track the bundles needed for each route, providing insights into unused parts of the application or the effectiveness of specific buttons as calls to action. It can also preload all the required bundles from a specific link, ensuring instant browsing without waiting for file downloads.

Technical Requirements to Start a New Qwik Application

Now that the reasons for Qwik's uniqueness and the benefits it offers have been explained let's delve into the technical requirements for setting up a Qwik application on a workstation.

Getting started with Qwik is a straightforward process. The following tools are necessary:

- Node.js v20.12.2 or higher
- A preferred IDE (VSCode is recommended)

Since Qwik is built on TypeScript, familiarity with this language is essential. TypeScript enhances JavaScript with types and various other features. To gain a deeper understanding of TypeScript, reading "Beginners Guide to TypeScript"[21] by Christian Santos is recommended.

> You can use JavaScript for your Qwik application, but TypeScript is the preferred way.

[21] https://www.newline.co/beginners-guide-to-typescript

Creating a New Project with Qwik CLI

In the past, structuring a web application involved writing everything from scratch and struggling to find the correct configuration for different libraries. Fortunately, this has changed, and it is now expected for every tool to have a command line interface (CLI) feature.

Creating a Qwik application is a simple process. The Qwik CLI needs to be executed in the shell. Qwik supports various package managers, allowing you to choose your preferred one and run one of the following commands:

```
npm create qwik@latest
## or
pnpm create qwik@latest
## or
yarn create qwik
## or
bun create qwik@latest
```

The Qwik CLI will provide guidance through an interactive menu. It will prompt you to specify the folder where you want to create the application, select one of the available starters (Empty App, Basic App, etc.), and decide whether to install the dependencies and initialize a git repository. Once these steps are completed, you will be ready to proceed.

The installed packages will include `Qwik`, the framework with its primitives and logic, and `Qwik City`, the library responsible for managing routing, middleware, and more. Both packages are maintained and managed by the Qwik team.

While it is possible to use Qwik without Qwik City and manually handle routing, it is not the most efficient approach. It is recommended to follow the standard approach and utilize Qwik City.

By default, the project is configured with Prettier and some `npm scripts` for code formatting using the default settings. The same applies to ESLint, which is configured with recommended settings for TypeScript and a special plugin called `eslint-plugin-qwik`, developed and maintained by the Qwik team. This plugin enables you to standardize the syntax of your application according to official rules, ensuring that you can utilize Qwik to its full potential. It provides warnings to assist you in resolving any issues and standardizing your code in the most effective manner.

npm vs pnpm

When dependencies are installed in our projects using `npm`, a `node_modules` folder is created for each project. These folders can occupy a significant amount of disk space, potentially reducing the available free space. There are npm packages available, such as npkill[22], that allow you to remove these folders from various projects via the command line, freeing up space.

[22] https://www.npmjs.com/package/npkill

However, with the introduction of `pnpm`, this issue can be permanently resolved. pnpm takes a different approach by creating the `node_modules` folder during dependency installation. Instead of downloading the necessary files within the project, it downloads the dependencies into a global cache. Within the project, it establishes symbolic links between folders to reference the dependencies. If a dependency already exists in the local directory, it will not be downloaded again.

For instance, if we download the latest version of TypeScript, it will be downloaded into the global cache. If another project requires the same version of TypeScript, it will reuse the package from the cache. This simple management technique saves disk space and download time, as we often work on similar projects with the same technologies, resulting in dependencies that are already present in our system. Even if we delete the `node_modules` folder and reinstall everything, no downloads will be performed because the dependencies are already available in the global cache.

Understanding the Directory Structure

Qwik supports files that return JSX (such as .tsx) as well as files that allow you to write Markdown (such as .md and .mdx). Markdown is a markup language with plain text syntax that can be converted to HTML and other formats. It is commonly used for formatting README files, writing forum messages, and creating formatted text using a simple text editor. It is also convenient for creating documentation pages due to its simplicity.

A typical Qwik project created using the CLI has the following structure:

```
qwik-app
├── README.md
├── package.json
├── public
│   └── favicon.svg
├── src
│   ├── components
│   │   └── router-head
│   │       └── router-head.tsx
│   ├── entry.dev.tsx
│   ├── entry.preview.tsx
│   ├── entry.ssr.tsx
│   ├── global.css
│   ├── root.tsx
│   └── routes
│       ├── users
│       │   ├── users.css
│       │   └── index.tsx
│       ├── index.tsx
│       ├── layout.tsx
│       └── service-worker.ts
```

```
22 ├── tsconfig.json
23 └── vite.config.ts
```

public/

This folder contains the static files of our application, such as images, fonts, icons, etc. During the build process, these files are copied to the `/dist` directory and served at the root URL path. For example, the file `public/favicon.svg` will be served at the address `https://my-website.com/favicon.svg`.

src/routes/

This is a special folder that Qwik City uses to create routing paths. Folders and files within this directory have special meanings based on naming conventions.

- **routing paths based logic**: the logic that is considered to base the routing of our application is quite linear without fortunately too many exceptions. Each nested folder generates a new application route, so let's imagine that we have an `orders` folder inside the `src/routes/` folder, this folder will be considered as a new URL path available in our application. If we create an `index.tsx` file with a Qwik component inside the folder, this will be rendered to us when we query the URL `https://my-website.com/orders/`. Our homepage is a special case because it does not contain paths, in fact, the index.tsx present inside the `src/routes` folder is our homepage. Here are some examples to give you an idea:
- https://my-website.com/ -> src/routes/index.tsx
- https://my-website.com/orders -> src/routes/orders/index.tsx
- https://my-website.com/orders/detail/ -> src/routes/orders/detail/index.tsx

  Managing parameters within the URL is also possible. For example, to view a specific order, the URL would typically be something like `https://my-website.com/orders/123/`, where `123` is the identifier of the order. To handle this, a folder `src/routes/orders/[id]/` can be declared with an `index.tsx` file inside. In this file, the parameters from the URL can be accessed using the `useLocation().params.id` API.

  Additionally, URLs like `https://my-website.com/shop/clothes/t-shirts/` can be achieved. To accomplish this, a folder `src/routes/[...shop]` can be defined using naming conventions. In this example, `shop` is the string parameter accessible as `useLocation().params.shop` in our component, and it captures all the path after `/shop`.
- **layout.tsx**: Layout is an important concept in Qwik City as it relates to routing. In most applications, a fixed structure is required on all pages, such as a header and a footer. This can be achieved by declaring a `layout.tsx` file inside the `src/routes` folder. This file contains a special syntax that will now be analyzed.

  FILE: `src/routes/layout.tsx`

```
export default component$(() => {
  return (
    <div>
      <Header />
      <Slot />
      <Footer />
    </div>
  );
});
```

In addition to the `Header` and `Footer` components, there is a special component called `<Slot />`. This component instructs the framework to replace `<Slot />` with the content of our page. For example, if we are on the homepage, it will replace `<Slot />` with the component present in the `src/routes/index.tsx` file. Based on the requested URL, it will follow the logic described earlier.

Additional layouts within the header and footer are also possible. For example, a side menu can be included only in the admin section. A `layout.tsx` file can be created inside nested subfolders, such as `/src/routes/admin/layout.tsx` to achieve this. This additional layout will be included within the layout that contains the header and footer. Finally, the page (the `index.tsx` file) will be rendered inside the layout present in the admin folder.

If a specific layout is not desired for a particular page, named layouts can be used. By giving a layout a name, such as `layout-special.tsx`, a special layout can be declared that can be used by pages instead of inheriting the default layout. To activate the special layout, a file of this type can be defined: `src/routes/contact/index@special.tsx`. By using `@<name>` in the file name, the framework is instructed to use the layout with the same name. This provides the flexibility to manage different cases as needed.

The `src/routes/404.tsx` file is used to handle undefined routes in our application, such as `https://my-website.com/not-existing-route/`. When a route is not defined, the 404 page is used to show the user a service page. 404 pages can also be nested like layouts, and the more specific 404 page is used when an undefined route is encountered. For example, if a 404.tsx file is inserted into the `src/routes/user` folder, it will be used for the URL `https://my-website.com/user/not-existing-route`, instead of the generic 404 page.

It is also possible to declare a folder and bypass the routing logic using the `(myFolder)` annotation with round brackets. This allows the creation of a folder where files can be inserted without being subject to the routing logic. This folder is ignored by Qwik City and serves as a logical grouping of project files. For example, if the structure is `src/routes/orders/(myFolder)/[id]/detail/index.tsx`, the URL to view the page will be `https://my-website.com/orders/123/detail`. The `(myFolder)` folder is ignored and only serves developers to logically group project files.

src/components/

This directory conventionally hosts the components of our web application. However, a different folder can be chosen if desired.

src/entry.*.tsx

- `entry.dev.tsx` This file is the development entry point using only client-side modules, without SSR.
- `entry.preview.tsx` This file is the bundle entry point for `preview` and it serves your app's built-in production mode.
- `entry.ssr.tsx` The application is rendered server side and this file is the entry point to start the Server Side Rendering (SSR) process.

src/global.css

This file is used for global CSS styles. Global styles can be defined in this file.

tsconfig.json

This file contains the TypeScript compiler configuration. It is present because the project is written in TypeScript and contains the necessary compiler options.

vite.config.ts

Qwik uses Vite to build the project, and this file contains the Vite configuration. The default Qwik project includes some Vite plugins used during the compilation and build phases:

- `tsconfigPaths()`: This plugin is used to load modules whose location is specified in the paths section of tsconfig.json.
- `qwikCity()` and `qwikVite()`: These plugins, developed by the Qwik team, are important for using the Qwik Resumability approach.

  As developers we write the code with the best possible Developer Experience (DX) and the Vite plugins created by the Qwik team can optimize the code for us, obviously without changing the behavior, but only using the new Qwik mental model.

Middleware

For instance, to expose RESTful APIs or GraphQL APIs, the middleware server that Qwik City provides can be utilized. This feature proves beneficial for centralizing authentication, security, logging, and many other logics. Middleware comprises a set of functions that are called in a specific order. If the `index.tsx` has a default export component, then the HTTP request will return the component. When visiting a specific route of the application, since the default component is exported, this is returned and displays the requested page as if it were the last step of the chain.

This example shows a simple `onRequest` middleware function that logs all requests because `onRequest` intercepts all HTTP methods.

File: `src/routes/layout.tsx`

```
export const onRequest: RequestHandler = async ({
  next,
  url,
}) => {
  console.log("Before request", url);
  await next();
  console.log("After request", url);
};
```

To intercept a specific HTTP method, one of these variations can be used. If both `onRequest` and `onGet` are used, for example, both will execute, but `onRequest` will execute before `onGet` in the chain.

```
// Called only with a specific HTTP method
export const onGet: RequestHandler = async (requestEvent) => { ... }
export const onPost: RequestHandler = async (requestEvent) => { ... }
export const onPut: RequestHandler = async (requestEvent) => { ... }
export const onPatch: RequestHandler = async (requestEvent) => { ... }
export const onDelete: RequestHandler = async (requestEvent) => { ... }
```

Creating a rest API that returns a JSON based on some backend logic is straightforward. The `json(status, object)` method can be called. The json() method will automatically set the Content-Type header to `application/json; charset=utf-8` and JSON stringify the data.

```
export const onGet: RequestHandler = async ({ json }) => {
    /// your backend logic (e.g., read from the Database)
  json(200, <your_response_object_here>);
};
```

Here, the `json` method is taken from the parameters, but there are many other parameters that can be used. The most useful and interesting ones that will also be used in the subsequent chapters are listed below.

- **sharedMap**: Use sharedMap as a means to share data between middleware functions. The sharedMap is scoped to the HTTP request. A common use case is to use sharedMap to store user details so that other middleware functions can use it.

```
export const onGet: RequestHandler = async ({ headers, json }) => {
    // read
    sharedMap.get('user') as User;
    // write
    sharedMap.set('user', user);
    [...]
};
```

- **headers**: Use headers to set response headers associated with the current request. (For reading request headers, see request.headers.) Middleware can manually add response headers to the response using the headers property.

```
export const onGet: RequestHandler = async ({ headers, json }) => {
    headers.set('X-SRF-TOKEN', Math.random().toString(36).replace('0.', ''));
    [...]
};
```

- **cookie**: Use cookie to set and retrieve cookie information for a request. Middleware can manually read and set cookies using the cookie function. This might be useful for setting a session cookie, such as a JWT token, or a cookie to track a user.

```
export const onGet: RequestHandler = async ({ cookie, json }) => {
    // read
    cookie.get('COOKIE_KEY')?.number() || 0;
    // write
    cookie.set('COOKIE_KEY', count);
    [...]
};
```

- **env**: Retrieve environmental property in a platform-independent way.

```
export const onGet: RequestHandler = async ({ env, json }) => {
    env.get('ENV_VARIABLE_KEY_HERE'),
    [...]
};
```

- **redirect()**: Redirect to a new URL. Note the importance of throwing to prevent other middleware functions from running. The redirect() method will automatically set the Location header to the redirect URL.

```
export const onGet: RequestHandler = async ({
  redirect,
  url,
}) => {
  throw redirect(
    308,
    new URL("redirect_url", url).toString(),
  );
};
```

Other parameters can be exploited, such as `URL`, `basePathname`, `params`, or `query`, which are self-explanatory, plus others that will not be used during these chapters but are available in the official documentation.

Plugins

Special files in the `src/routes` directory named `plugin.ts` or `plugin@<name>.ts` can be created to manage incoming requests. These files have the peculiarity that they can manage requests with maximum precedence, even before layouts. Multiple plugin.ts files, each with a different name `@<name>`, can be had, and the execution order of the various files is based on the alphabetical order of the files (`plugin.ts` will always be first). However, naming the various files is very useful for quick identification.

Summary

This chapter is full of advanced concepts. It begins by explaining what bundlers are. The transition to Vite is discussed, explaining why it performs better and why it has become the de facto standard for modern front-end development. We start with Webpack and its limitations. Accompanied by two code snippets, the way Vite manages dynamic imports is analyzed. Vite provides the opportunity to create plugins, and a simple example is created that paves the way to see what the Qwik team created instead.

First, the Qwik compilation process is analyzed, and thanks to the `vite-plugin-inspect` plugin, a deep dive is taken into the details of what happens in the build process. The grouping of files and how the service worker can understand which symbols are inside a given bundle are explored in the division between' symbols' and' bundles'. This entire process is possible because the `Qwik Optimizer` can do an amazing job. Here, the use of plugins has been pushed to the extreme. Normally, these plugins are used to perform a one-to-one transformation of the files; for example, a TypeScript file input produces an output JavaScript file. However, to break it down into symbols and make the mental model of Resumability possible, more has been done. A file is entered, and multiple JavaScript files are produced based on the `$` marker. A series of logic is employed that makes this multi-transformation possible. Producing multiple files from a single input is a feature not provided by Vite, but it has been implemented in the `vite-plugin-qwik` plugin to achieve the result. The analysis then proceeds to how to create a new Qwik application and the technical requirements to start developing.

The scaffolding of a new Qwik application is examined, the configurations are analyzed, and a deep dive is taken into the logic of the basic routing file. The handling of layouts, named layouts, and the management of

the 404 page are discussed with all its related facets. Finally, a deep dive is taken into the details of middleware, understanding their purpose and the order in which they are executed. With some examples, everything is clarified, and then a deep dive is taken into the details of the plugin files, which are very useful for executing logic even before processing the layout files. In the next chapter, the Core Web Vitals will be analyzed, and it will be seen how Qwik allows for achieving sensational results. It will also be understood how to deploy an application to make it accessible via a public web address.

SEO and Core Web Vitals

SEO

SEO stands for Search Engine Optimization and is a set of best practices to ensure that a site is well-indexed by search engines. This optimization will bring advantages in terms of traffic, and therefore, the business will also benefit significantly. This chapter will explore some best practices to ensure the application is perfectly configured for excellent SEO.

It's important to understand that search engines scan web pages to index websites and ensure that the result of a search is as accurate as possible. This scan has a limited time for each site, and therefore, the more useful information that can be provided in this short time, the better it is for the site/application. The architectures seen in the previous chapters, Server Side Generation and Server Side Rendering, can be used to provide a quality result in the shortest possible time. These two approaches guarantee the return of an HTML page containing crawlable content to the search engine crawler. So, if these technologies are used, a good start is ensured.

The fundamental aspect of obtaining a good ranking is to have interesting content or provide a useful and frequently used service. The more visitors a site receives, the more the search engine will consider it an interesting result to show. It is not only necessary that the content attracts users, but visitors must also spend time within the application.

Bounce rate

A first metric to consider and constantly monitor is the `bounce rate` because it indicates how much people appreciate or, more importantly, don't appreciate the application or the user experience. This metric measures how many people visit a single page on the application or site and don't interact with the page before leaving. According to Google, GA4[23]'s engaged sessions meet at least one of the following conditions:

- Lasts longer than 10 seconds
- Has a conversion event
- Has 2+ page views

So, for example, it measures users who leave an e-commerce site without purchasing something or interacting (e.g., clicking a link). The contents can certainly influence this metric, but it could also depend on the application's loading time, which is too slow, and therefore, the user prefers to choose one of the competitors. Fortunately, with the use of Qwik, any slowness problems can be avoided.

A high `bounce rate` value means that the user's overall session duration is short. On the other hand, a low value means that they spend time on the page and interact with it. It should be noted that a high value

[23] https://support.google.com/analytics/answer/10089681

cannot always be considered negative; it all depends on what type of application is in use. For example, if the application provides news or other content that involves sessions from a single page, it is perfectly normal to have a high bounce rate value. If this is not the case, then this high value is a bad thing. A bounce rate of 35% or less is good, while a bounce rate of 60% or more is high, and therefore, there is room for improvement and encouragement for users to stay more in the application.

To calculate this value, Google Analytics[24] can be used.

Dwell time

By measuring the time that users spend on a page before returning to the Search Engine Results Pages (SERP) we can obtain what is defined as "Dwell time". How long does the user stay on a page before moving away? This metric answers exactly this question. Let's say you are looking for how to integrate Google Maps with Qwik, and from the search results, you click on one of the first results. It's a page full of advertisements, so you go back to the SERP after only 5 seconds; the Dwell time is, therefore, 5 seconds. You click on another article, and it is well-written, pleasant to read, and very useful. You spend several minutes reading the detailed description; in this case, the Dwell time is much higher.

On the other hand, users may find the information they need at the top of the page and quickly return to the SERP, so a shorter dwell time does not necessarily indicate lower content quality or user dissatisfaction. The algorithms that calculate the ranking are not public; therefore, the best practices are the result of the experience that various experts have shared. Given the statement made earlier, Dwell time is probably not a ranking factor, at least not a direct one. However, dwell time can still provide insights into user behavior. If users tend to stay on the pages longer, it could mean they find the content engaging, and shorter stays could mean there are underlying issues causing users to bounce off the pages.

Semantic HTML

It is known for sure that if websites are difficult to read, they are ignored by search engines because they prefer those that have been updated with modern HTML features. Therefore, sites built a long time ago are disadvantaged. One of the characteristics of modern websites is the use of HTML5, which has very specific semantics that search engine crawlers prefer because it is more understandable by better defining the different sections and layout of the web pages.

Let's take this example of a non-semantic HTML <body> tag.

```
<body>
  <div>Header</div>
  <div>Main content</div>
  <div>Footer</div>
</body>
```

Instead of using for the various sections, it is more appropriate to use what HTML5 makes available, e.g., the related <header>, <footer>, and <main> tags.

[24] https://support.google.com/analytics/answer/10089681

```
<body>
  <header>header</header>
  <main>main content</main>
  <footer>footer</footer>
</body>
```

By doing this, the different sections in the HTML page have been correctly represented. They are more descriptive tags than tags, which make them difficult to read by search engines but also by screen readers. In fact, by using the correct semantics, the site also becomes more accessible, and this also helps the application to be better.

For navigation, however, the <nav> tag can be used, which contains the links that create the menu. Let's see an example:

```
<header>
  <img src="logo.png" alt="logo" />
  <nav>
    <a href="one.html">Page One</a>
    <a href="two.html">Page Two</a>
  </nav>
</header>
```

Here, it can be seen that the <nav> tag has been used for the menu, and this is correct. The `alt` attribute has also been used for the image, which improves the accessibility of the page because screen readers can recognize the attribute and help the user.

The main content should be inserted inside the <main> tag, and inside it, <article> and <section> can be used to respect the semantics.

Here is an example of correct semantics:

```
<main>
    <article>
        <h1>[...]</h1>
        <section>
            <h2>[...]</h1>
            <p>[...]</p>
        </section>
        <section>
            <h2>[...]</h1>
            <p>[...]</p>
        </section>
    </article>
</main>
```

The <article> tag is a standalone section on the page and can contain several <section> tags within it.

<section> although similar to can enclose logical groups of related content.

There are many other semantic tags in HTML5. The list shown here is intended to be indicative and certainly does not cover all possible tags. For that, the official HTML documentation can be referred to. The important thing to understand is that if the correct HTML tags are used, the application can be analyzed better by search engines and will, therefore, be indexed better. Furthermore, thanks to the use of the correct syntax, inclusive and accessible applications can be created to reach all users without distinction.

Link dofollow vs nofollow

Let's start by saying that this attribute is invisible to the end user; nothing changes in visual and behavioral terms. However, the situation changes for search engine bots; by default, a link has the implicit `dofollow` attribute. It has been seen that the search engine crawler is in a hurry, and therefore, the aim is to make it crawl the application in the shortest possible time and in the best possible way. Here, the links on the pages have a further configuration to help this scan. The normal behavior of the bot that crawls the pages would be to follow all the links on the page and take the information obtained into consideration for the ranking factor.

Let's imagine there is a recipe blog, and on one page, a specific recipe is being explained. A certain step of the recipe is very complex, and it has been explained several times in the blog, so instead of explaining it again each time, a specific post for this step has been created, and in each recipe, the link to it is put. This is useful for the end user; if they already know the step because they are an expert, they can skip the description of the procedure. If they feel unsure, then they can go and see how this particular procedure is performed. This is a link for the search engine crawler, and as default behavior, it follows it. If there are several pages linking to this particular passage, that page will gain scores in terms of credibility and indexing. It then happens that since the description is particularly valid and accurate, another very authoritative site inserts a link to the page in one of its articles, so a backlink has been received, and this gives credibility to the search engine logic. The more famous the site that mentions it, the more consideration will be acquired. By observing how many incoming links a page has and from which sites, the search engine will make a reason, such as if many people and sites link to that particular page, it must necessarily be a good page!

In the past, the logic was much more basic; it was enough to receive backlinks, and the more that was had, the more popular the page was. Soon, to hack the system, fake sites were built just to create backlinks to their main sites and to distort indexing and position themselves in a better position in search results. Today, this is no longer the case; this practice has been blocked. It is precisely to counter this practice, which involved sending spam with links to blogs and external sites, that the `nofollow` attribute was introduced to stop these bad practices in their tracks. In practice, with this attribute, the crawler is being told not to give weight to the link, not to follow it, and therefore, in short, to ignore it.

Imagine that comments can be inserted in the recipe blog. To increase user engagement, the risk is taken that someone inserts a link to the site to increase its visibility. So, by automatically adding this `nofollow` attribute to all the links in the comments, wrong information can be avoided being given to the search engine. As managers of the blog, it will be their responsibility to check the various comments, but in the meantime, a big problem has been blocked. What is being inserted with `nofollow` is, however, a suggestion that is being given to the search engine, and it is not known for sure whether it will then follow the link and consider it. This is because the operating algorithms are secret. Google added two new attributes in 2019, `rel="ugc"` and

`rel="sponsored"`, which complement the `nofollow` attribute. With `rel="ugc"`, it is being said that they are links created by users over which there is no control, as in the example of the comments in the recipe blog. With `rel="sponsored"`, it is explicitly being said that the link is sponsored or paid for.

In general, if there is no desire to be associated with the link on the page, the `nofollow` attribute must be used, and the two new attributes just seen can also be specified. For all other cases, there is `dofollow`, which is not an existing value for the `rel` attribute, but it is simply the default value for links. In practice, by not specifying anything for the links, it is being said that the link is considered useful for the user, but also for the search engine to strengthen the positioning of the linked content because quality feedback is transferred to the linked site. One might think of inserting `nofollow` everywhere, but this would go against the principles of the web, where links are part of the ecosystem itself. So, this attribute must be carefully measured to avoid being penalized and only used when the source is not certain, or there is no control over the users who post the content.

Sitemap and robots files

This file is a special document that assists search engines in gaining an overview of all the pages available on a given site, along with its general structure. If an application has many pages (over 200), it is advisable to have this specific file. In this file, it's only necessary to list the indexable pages, and additional information, such as the date of the last update of a page, can also be inserted. This file essentially says: "Hello search engine, instead of investing precious time manually scanning the various links and finding out which ones have changed since the last encounter, here is a list of them, so less work is required, and the process can go faster."

The Sitemap is designed for search engines and uses the XML format. Here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://www.my-app.com/</loc>
    <lastmod>YYYY-MM-DD</lastmod>
  </url>
  <url>
    <loc>https://www.my-app.com/contacts.html</loc>
    <lastmod>YYYY-MM-DD</lastmod>
  </url>
  <url>
    <loc>https://www.my-app.com/about.html</loc>
    <lastmod>YYYY-MM-DD</lastmod>
  </url>
</urlset>
```

The first part defines the v.1.0 standard and character encoding, while `urlset` contains all the URLs included in the sitemap and describes which version of the XML Sitemap is being used. Each URL defines the location and the date of the last modification; the latter can have different formats:

- **Complete dates**: YYYY-MM-DD
- **Complete date plus hours and minutes**: YYYY-MM-DDThh:mmTZD
- **Complete date plus hours, minutes, and seconds**: YYYY-MM-DDThh:mm:ssTZD
- **Complete date plus hours, minutes, seconds, and a decimal fraction of a second**: YYYY-MM-DDThh:mm:ss.sTZD

In the past, it was possible to define the priority and frequency with which the change was expected for that specific URL, but as of today, support for these two extra properties has been removed.

The Sitemap must be served on the site in the main directory `https://www.my-app.com/sitemap.xml` precisely by convention and to facilitate identification by search engines. Keeping the Sitemap updated ensures that an accurate picture of the site is always available, so it should be generated whenever something changes in the structure, for example, the removal of a page. If the `lastmod` attribute is used, then it's important to update it correctly so that the modified contents are highlighted. Another aspect not to be overlooked is to refer to the XML Sitemap in the `robots.txt` file.

robots.txt

The `robots.txt` file is a text file that specifies which contents can be scanned and which cannot. This file may seem very similar to the previous one, but there is a significant difference. The `sitemap.xml` shows search engines the general structure of the application, while `robots.txt` declares what should be excluded from the indexing. This file saves the search engine time and prevents the site from being overloaded with requests. It is composed of one or more rules that grant or limit access to specific content.

However, this does not guarantee blocking the indexing of the page itself; it is merely a suggestion. The `noindex` attribute within the page's metadata must be used to block indexing by search engines.

Let's see an example:

```
[...]
<head>
  [...]
  <meta name="robots" content="noindex" />
  [...]
</head>
[...]
```

By doing this, the page can be excluded from the Search Engine Results Pages (SERP). This `robots.txt` must be served on the site in the main directory `https://www.my-app.com/robots.txt` precisely by convention and to facilitate identification by search engines.

Let's see an example file:

```
User-agent: *
## Directories
Disallow: */profiles/
Disallow: */scripts/
Disallow: */includes/
## Disallow users paths
Disallow: /users*

## Sitemap details
Sitemap: https://www.my-app.com/sitemap.xml
```

The available fields are:

- **User-Agent**: this field is used to specify the name of the crawler that must comply with the restrictions, with * the rules apply to all search engines.
- **Disallow**: with this term, the pages of the site that must not be scanned during indexing can be indicated. As in the example, a specific path or URL can be defined. Furthermore, for each User-Agent, one or more entries can be defined. As mentioned in the previous paragraph, it is necessary to indicate the path that leads to the sitemap.xml file, which, by standard, is in the main directory.

Partytown

To analyze metrics related to SEO, external services loaded on the page are typically utilized, the most notable of which is Google Analytics[25]. Besides these scripts for analyzing SEO-related metrics, other scripts can be inserted to help track events on the page. For instance, consider a scenario where there's a need to monitor how many users interact with a specific banner or button. With these third-party scripts, the necessary code can be inserted into the application, collect the events, and then analyze the collected data to make considerations and adjustments.

These are just examples of third-party scripts that can be inserted into the page. Everything seems perfect and simple to implement. However, these scripts, which are loaded when the application starts, slow down the rendering of the page and negatively impact its performance. As discussed in depth in previous chapters, downloading and executing JavaScript code upfront inevitably slows down the application.

Third-party scripts have many downsides:

- They send too many network requests. When rendering a page it is very important to limit the number of network calls because the more we make, the more we slow down the loading process.
- They send too much JavaScript. This can block the DOM construction because the main thread is busy. The result is a delay in the speed at which pages can be rendered.
- Parsing and running CPU-intensive scripts can delay user interaction and drain battery life.
- They often have insufficient HTTP caching, which forces resources to be frequently retrieved from the network.

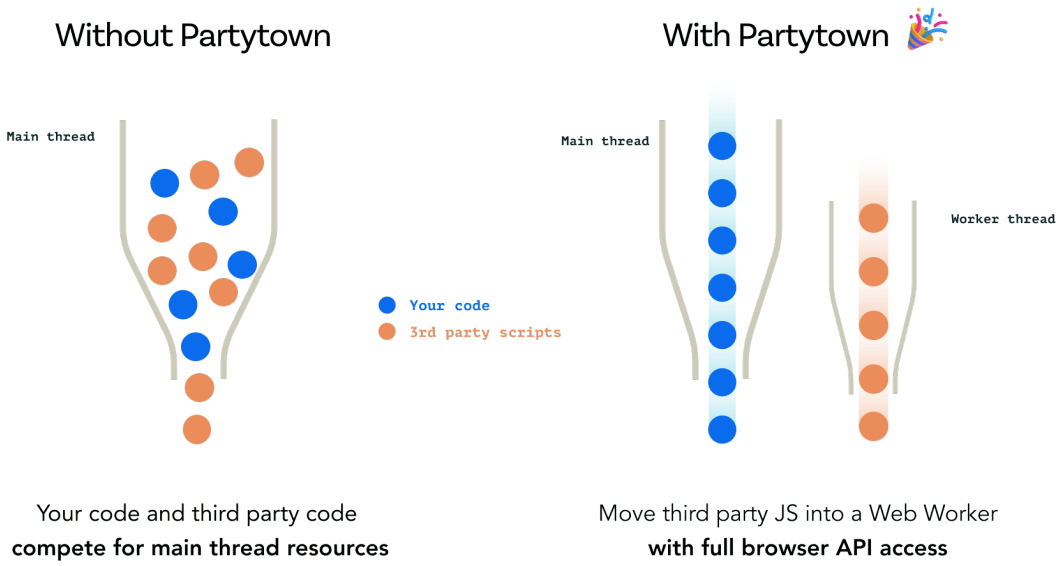
[25] https://support.google.com/analytics/answer/10089681

- They use legacy APIs (e.g., document.write()), which are known to be detrimental to user experience.
- Third-party scripts also often use embedding techniques that can block window.onload, even if the embedding uses asynchronous or deferred.

Almost all public sites nowadays run third-party scripts, but how can this problem be solved? Partytown[26] is a library that aims to definitively resolve the overhead caused by the download and execution of these scripts by the browser.

What Partytown does is move the loading of third-party scripts to a separate service worker to help speed up sites by dedicating the main thread to code and offloading third-party scripts to a web worker. Additionally, Partytown can create a third-party script sandbox that limits third-party script access to the main thread.

Without Partytown, code and third-party code compete for main thread resources, but with Partytown, performance is significantly better.



Partytown

Image from https://partytown.builder.io/

Let's see how a third-party script can be incorporated to utilize Partytown. To use Partytown, the developer must decide which scripts they want to move to the web worker using an opt-in approach. This means that other scripts would remain unchanged. To activate Partytown, it is necessary to insert the `type="text/partytown"` attribute into the scripts that the library should manage. This `type="text/partytown"` attribute does two things:

- It prevents the main thread from executing the script because it is not a known type and is therefore ignored by the browser.

[26] https://partytown.builder.io/

- It provides a selector for Partytown to query, for example, document.querySelectorAll(‘script[type=”text/partytown”]’)

```
- <script>...</script>
+ <script type="text/partytown">...</script>
```

The web worker is provided with scripts to execute within the worker thread. The web worker creates JavaScript proxies to replicate and forward calls to main thread APIs (such as DOM operations). The service operator intercepts requests and can then communicate asynchronously with the main thread. This mechanism allows the application to be relieved from managing third-party scripts and enables it to have amazing performance. One nice thing is that it works with practically all modern frameworks because it is just using JavaScript code and nothing else.

Core Web Vitals

In 2020, Google announced the introduction of a series of metrics that evaluate the user experience of a website and are considered essential for a good user experience and a more pleasant web, the Web Vitals. Google is actively committed to making the usability of websites simpler and more effective with the aim of facilitating users’ browsing experience and providing answers quickly and clearly. The Web Vitals are standard metrics for measuring these aspects. Among these are three considered most important, which together form the `Core Web Vitals (CWV)`, which measure the loading speed, the page’s response times, and the layout’s stability. These metrics are important because they are integrated with SEO through a set of specific UX indicators that actively contribute to the ranking of the web application. Here are the metrics in question:

- Largest Contentful Paint (LCP)
- Cumulative Layout Shift (CLS)
- Interaction to Next Paint (INP)*

*Interaction to Next Paint replaced First Input Delay (FID) in March 2024

Let’s discover these metrics in detail and explore some tricks for optimizing them.

Largest Contentful Paint (LCP)

The LCP metric indicates the rendering time of the largest element visible within the screen when loading the site and measures the performance of the loading. What matters is what appears in the opening viewport of the site, while everything “below the fold” is not considered. This metric assumes that the largest visible object in the window is the main content and penalizes invasive content such as very large headlines, huge opening images, videos, banner ads, and pop-ups. Note that the download time is not considered but the amount of space that the object uses in the user’s window. So, all elements of type `<img>`, `<svg>`, `<video>`, or blocks that contain informative texts for the page are important. Therefore, as a general rule, it is important not to place videos or images “above the fold” that are too large to avoid being penalized. This metric is expressed in seconds and should be considered in this way:

- < 2.5 sec. -> Good
- < 4.0 sec. -> Needs Improvement
- > 4.0 sec. -> Poor

If images are intended to be shown in the viewport of the page, it is important to use the right formats to load the images. Using WebP, which is a modern format, provides superior image compression, allowing for the reduction of image size (e.g., SVGs, PNGs) to make loading faster. Furthermore, CDNs can be used to host images, improving response times and this metric. Additionally, external image resizing services allow for on-the-fly formatting and requesting the most suitable size for the user's resolution.

Here is an example:

https://‘your-image-resizing-provider/your-image.ext‘?w=1000&h=600&format=webp.

By doing so, the image can be obtained from the cache and resized and formatted if necessary. If the format and size already exist in the provider's cache, the response will be immediately returned without any computation.

Cumulative Layout Shift (CLS)

Cumulative Layout Shift measures the displacement of visible elements and layout changes that occur suddenly during the entire duration of the page visit, causing a shift after loading the site area due to scaling, with a disturbing effect. So, imagine having an advertising banner that appears on the homepage after a few seconds, moving the layout of the page, which negatively impacts the CLS metric. It must be said that this metric is the result of very complex calculations, but fortunately, the Qwik team is very thoughtful about these metrics because they know that they are very important. In fact, they have developed Vite plugins to make our lives easier. There is a specific check for CLS that visually warns us on our page of movements that will penalize us and create a bad result for this specific metric. There is an out-of-the-box optimization to ensure that the right attributes are assigned (e.g., srcset) to optimize loading and possibly perform lazy loading to obtain excellent performance with images.

The solution integrated by Qwik is based on the `vite-imagetools` library, so it is not necessary to install additional packages or components in the application. This integration allows any local image to be imported into the code, and it will automatically be converted into multiple WebP images, one for each breakpoint (200px, 400px, 600px, 800px, 1200px), and processed to reduce its size. The application will contain an `<img>` element at the end of the process, and through the srcset attribute, the image origin is set for different solutions, so the browser will load the image best suited to the user's resolution.

This integrated solution offers several advantages:

- No JavaScript is required in the client to perform these transformations
- Zero layout shifts (automatic width/height)
- Hashed, immutable cached images
- Automatic optimization of the .webp/.avif format
- Automated srcset generation
- Extensible (use any `<img>` attribute)

Let's see an example:

```
// Add the ?jsx suffix at the end of the import
import Image from '[IMAGE_PATH]?jsx';

export default component$(() => {
  return (
    <div>
      // Use the image in the template as a component
      <Image />
    </div>
  );
});
```

This script will generate the following `<img>` element in the HTML page:

```
<img
  decoding="async"
  loading="lazy"
  srcset="
    /@imagetools/141464b77ebd76570693f2e1a6b0364f4b4feea7 200w,
    /@imagetools/e70ec011d10add2ba28f9c6973b7dc0f11894307 400w,
    /@imagetools/1f0dd65f511ffd34415a391bf350e7934ce496a1 600w,
    /@imagetools/493154354e7e89c3f639c751e934d1be4fc05827 800w,
    /@imagetools/324867f8f1af03474a17a9d19035e28a4c241aa1 1200w"
  width="1200"
  height="1200"
>
```

Note that `decoding="async"` and `loading="lazy"` are the default.

- **decoding=”async”**: indicates that the image will not block the rendering of the page while the image is being decoded. For further reference, please check the MDN web docs[27].
- **loading=”lazy”**: allows the browser to delay the loading of an image until it is visible in the viewport, which helps improve page loading performance
- **srcset**: allows you to choose the most appropriate image based on the resolution and size of the device screen. Setting width and height attributes prevents layout reflow, which hurts the CLS score.

The default decoding and loading behavior can be changed by manually setting them, and the width and height can also be manually specified:

[27] https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement/decoding

```
// decoding and loading
<Image decoding="sync" loading="eager" />
// width & height
<Image decoding="sync" loading="eager" style={{ width: '300px', height: '200px'}}  />
```

There are also integrations for optimizing images loaded from external sources. There is the `qwik-image` package developed by Qwik and another package called `@unpic/qwik`. Both libraries offer the possibility to use external images and perform optimization via CDN services such as Cloudflare, Builder.io, and many others. As mentioned before, it is very important that the image already takes its place on the page to avoid annoying movements that penalize the user experience and the CLS. With qwik-image[28], a placeholder can be rendered behind the scenes, which will be replaced as soon as the image is loaded. This trick guarantees an optimal CLS result.

First Input Delay (FID)

First, Input Delay is a very important metric that measures the time that passes between the user's iteration (e.g., click on a button or link) and the browser's response to this iteration. Some events are ignored in this calculation. For example, zooming or scrolling are not taken into account when calculating the FID. A good value for this metric is within 100 milliseconds, which means that users should be able to interact with the page immediately. This metric only takes into consideration the first input delay because, often, the first feedback received allows us to create an overall impression of the quality and reliability of navigation. The main problem that does not allow good results for this metric is loading too much JavaScript upfront. With Qwik, as analyzed in the previous chapters, all this is resolved by the framework thanks to Resumability, so an exceptional FID can be achieved. This metric has been replaced by `Interaction to Next Paint` in March 2024 because the latter is entirely effective.

Interaction to Next Paint (INP)

This metric replaced the First Input Delay (FID) in the Core Web Vitals and evaluate responsiveness, recording the latency of all interactions throughout the page lifecycle. We can obtain this metric by measuring the time between the user's action (e.g., key press, click) and the subsequent interface update. The highest value that is recorded among all interactions is the one considered to record the INP value. As we can imagine, a low value indicates that our page is responsive and therefore always responds reliably. This is a metric that measures the entire process of our application. Measure all interactions, not just the first one the user performs. It's a well-rounded metric just like Cumulative Layout Shift (CLS). So by measuring this metric we try to highlight applications that have a very short time between interactions and subsequent rendering.

A good value for this metric is less than 200 milliseconds, but if it is more than 500 milliseconds, then it is a warning sign that something needs fixing. Since this metric measures the entire cycle of events that starts from the user's iteration to the user interface update, one of the tips that can be given to optimize this metric is to minimize the size of the DOM because a larger DOM requires more computation time. Furthermore, what is even more important is to measure which operations need to be optimized. If we know who our enemies are, we can defeat them better. Tools like PageSpeed Insights[29] can be our friends in finding what to optimize.

[28] https://github.com/qwikifiers/qwik-image
[29] https://pagespeed.web.dev/

Other Web Vitals

There are other metrics that, although not considered core, can give us an indication of whether our application is good or whether it is possible to improve it. They are not to be absolutely optimized because they are not considered core metrics, but provided that this does not impede the ability to obtain good scores in the metrics that matter. Let's look at these other interesting parameters that help give us a complete picture.

First Contentful Paint (FCP)

This metric measures the time from the page begins loading to when any part of the page's content is displayed on the screen. Any image, button, text, or HTML elements in general, as long as they are visible, are considered. This metric distinguishes from the LCP metric, which instead aims to measure when the main contents have been rendered on the page. A good score for this metric is 1.8 seconds or less. If the application is above this number, there is something to fix. To improve results on this metric, ensure that CSS is minified and all unused CSS is removed. Also, try to remove unused JavaScript. Pre-connecting to the origins that provide the resources requested on the page can also be helpful. Additionally, try to limit the size of the DOM and keep it as minimal as possible.

Time to First Byte (TTFB)

This metric measures the time between requesting a resource and the first byte of a response begins to arrive. A good result for this metric is 0.8 seconds or less. This metric precedes the other analyses that are user-centric, and it measures the waiting time for requests to understand if the user will have to wait. Above 1.8 seconds, the result is considered poor, and in the middle, it needs improvement. Results can be improved by relying on CDNs for the distribution of the resources necessary for the application or by using frameworks that allow HTML streaming to the browser. Qwik is one of those frameworks that can render HTML chunks, so as soon as the browser receives the first block, it will immediately be shown on the page, even if the second block is still being received. This flow is called streaming.

Total Blocking Time (TBT)

This metric measures the time when the browser's main thread gets blocked. Whenever an activity that takes more than 50 milliseconds occurs in the main thread, it is considered blocked because it effectively prevents the responsiveness of user input. Therefore, if a user interacts with the page in the middle of a long activity, the browser must wait for the activity to complete before it can respond. If the activity is long enough (anything over 50 ms), the user will likely notice the delay and perceive the page as slow or unstable. A good user experience can be appreciated if the Total Blocking Time is less than 200 milliseconds. To improve this metric, reduce JavaScript execution time, reduce third-party code, and minimize the work of the main thread.

Time to Interactive (TTI)

This metric measures the time it takes for the page to become interactive to user input. If this metric is unsatisfactory, it indicates that something is not correctly working when interacting with a site. The TTI wait until the DOM construction is complete to consider the page interactive, meaning that it can only be measured after the DOMContentLoadedEnd event. To improve this metric, eliminate any unnecessary scripts in the application or perform asynchronous execution of some elements to free the browser.

Labs Data vs Field Data

The performance of a web app can be measured using various tools. These tools can be differentiated into two types based on the data they extract:

- **Labs data**: This data is excellent for debugging during daily development as it offers immediate feedback and is collected in a controlled environment. When measuring web performance with laboratory tools, the website is loaded into a remote device, and a good connection speed and the metrics mentioned above are measured, among other things. Labs data is useful for reproducing and debugging potential performance issues, but it doesn't provide real data about users.
- **Field data**: Also known as RUM (Real User Monitoring) data, field data is the collection of real user information. It helps understand what real-world users are actually experiencing on an application and often brings to light problems that are difficult to capture in a laboratory setting. The results can be influenced by many external factors, which have no possibility of being measured in local tests. For example, different network conditions, data received from different devices, and different geographical locations, in short, a whole series of variables that are often outside of direct control.

  Search engines use field data for ranking and to analyze the user experience on our application, so although lab data is important in day-by-day development, the real ones make the difference.

Why can Labs Data and Field Data be Different?

It is normal to obtain different values when comparing these data because real data measures data from various sources in different scenarios and is, therefore, more heterogeneous. Field data comes from real users, while lab data comes from a fairly powerful machine with a good connection somewhere in the world. Normally, some discrepancies in the metrics might be observed. As seen earlier, there may be users browsing a web application on different devices, and with both fast and slow network connections, and the field data reflects this. Real data might reveal a problem, but lab data does not show anything to improve, which can create confusion.

Labs Data vs Field Data

- Simulated and artificial instead of real users
- One generic network connection instead of several network connections
- A single device is chosen instead of several devices
- A single default location instead of different geographic locations

How to Collect Labs Data and Field Data

The best way to monitor an application is to use Lighthouse[30] or PageSpeed Insights[31]. These tools provide a quick test to extract lab data to get a quick overview of performance. PageSpeed Insights also collects real user data in the first upper section. But how is this data collected? Measurement is based on users who have

[30] https://developer.chrome.com/docs/lighthouse/
[31] https://pagespeed.web.dev/

opted-in to browsing history sync, have not set a sync passphrase, and have enabled usage statistics reporting. If all these configurations are set, then the user will submit their data to feed this metric. This means that the user must visit the site using Chrome and have the correct configuration to send their real data. So, if there are visitors to an application, these metrics may not necessarily be available as a result.

PageSpeed Insights also differentiates the results between Mobile and Desktop, so the mobile part may be populated while the Desktop part does not have enough data to provide a result. All this data is stored on the Chrome User Experience Report (CrUX) and is accessible through various tools and APIs. It is important to note that it only provides information about users using Chrome and not other browsers.

Summary

In this chapter, many concepts that are often overlooked or insufficiently examined have been explored. The analysis began by examining how search engines index web applications. However, it was learned that the algorithms that evaluate and compile the results are not public. Therefore, based on experience and the information shared directly by users, algorithm owners, or SEO experts, the necessary adjustments to the web application can be made.

With Qwik, there is an advantage because the search engine dedicates a limited amount of time to each site. Being able to return pages quickly gives an edge. Offering interesting and attractive content is the fundamental basis for keeping users satisfied. Consequently, the `bounce rate` and `dwell time` metrics also benefit from this. These metrics calculate how much time users spend on the platform or whether they leave after a few seconds. This provides search engines with crucial information because it indicates that the search result (e.g., the web application) is valid and can be suggested again in the reliable results for other users.

To facilitate the scanning of the application, it has been learned that the use of semantic HTML is very important. Based on the tags used, what the HTML will contain can be implicitly understood. For instance, it is known that the nav tag will contain the HTML for navigation. This aids not only search engines but also screen readers, which many people use to consume web content. This allows for full accessibility and inclusivity to everyone. Moreover, the issue of accessibility is increasingly important because there are very specific laws worldwide that require public sites to be accessible to ensure equal opportunities for all.

It's been learned that special attributes in links can be used to communicate intentions explicitly to search engines. The structure and importance of the `sitemap.xml` and `robots.txt` files were also examined. These files can improve the position in the SERP.

During the chapter, an introduction was made to an innovative library, Partytown, which can manage the execution of third-party scripts using the web worker. This is crucial for offloading the main thread of the browser. By doing this, there will undoubtedly be better Web Vitals, another topic of this chapter. Web Vitals are a series of metrics that analyze the application and help understand if a good experience is being provided to the user. Three main metrics are most important to monitor and optimize, but the others are also not to be overlooked. Search engines consider all these metrics when indexing the application and determining its ranking in user searches.

Some tips for improving these metrics were also learned. Fortunately, with Qwik, there won't be problems achieving the maximum scores in all the above categories. This is because Resumability and the best practices that the Qwik team encourages to follow help achieve exceptional results, which are difficult to attain with other frameworks.

In the last part of the chapter, the difference between lab data and field data was discussed. These types of data serve different purposes, primarily to understand why the application does not have good indexing. Field data are particularly useful because they record the real use of the application and provide a global view under different hardware and network conditions. The next chapter will explore how to deploy the Qwik application and discover the advantages of cloud deployment versus on-premise server deployment.

Deploy Qwik in production

Deploy Qwik to Production

Understanding the importance of a well-configured, high-performance, and searchable site is crucial. However, the most critical aspect for all users is the ability to publish an application on the web. This ensures that the application is accessible to everyone, including desktop users, mobile users, and many others. In the past, an on-premise solution was used to deploy applications. More recently, Virtual Private Servers (VPS) have been utilized, effectively offering a virtual server where an application can be hosted.

VPS and Housing Solutions

As mentioned, this method of publishing applications was widely used in the past. For some companies, it is still the most appropriate way to publish their applications. It should be noted that with the advent of the Cloud, this type of solution is, in some opinions, outdated. However, the fact remains that having full control of the configuration and complete management of the operating system on which an application runs is an essential necessity for many.

Delving into what it means to have full control of the operating system, the `Virtual Private Server (VPS)` is a type of hosting that allows the purchase of a virtual server and access it with specific credentials to manage it. This gives the ability to install software and updates based on specific needs. An application can be installed to make it publicly accessible and available at any time.

The VPS approach offers full control and guarantees optimal performance based on the plan chosen to purchase. It is the responsibility of the user to select the most suitable size for the server. Here are some sample sizes for a popular VPS provider:

- **S Plan**: 1 vCore, 2 GB RAM, 40 GB SSD, 250 Mbps
- **M Plan**: 2 vCore, 4 GB RAM, 80 GB SSD, 500 Mbps
- **L Plan**: 4 vCore, 8 GB RAM, 160 GB SSD, 1 Gbps
- **XL Plan**: 8 vCore, 8 GB to 32 GB RAM, 160 GB SSD to 640 GB SSD, 2 Gbps

There are solutions for all needs, and plans can also be changed on the fly if necessary. Costs were deliberately not included because they change rapidly, and each seller has their dedicated offers.

Generally, the parameters to consider are mainly the allowed disk space, the amount of RAM, the type of processor, and the available bandwidth, both in terms of size and availability. These servers can be physically located around the world and can be physical or virtual. They are usually very powerful machines on which dozens or sometimes hundreds of virtual servers can be shared.

Personal space on the server is shared with other users. However, the various spaces are well isolated, so there is no perception of being on a shared server. Each user has their own space and full control of the machine.

This is guaranteed because virtual machines are used, a technology that allows running a virtual instance and provides virtualized server resources on a physical server shared with other users.

Most of these VPS are instances with a Linux operating system because the graphical interface is unnecessary to make them work. To publish an application, creating a web server to expose it to the public will be enough. Sometimes, however, choosing Windows or macOS machines is also possible. But in general, the preferred distribution can be chosen for all solutions.

For example, a Linux server can be used to install a web server and publish a site. By doing so, not only which web server to use (for example, Apache or Nginx) can be chosen but also the technology, in this case, Qwik. Another advantage is that access to these instances is password-protected, preventing anyone from sabotaging the system. They always have guaranteed uptime and are often economical and functional solutions for small sites with limited visitors. However, databases can be configured and used, and usually, with the most complete plans, it is possible to activate automatic and recurring backups to protect data. However, there may be limitations on the installed software, so it is a good idea to check carefully and see what is possible in these instances.

These virtual machines are usually equipped with a public IP, which allows an application to always be reachable at the same address. It will be enough to register the domain chosen for the application to always be able to reach it. e.g., `https://www.my-website.com`. In general, however, it must always be considered that the physical server is shared with other service users. The alternative to having a server always active and functioning is housing.

The `housing` solution is very similar to the one seen previously, except that the server on which an application is run is owned by the user and is located in the supplier's data center. The supplier will offer connectivity, fire systems, UPS, and so on. Clearly, for both solutions, technicians are needed to install the various packages and maintain the system from a software point of view. This requires a significant amount of energy and time. The alternative is to use "managed" servers where all these activities will be taken care of by the supplier, who will then invoice the user.

These solutions are quite complex, and nowadays, there are better choices for deploying applications. This is especially true if a startup is involved or there is a need to validate a business idea or simply implement a side project.

Deploy to Server

The numerous integrations present in Qwik can be taken advantage of to perform deployment on a server instance. These integrations can be used via the command present in Qwik. In fact, it is just necessary to move to the folder of the Qwik project and type from the terminal:

```
npm run qwik add
## or
pnpm qwik add
## or
yarn qwik add
## or
bun qwik add
```

Within a Qwik application, a list of integrations will be offered that allow the configuration of the deployment of an app very simply. In this list, various integrations can be found to be able to deploy on a server.

```
- Adapter: Node.js Server
- Adapter: Deno Server
- Adapter: Node.js Express Server
- Adapter: Node.js Fastify Server
```

Let's see together how to deploy with the `Fastify` web framework, which stands out for its simplicity and extreme speed, making it the point of reference.

> Using a framework instead of developing everything by hand guarantees us, first of all, that we are covered from the security point of view; furthermore, directly from the framework documentation, we can have the directives on how to develop with the best practices that allow us to create efficient code. Many organizations use `Fastify`, and the project is part of the OpenJS foundation and is included within the At-Large projects, which are those that are considered important for the general ecosystem. The OpenJS foundation aims to support the healthy growth of JavaScript and web technologies.

By selecting the `Adapter: Node.js Fastify Server` from the list seen above, an application will be configured automatically. Here's what will be seen in the terminal.

```
□  □  Ready? Add fastify to your app?
|
|  □ Modify
|     - package.json
|     - README.md
|
|  □ Create
|     - src/entry.fastify.tsx
|     - src/plugins/fastify-qwik.ts
|     - adapters/fastify/vite.config.ts
|
|  □ Install pnpm dependencies:
|     - @fastify/compress
```

```
14 │     - @fastify/static
15 │     - fastify
16 │     - fastify-plugin
17 │
18 │  New pnpm scripts:
19 │     - pnpm build.server
20 │     - pnpm serve
21 │
22 ┌    Next Steps  ──────────────────────────────────────┐
23 │                                                     │
24 │  Now you can build a production-ready Fastify app:  │
25 │                                                     │
26 │  - pnpm run build: production build for Fastify     │
27 │  - pnpm run serve: runs the production server locally  │
28 │                                                     │
29 ├─────────────────────────────────────────────────────┘
```

It can be read that dependencies will be automatically added to a project, some configuration files, and two extra scripts that will allow building a project using `Fastify`. Here, starting the `build` script will run the build in a project so it can then be moved and run on a server. The script `"serve" : "node server/entry.fastify"` can be analyzed, and it can be seen that to run an application in production, it will be sufficient to use Node.js and launch the `server/entry.fastify` file with `node`, which is produced by the build. Everything needed to run the application is inside the server folder. It can be moved to a server in VPS or Hosting, and the Fastify server can be run there. The only thing to remember is to install the dependencies required by the script to run correctly. In fact, the server imports Fastify, and if it hasn't been installed, the server won't start.

Once the Fastify server is up and running, the application will be available on port 3000 (the default one). All that needs to be done is to configure Apache or Nginx to serve the application to the public address of the domain. e.g., `https://www.my-website.com`. To handle multiple requests and take advantage of all the server's CPUs, the Node.js cluster[32] API can be used to run multiple Node.js instances that can distribute workloads across application threads with process isolation.

Scalability with Clustering

Since Node.js uses a single thread to run a process, all requests are handled by a single CPU. This can lead to an overload of the single process, which can slow down performance if it handles too many requests. If the process crashes due to an error, users can no longer access the application. Fortunately, the Node.js cluster[33] can be taken advantage of, which allows scaling applications on the same machine and running them simultaneously. A load balancer is also provided behind the scenes to level the load evenly via the `round-robin`[34] algorithm. So if an instance crashes, users can be served by the other processes that are still available. Furthermore, the instance that crashed can be rerun to restore the situation to the optimal one.

[32] https://nodejs.org/api/cluster.html#cluster
[33] https://nodejs.org/api/cluster.html#cluster
[34] https://en.wikipedia.org/wiki/Round-robin_scheduling

It has been seen that it is possible to start the Fastify server thanks to the `node server/entry.fastify` command. So, in effect, the code that allows allocating a CPU to run that script is being executed. Let's try to use the Node.js cluster. Below an example will be seen that is not intended to cover all possible cases but simply to give an idea of how to best resolve the situation.

Here is an example:

FILE: `server/my_cluster.js`

```
import cluster from "cluster";
import os from "os";
import { dirname } from "path";
import { fileURLToPath } from "url";

// The total number of CPUs is 8
const cpuCount = os.cpus().length;
const __dirname = dirname(fileURLToPath(import.meta.url));
const entryPoint = "/entry.fastify";

// `setupPrimary` changes the default 'fork' behavior.
cluster.setupPrimary({ exec: __dirname + entryPoint });

for (let i = 0; i < cpuCount; i++) {
  // Starting worker
  cluster.fork();
}

cluster.on("exit", (worker, code, signal) => {
  // worker worker.process.pid has been killed
  // Starting another worker
  cluster.fork();
});
```

Let's analyze this file. `cluster` is being imported, and the number of CPUs is being obtained. In the machine where the script was run, there are eight of them. The file that allows starting the Fastify server, the famous file seen previously, is then defined. Using `setupPrimary`, the behavior of the `fork` command that will be used in the following lines is configured. In fact, for each CPU, `cluster.fork()` will be run, which allows starting an isolated server instance. Finally, the `exit` event is listened to restart a new instance and thus restore full functionality. It is clear that if the server fails due to a bug, continuously restarting the instances will not solve the problem.

To start this script, simply edit the `serve` script like this: `"serve": "node server/my_cluster"`. In this way, the desired result is achieved. It is obvious that having more instances allows for handling more requests. It would not make sense to insert benchmarks calculated on a local machine. It is obvious that one instance is less effective than eight. It should be pointed out that this solution is perfectly functional, but it is not ready for production, so it should still be refined as best as possible.

All this configuration requires time and technical skills, but it is excessive if simply having to validate an application or if a startup wants to start its own business. In recent years, a series of cloud web hosting services and automation platforms have been created that accelerate development productivity.

Cloud Solutions

Cloud solutions platforms are a recent development in the evolution of the web. Many companies have emerged to address the complexities of deploying applications. As seen in previous sections, numerous factors are at play to make an application accessible to everyone via the web. These platforms automate all the necessary steps, saving valuable time to focus on application development and implementing new features for the business. Deploying to these platforms is very straightforward. From the list of integrations available for Qwik, one can choose the one that allows deployment to Vercel. By typing `pnpm qwik add vercel-edge` in the terminal, this output will be received:

```
□  □  Ready? Add vercel-edge to your app?
│
│  □ Modify
│     - package.json
│     - .gitignore
│     - README.md
│
│  □ Create
│     - vercel.json
│     - src/entry.vercel-edge.tsx
│     - adapters/vercel-edge/vite.config.ts
│
│  □ Install pnpm dependency:
│     - vercel
│
│  □ New pnpm script:
│     - pnpm build.server
│     - pnpm deploy
│
□    Next Steps  ──────────────────────────────────────────────────────────────□
│                                                                              │
│  Now you can build and deploy to Vercel with:                                │
│                                                                              │
│  - pnpm run build: production build for Vercel                               │
│  - pnpm run deploy: it will use the Vercel CLI to deploy your site           │
│                                                                              │
├──────────────────────────────────────────────────────────────────────────────□
```

As with the previous integration, some configuration files are automatically added to the application, dependencies are installed, and two extra scripts are added. Now, the `build` script can be run to package everything for publishing the production application. All that's left is to run the `deploy` script, which will use the Vercel CLI to operate.

Obviously, to deploy with Vercel, you need to have a working account.

```
Vercel CLI
? Set up and deploy “~/qwik-app”? [Y/n] y
? Which scope do you want to deploy to? giorgioboa
? Link to existing project? [y/N] n
? What’s your project’s name? qwik-app
? In which directory is your code located? ./
Local settings detected in vercel.json:
Auto-detected Project Settings (Qwik):
- Build Command: vite build
- Development Command: vite --port $PORT
- Install Command: `yarn install`, `pnpm install`, `npm install`, or `bun install`
- Output Directory: dist
  Linked to giorgioboa/qwik-app (created .vercel)
  Inspect: https://vercel.com/giorgioboa/qwik-app/xxxx [1s]
  Production: https://xxx.vercel.app [43s]
```

As can be seen, some questions will be asked, but subsequent deployments will start immediately without requiring further information. At the end of the process, the public address of the application will be displayed. That's the advantage of these modern platforms - having an application up and running is fantastic. The ability to deploy with a terminal command is truly remarkable.

Platforms such as Vercel, Netlify, and Cloudflare, to name a few, go even further. It's possible to connect pipelines directly to a GitHub repository to carry out automatic deployment every time something changes in the code and, consequently, in the GitHub repository. Environment variables can be declared to parameterize the code and centralize constants such as private keys or external URLs. It's also possible to create different environments to verify developments in the classic staging environment. This allows deployment in various instances, providing maximum flexibility and agility.

Through the dashboard, the various applications that have been published can be monitored, traffic can be analyzed, and the storage used can be assessed. Some providers also provide a history of all the operations carried out to keep everything under control. Vercel offers a unique feature that allows the purchase of a public domain directly through their platform to be associated with the application. If there are no preferences, their convenient service can be used to avoid finding the best supplier from which to purchase the domain.

In the monitoring pages, there's also a section regarding the Core Web Vitals, the famous metrics that have been discussed in previous chapters. Thanks to this tool, the app's progress can always be well-informed, and there will be no excuses for not monitoring these important parameters.

All these services and features are immediately available, thanks to the platform. Otherwise, everything would have to be implemented manually in a customized way. Creating these solutions from scratch may be considered challenging and not without costs. It's important to weigh the pros and cons of the various solutions and then decide on the one that best suits the needs.

For example, an external service like `Sentry` could be used if there was a desire to implement a monitoring system. This service offers the ability to collect application logs and any errors through a quick and easy SDK. This paid service guarantees reliability and resilience and offers the assurance of always being available to receive events, logs, or errors from the application.

Here, too, payment for an external service is still required, and therefore, in terms of costs, the situation needs to be assessed on this occasion as well.

Branching Strategies

Once the deployment system is set up, multiple environments will likely be needed to test the application. Cloud solution services already offer the possibility of creating multiple environments for deployments. Differentiation of code with environment variables is possible, so instead of using the production APIs, the test environment ones can be used. Keep in mind that this feature is also possible with other solutions. This guarantees security because, for example, users can be created and data modified without creating problems in the production environment, but everything can be tested in a protected environment, the test environment. The same thing can apply to the dev environment and other cases. With Cloud solutions, it takes just a few clicks, while recreating the various environments with the Virtual Private Server (VPS) or housing solution will require more effort. To test newly developed features, a development environment will be needed, but not only that, the ability to version code to keep track of all the changes will also be necessary.

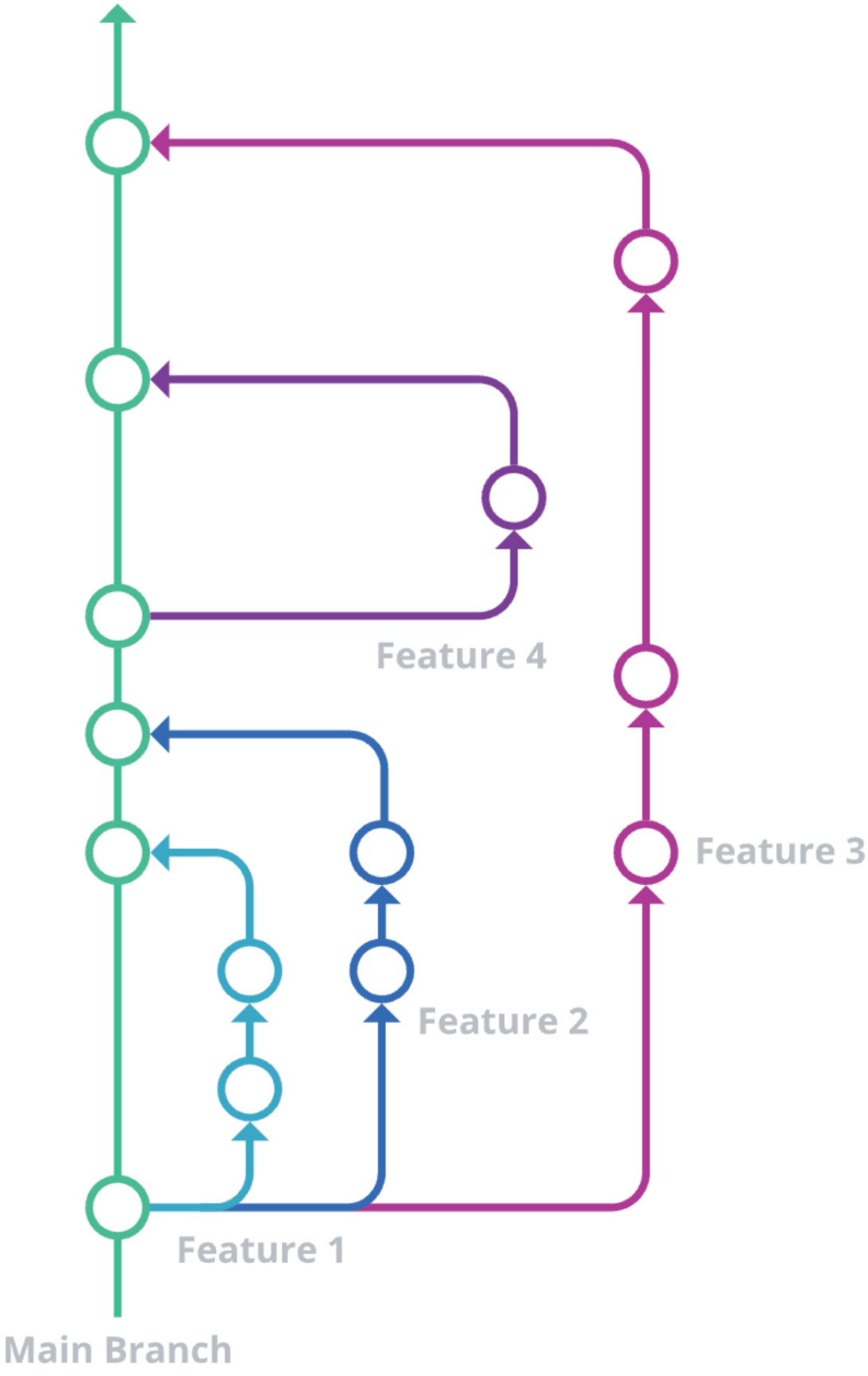
Nowadays, practically everyone uses Git[35] as a versioning method, and it has become the standard precisely because it is very simple and well-documented. This ensures that changes are never lost, and the author of a certain codebase line can be quickly identified. Furthermore, by versioning the code, the complete history of the application can be maintained. Going into detail about Git would take another book to delve deeper into the topic, but to better manage the fact that different versions of the application can be tested in various environments, it is important to know that there are different methods for organizing versioning of the projects. However, the focus needs to be on the term `branch` because it will be used often in the next paragraphs. When using `Git` to version code, a primary branch (typically named 'main') serves as the repository for all code changes. With each subsequent process, a chronological record of saved versions is established.

A branch is a parallel branch detached from the main one. This branch has a life of its own and is used to create new features or resolve bugs, in short, to make changes in the code. Imagine having to change the background color of a page from white to black. From the main branch, a new branch is created. On this branch, there will be the possibility to change the color and try the modification locally. As mentioned, this does not affect the main branch, which will remain with the old background color, i.e., white. Once the change is declared ok, the changes made can be merged into the main branch. This type of operation has allowed other developers to work in parallel because, in the meantime, other features have been precisely implemented. After all, this methodology facilitates collaboration on the same code base.

[35] https://git-scm.com/doc

GitHub Flow

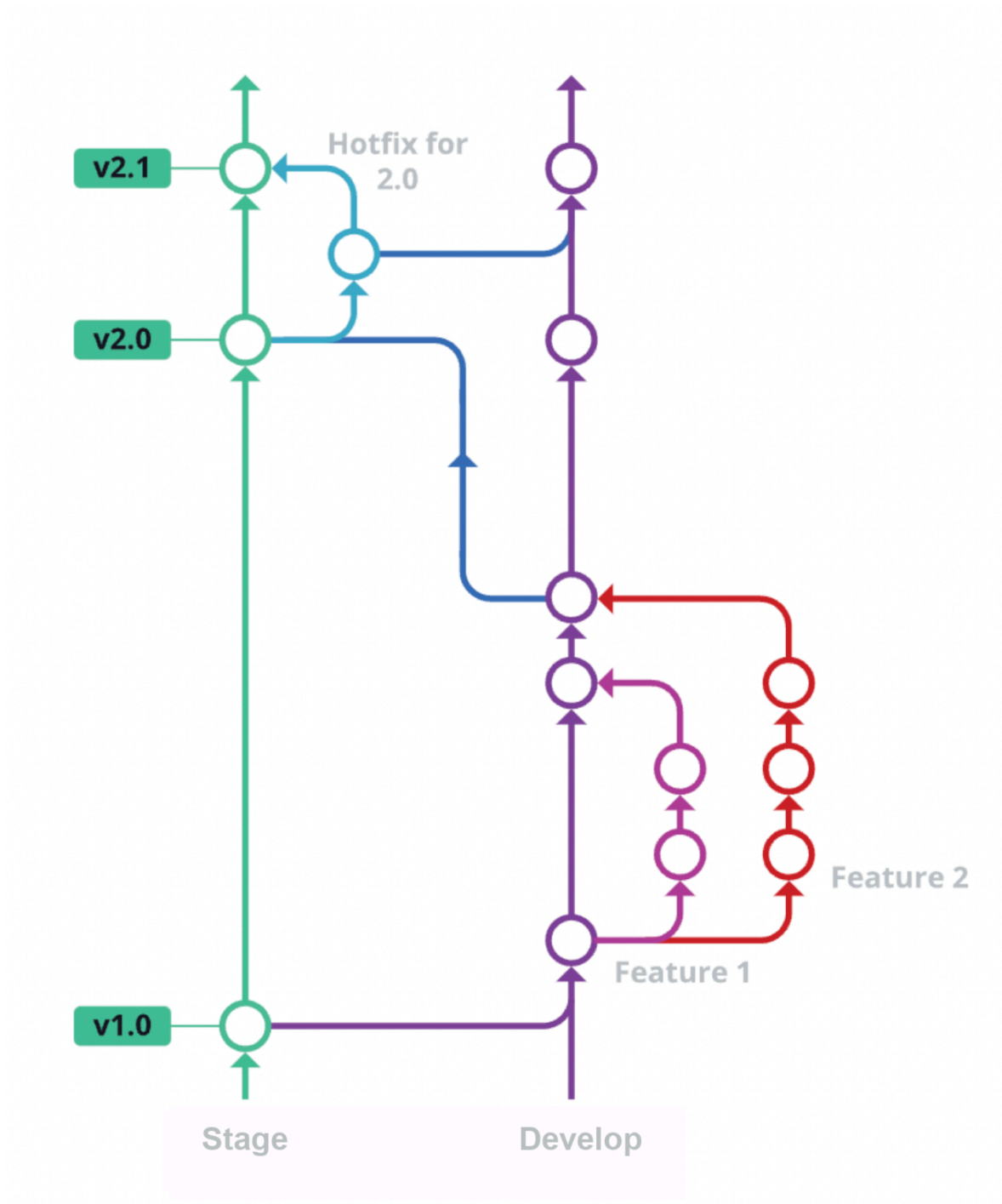
Start by analyzing GitHub Flow, also called Trunk Based Development, a versioning flow that does not lend itself well to managing multiple branches, but precisely because of its simplicity, it is the right hook for subsequent flows.


Feature 4
Feature 3
Feature 2
Feature 1
Main Branch

This way of versioning code is certainly the simplest. As mentioned before, there is a main branch. For each feature that is requested, a branch `Feature 1`, `Feature 2`, `Feature ...` is created, the modification is implemented in an isolated and independent manner, and then once satisfied with the work, the changes can be included in the main branch. In the figure, it is seen that by doing so, development can occur in parallel, and the main branch will remain unchanged until the changes are transferred. So, with this approach, the main branch contains the code that is in production, and as changes are included, production will also be updated. The advantages of this way of working are simplicity and agility, and since every change is included punctually, there is maximum alignment precisely because the code included from the various branches is limited and punctual. On the other hand, having multiple control environments is impossible; the modification is included directly in the main branch.

GitLab Flow

This way of managing versioning is more complex than the previous one, but it allows the goal of having multiple environments to be able to test the application to be achieved.



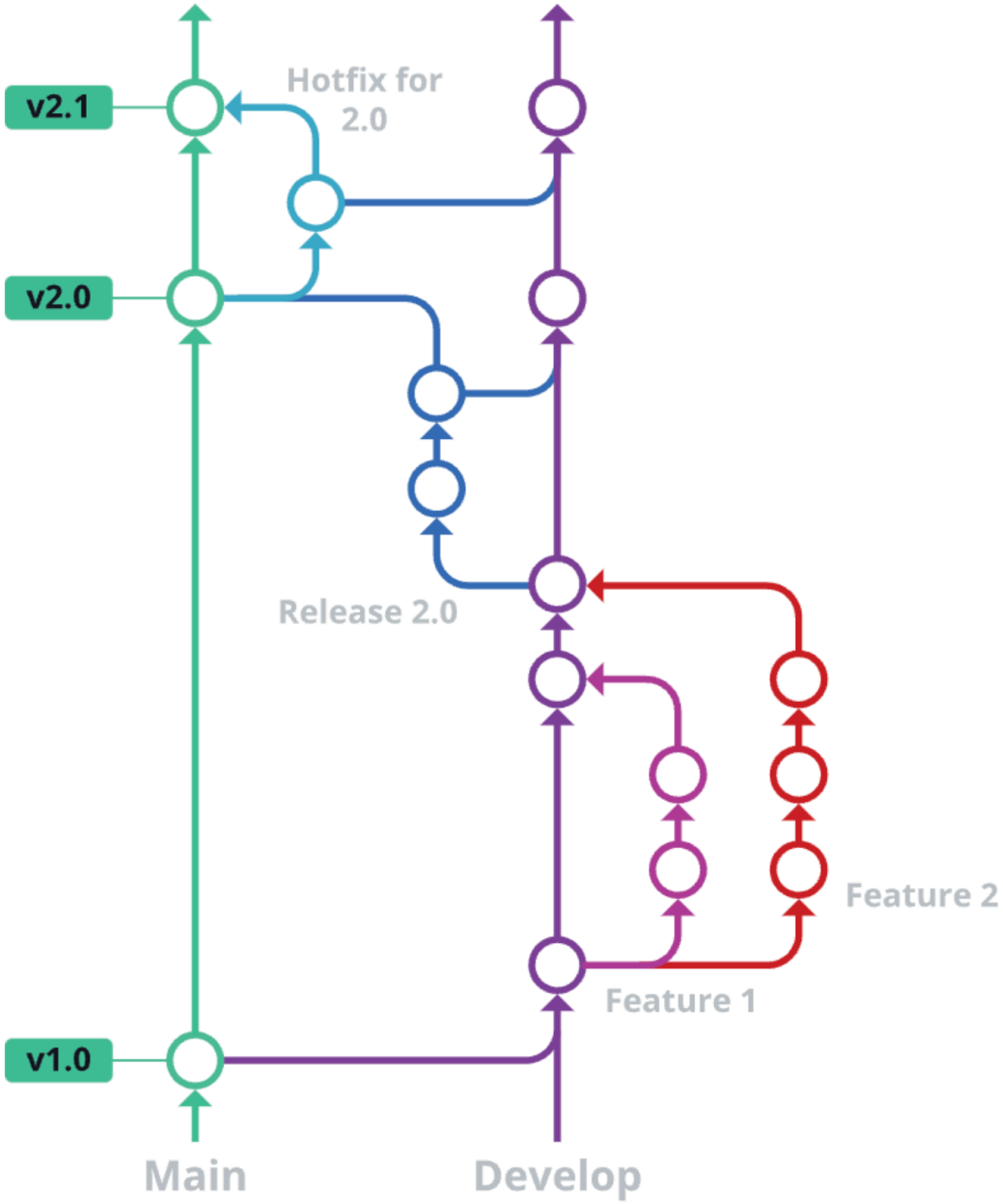
gitlab-flow

There are two parallel branches, which are respectively those of `Develop` and that of `Stage`. These are the names of the two environments that are desired. There will also be the production line, which will be a third parallel line, but the focus is narrowly on two environments to give the idea. There are these two parallel histories, and features are added via branches `Feature 1`, and `Feature 2` in the `Develop` branch. To this

branch, it is possible to attach a pipeline that says: Every time something changes, execute a deployment in the `Develop` environment. By doing so, there will be the possibility to go and check the application to understand if everything works as desired. Once satisfied with the changes, the code can be dumped into the `Stage` environment, often also called `Test` or `QA` (Quality Assurance). Here, too, through a pipeline, the environment will automatically be updated, and it will be possible to run tests to ensure everything works. The `Develop` and `Stage` environments are separate; they will write and read from separate external sources (database, API, etc., etc.), allowing maximum isolation. Usually, at each alignment of a `Stage` branch, a release can also be created for the application and, therefore, a version advancement. If changes are needed to the code in the `Stage` branch, a `hotfix` can be run starting directly from the version that suffers from the problem. It should be noted that the `hotfixes` are also developed in separate branches, and once the changes are confirmed, the hotfix will be transferred to the `Stage` branch. The developments in the `Develop` branch have continued, but it is necessary to include the `hotfix` also in the `Develop` branch to avoid the problem recurring in a subsequent release. This whole process can be repeated for each specific environment; therefore, even if it is not present in the photo, a `main` branch will contain the production code. The same hotfix and code transfer reasoning seen previously also apply to any parallel environments that are desired to be created. The production deployment that points to real data will then be attached to the `main` branch, so the users can use the application. The benefits of this mode are many, but the greatest is certainly that of finally having separate and isolated environments from each other.

Git Flow

This versioning method is the most complex of those seen so far. It was created for large code bases and to guarantee better control and precision of the code base.



git-flow

This methodology is used in large companies, and all the major tools support Git Flow, but precisely because it is complex, it requires an extra Git tool to be able to manage it better. Start by saying that it is similar to the previous one. However, if you want to be a purist of this methodology, very specific names have to be given to the various branches. In this way, the scope of the individual branch is clear to everyone. Furthermore,

once the code is ready for a release, a release branch will be created. In this release branch, any changes are then made if necessary. GitLab flow and Git flow are very similar, but as previously mentioned, Git flow forces the adoption of some best practices that guarantee precision but add some extra steps that weigh down and slow down the process. This type of methodology is not suitable for everyone. If agility and speed are needed, it is not ideal, but in large companies where a certain structure is needed, and many developers work on the code, it is certainly a choice to consider.

Progressive Web App (PWA)

Nowadays, real data collected in the field confirms that more than 65% of individuals connect to the web via mobile devices. However, another fact that warrants further consideration is that 73% of purchases for luxury goods, pharmaceuticals, cosmetics, and much more are made via mobile. Therefore, ensuring an optimal user experience for mobile is essential for business success and can make a significant difference compared to competitors.

Once an application is publicly available, it likely has a public address that uses the `https` protocol. This protocol guarantees that the application uses secure internet communication. Using this protocol also indirectly enables support for the Progressive Web App (PWA) mode, which improves the experience of users who use the mobile application.

PWAs are a hybrid between web applications and mobile apps. This technology uses the browser and does not require installation via the app store, which is why it is often preferred over native apps. Users start using the web app via the internet, and when they decide that it is useful, they can install it among the icons of the mobile phone by pressing a button. Data shows that, apart from the famous apps already pre-loaded on mobile phones (such as WhatsApp, Facebook, YouTube, etc.), people tend not to install new apps, especially if they are not already famous. Consider Amazon, for example. Initially, individuals buy two or three products from the web application, and only when they find it useful do they decide to install the mobile app. Numerous advantages are present in web applications in all respects.

- **Look and feel**: As with native apps, an icon on the homepage and on the application menu of the device, which is highly customizable, is present. Thanks to a very granular configuration, it is possible to define many features that bring the user experience and graphic perception of a native application closer.
- **Easy to update**: Like a normal web application, to release an update, it will be sufficient to deploy changes, and they will immediately be available to the end user. There is no need for versioning; for example, when going through the stores of smartphones, the user will simply see the updated application.
- **Universal codebase**: With this approach, it is not necessary to have dedicated development teams. Usually, when wanting to create native applications for iOS and Android, separate codebases are needed to manage the two developments. This approach quickly turns into a problem because the codebases, if not managed diligently, will begin to be misaligned, both in terms of logic and in terms of graphic appearance. Imagine implementing access to the camera in the two code bases, clearly being two different languages. There will be two implementations, and the same will happen for the graphics part. Having the two pixel-perfect solutions will be impossible, and there will always be differences. Over the years, hybrid approaches have emerged to address challenges on both iOS and Android using a unified code base. While effective in resolving previous issues, it requires an additional code base to implement the web application alongside the existing one.

- **No alley**: Native applications, as well as those downloadable from app stores, are subject to fees, especially if they generate income. Typically, stores impose a 30% commission on app purchases, which, though high, is currently unavoidable. The PWA (Progressive Web App) approach provides an alternative, circumventing the entire store system. Additionally, when publishing an app through traditional channels, it must undergo checks by store managers to ensure compliance with predefined rules. However, with the PWA approach, these checks become unnecessary, as the application can be installed directly from the browser.
- **Offline support**: In fact, thanks to the service worker, full support so as not to depend entirely on the internet in case of poor connectivity is provided.

There are also downsides to using PWAs that need to be taken into consideration. Native applications usually consume less battery than their PWA counterparts. This is because a PWA is embedded within a browser instance. The second negative aspect is being inside the browser; full access to the native APIs of the device is not available and will always be limited by the use of those provided by the web browser. Today, it is possible to do many things with the browser API, and interesting features are released with each new version. The capabilities of PWAs can be checked at this site `https://web.dev/learn/pwa/capabilities`, and thanks to this overview, the best choice for a specific use case can be understood.

There is not much to configure to prepare a Qwik application to also be a PWA. A configured service worker is already present thanks to the mental model of Resumability. The presence of this file is an essential condition for creating a PWA. The `manifest.json` file is also necessary to best configure everything, but already in the starter offered by the Qwik CLI, this file is present and functioning. It just needs to be modified to add the missing properties to support PWA.

Here is an example:

```
{
  "$schema": "https://json.schemastore.org/web-manifest-combined.json",
  "name": "qwik-project-name",
  "short_name": "Welcome to Qwik",
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#1D4ED8",
  "background_color": "#fff",
  "description": "A Qwik project app.",
  "icons": [
    {
      "src": "logo-512-512.png",
      "sizes": "512x512",
      "type": "image/png",
      "purpose": "any maskable"
    },
    {
      "src": "logo-192-192.png",
      "sizes": "192x192",
```

```
      "type": "image/png",
      "purpose": "any maskable"
    },
    {
      "src": "logo-144-144.png",
      "sizes": "144x144",
      "type": "image/png",
      "purpose": "any maskable"
    },
    {
      "src": "logo-96-96.png",
      "sizes": "96x96",
      "type": "image/png",
      "purpose": "any maskable"
    },
    {
      "src": "logo-72-72.png",
      "sizes": "72x72",
      "type": "image/png",
      "purpose": "any maskable"
    },
    {
      "src": "logo-48-48.png",
      "sizes": "48x48",
      "type": "image/png",
      "purpose": "any maskable"
    }
  ]
}
```

This file is analyzed in detail. There are some obvious properties which are `name`, `short_name`, and `description`, but others are more specific.

- **start_url**: The start_url is mandatory and indicates to the browser the starting page of the application so our user will always start from the page we have defined. start_url should direct the user directly to the app's homepage.
- **display**: The browser UI can be customized to display when the app launches. For example, the address bar and browser UI elements can be hidden. Our applications can also be launched on the full screen. The display property has these values:
  - fullscreen: Opens the application occupies the entire available viewing area.
  - standalone: it runs in its browser window and hides standard UI elements like the address bar.
  - minimal-ui: The application looks like a standalone app. The user has a minimal set of UI elements to control navigation (such as back and reload).

  - browser: A standard browser experience.
- **theme_color**: Set the color of the toolbar, and a color for the light and dark theme can be defined.
- **background_color**: This is used on the splash screen when the application is first launched.
- **icons**: A set of icons to use on the home screen when launching apps can be defined. Each icon must include `src`, a `sizes` property, and the image type. Finally, there is the `purpose` property, which defines the purpose of the image. The scope can have one or more of the following values, separated by spaces: `monochrome`, `maskable`, and `any`, and are intended to help provide information on what to do if the user uses different icon themes. An icon of at least 192x192 pixels and an icon of 512x512 pixels must be provided. If only these two icon sizes are provided, the others will be automatically resized to fit the device. For a pixel-perfect result, it is advisable to define them as in the example with 48 dp increments.

Thanks to the fact that the application uses `https`, a service worker, and the manifest, it is ready to try this possibility. The install button will already be visible in the browser. By pressing it, the application can be added to a mobile device as if it were a native application. Several use cases support the fact that it makes sense to invest in PWAs. The most famous are Uber, Starbucks, Pinterest, and Spotify, just to name a few. If these large companies have based their business on this method, then it makes sense to do a test to see if it can be successful. After all, thanks to Qwik, it requires zero effort.

Summary

In this chapter, a multitude of topics have been covered. The discussion began by examining various methods for deploying applications. Using VPS or housing is possible if a custom configuration for the server is desired. Consequently, optimal configuration of Qwik with Fastify was explored. It was also analyzed and discovered that Node.js clusters enable scaling of server instances with ease.

Beyond custom server configurations, simpler solutions are available. Using a Cloud platform can significantly reduce workload at a minimal cost, and these platforms often come with enticing free tiers. These platforms also provide a plethora of monitoring and configuration services. Considering the use of external services like Sentry would still require payment for the service. So, why not utilize what is already provided with the Cloud solutions? These solutions allow deployment across numerous environments with ease. Therefore, various methodologies for versioning source code have been examined. Lastly, with the application served via the `https` protocol, supporting the Progressive Web App mode, much like other major companies in the sector, can be considered.

It has been observed that Qwik is essentially already prepared for this mode, enabling support for mobile devices in an optimal manner. In the next chapter, exploration will be done on how to manage the application state within a Qwik application and how to style an application to make it visually appealing.

Style and render data with Qwik

Style and render data with Qwik

The style and ease of use of applications are critical to ensuring that the application gains traction and becomes recognized as the standard for a given domain. You can have the best application logic, but if it is not easy to use or graphically unappealing, no one will use it, and your work will be in vain. Let's imagine having a car-sharing application where you can rent cars, book rides, make payments, know the location of the vehicle in real-time, and much more. On the server side, many variables and logic need to be taken into consideration; just monitoring where a particular vehicle is located in real-time is very complex; let's imagine calculating the rates and checking their future availability. Therefore, on the backend side, the best technologies will be used, and many hours of effort will be spent to understand, analyze, and implement everything in the best possible way. A large part of the budget is aimed at these activities, but if your users are unable to make a booking or, even worse, they are unable to register them on the platform, you can imagine that your business will be affected.

One way to limit this problem is to conduct research by interviewing stakeholders to produce the best possible user experience. One of the next steps is certainly to carry out A/B testing with similar products. This will allow us to immediately receive feedback on the positive and negative aspects, and to correct the application to obtain the best user experience.

Finally, on the graphic side, you can also try to produce modern interfaces that make the use of the application itself captivating and pleasant to unleash the famous "wow effect" in your users.

Stakeholder engagement

To understand how to design an application, it is important to know who will use it, and therefore, it is essential to interview users to understand their needs. So, suppose you are designing a back-office application. In that case, you can think of starting from the users of the same to carefully map the processes they are currently using to perform certain operations. To encourage alignment and dialogue between everyone, it is usually necessary to try to organize meetings with several people, up to a maximum of 7/8 people. It will be your task to understand their needs, and you will have to be careful that everyone can say their opinion.

Sometimes, there are shy people within the group; there are different skills and roles. Your task is to try to involve all the people interested in the application process that you want to analyze, always keeping in mind that everyone's ideas are useful, but if there is a person present who has more decision-making power, it is clear that he will have the final word. It is very important at this stage to focus on the problems or slowdowns that the current flow is causing. The idea is to be able to improve it and make work easier after the introduction of the application. Time-consuming activities such as copying and pasting from one document to another are usually the most time-consuming and prone to human error tasks, and therefore, automating them is certainly a huge benefit for your stakeholders. It would not make sense to cause more slowdowns or problems with the new solution, and therefore, it is very important to be able to collect as much information as possible.

Thanks to this type of analysis, you will be able to get a very precise idea of the various screens that the application must contain, and you will also be able to draw the various flows.

Wireframes

Once you have well-mapped the flow you want to implement, you can move on to creating the first application screens, also called Wireframes. These Wireframes are coarse-grained graphic interfaces that are used to validate the design of information and interactions. In this phase, you should not worry about whether the primary color of your application is better, red or blue, but you should focus on interactions without worrying about stylistic aspects. These first graphical interfaces will be shared with the customer as soon as possible to validate the assumptions and immediately seek feedback from the platform users. It will then be an iterative process that will gradually be refined to lead to the creation of a more fine-grained interface that will lead us to the final solution. In the meantime, as the focus is on the most concrete solution, the domain entities to implement will also emerge. So, from this first purely design-oriented activity, you can slowly move on to the implementation of the code.

With this general overview, you have scratched the surface of this process and allowed you to understand that immediately starting to write code for your application is not the best choice. Many activities can be implemented to clarify the application domain and the needs that your users have. If you don't have the opportunity to interview your users because the application is aimed at the general public, you can implement other strategies. You can analyze the competitors to understand what flows and functions they all have in common, and then, based on those, you can give your approach to the application. Another viable option is to carry out a market analysis on a significant number of users to gather as much information as possible and start your process.

CSS and Built-In Styling Methods

Once you know how your application will look graphically, you can stylize your app using CSS. CSS, which stands for Cascading Style Sheets, is a basic rule that tells the browser how to display HTML graphic content. CSS has evolved a lot in recent years; in fact, nowadays, you can use different tools that adapt to all personal needs and tastes. Let's start by saying that Qwik does not force you towards a specific way of creating CSS for your application, but you can use practically any approach: CSS, Sass, CSS-in-JS, CSS modules, and many others. In this chapter, you will look at these different approaches.

CSS Modules

A CSS module is a CSS file in which all class names are scoped locally by default. Since the syntax to be used is CSS. But Sass, Less, and more can also be used.

> Sass and Less are called "CSS preprocessors" and are an evolution of style sheets, which allow you to use features that would normally be exclusive to programming languages, such as the use of variables and functions. The purpose is to speed up code writing and improve its maintenance. Sass and Less are similar languages and do the same thing, but browsers only read CSS code, which

> is why Sass and Less files must then be converted ("compiled") into regular CSS files. Hence the term "CSS preprocessors".

Let's see an example:

FILE: `button.module.css`

```
.button {
  color: black;
}
```

FILE: `button.tsx`

```
import buttonStyle from "./button.module.css";

export const Button = component$(() => {
  return (
    <button class={buttonStyle.button}>My Button</button>
  );
});
```

From this example, you can see that `buttonStyle` was imported directly from the CSS file and that, then, thanks to the `buttonStyle.button` syntax, you were able to apply the style to your component. This approach is certainly very good because it is based on the standard CSS syntax, and it is also possible to apply Sass, Less, and more. You also don't have to spend time adapting your CSS to the JS like with the CSS-in-JS technique.

CSS-in-JS

CSS-in-JS, in a nutshell, is a different way to style your components through JavaScript or TypeScript. In Qwik, you can use the `styled-vanilla-extract` library, which provides an extremely efficient CSS-in-JS solution without any runtime. Starting to use this approach is very simple because, thanks to the Qwik CLI, you can type this command `pnpm qwik add styled-vanilla-extract` in your terminal, and the entire configuration process will happen automatically. The dependencies will be added and installed to your project, and then you will be ready to build your components.

> The `styled-vanilla-extract` library needs a special Vite plugin, `vanillaExtractPlugin`; this configuration is added automatically by the Qwik CLI process.

Let's see an example:

FILE: `button.css.ts`

```
import { style } from "styled-vanilla-extract/qwik";
export const button = style({ color: "black" });
```

FILE: `button.tsx`

```
import { button } from "./button.css";

export const Button = component$(() => {
  return <button class={button}>My Button</button>;
});
```

Here, you can see that in the style sheet (written in TypeScript), the `styled-vanilla-extract/qwik` library is imported to be able to exploit the style function. By importing the style file from your component, the style is obtained and applied to your button. The positive things about this way of working are that the style is locally scoped, and there is no risk of clashing with class names or more specific selectors, as can happen with the classic CSS approach.

> Not all CSS-in-JS libraries are suitable for SSR and Qwik's way of compiling, and this approach must "serialize" your styles into plain CSS that can be inserted into the document. This is extra work for the browser, which can cause slowdowns in large applications. Luckily, you have `styled-vanilla-extract`, which works well with Qwik.

useStyles$

The native Qwik API `useStyles$` allows for a more standard approach, enabling the import of the style sheet via classic import.

Let's see an example:

FILE: `button.css`

```
.button {
  color: black;
}
```

FILE: `button.tsx`

```
import buttonStyle from "./button.css?inline";

export const Button = component$(() => {
  useStyles$(buttonStyle);
  return <button class="button">My Button</button>;
});
```

You can see that here: the style sheet is going to be imported via `useStyles$` and included in the component. This import will only run when needed, and double loading it in case of SSR hydration will be avoided. By the way, if there was a global stylesheet, it could be loaded directly into the main component. The most observant will have noticed that in the import, there is the term `?inline`. This is very important because it allows Vite to import the CSS file as a string. Note that in the example, CSS was used, but nothing prevents the use of Sass or Less.

useStylesScoped$

It is possible to use the native Qwik API `useStylesScoped$` to add an inline style inside the component, which allows you to define a style in a very simple way while remaining elegant.

Let's see an example:

FILE: `button.tsx`

```
export const Button = component$(() => {
  useStylesScoped$(`
    .button {
      color: black;
    }
  `);
  return <button class="button">My Button</button>;
});
```

You can see that directly inside the component, `useStylesScoped$` is used, and the style has been defined. This approach, although very fast and immediate, is less suitable for structured applications because, in my opinion, there is a risk of losing sight of the general context, and the IDE does not offer support with IntelliSense.

Tailwind CSS

Tailwind CSS is a CSS framework that stands out for its use of utility classes to implement styles, as opposed to the traditional CSS approach.

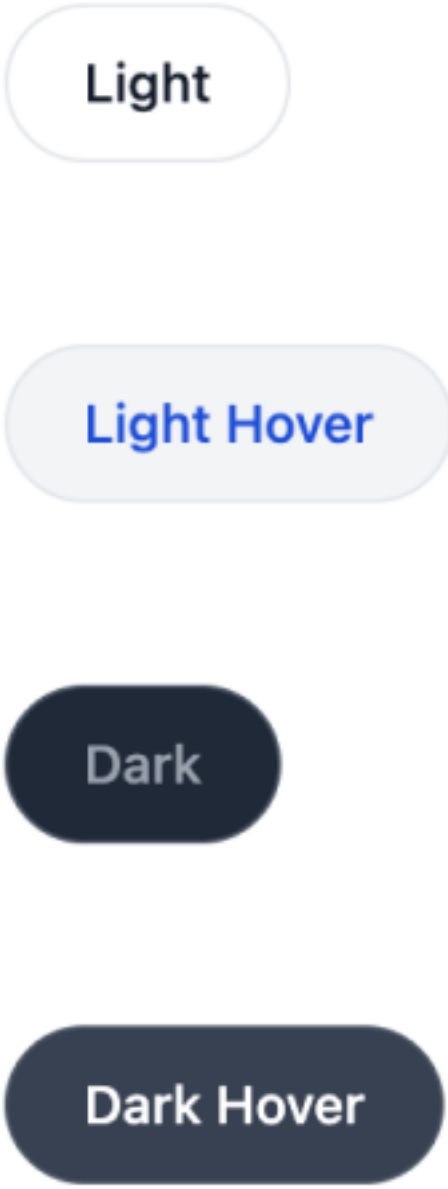
Let's look at an example:

FILE: `button.tsx`

```
export const Button = component$(() => {
  return (
    <button class="bg-blue-500 text-white">
      My Button
    </button>
  );
});
```

Here, you can use Tailwind CSS utility classes to define a blue background and white text color for your component. Instead of creating a class that combines these two properties, you can use two specific classes from Tailwind CSS. This approach allows for faster development but can result in verbose classes when creating complex styles.

Let's take a look at this example:



buttons

Here, you have a button available in two themes: `Light` and `Dark`. You also have hover variants for both themes. If you analyze the Tailwind classes used, you can see the following:

```
<button
  type="button"
  class="rounded-full border border-gray-200 bg-white px-5 py-2.5 text-sm font-mediu\
m text-gray-900 hover:bg-gray-100 hover:text-blue-700 focus:z-10 focus:outline-none \
focus:ring-4 focus:ring-gray-200 dark:border-gray-600 dark:bg-gray-800 dark:text-gra\
y-400 dark:hover:bg-gray-700 dark:hover:text-white dark:focus:ring-gray-700"
>
  [...]
</button>
```

As mentioned before, Tailwind can be verbose, and your button has many classes. Some CSS purists dislike Tailwind because it requires more classes compared to traditional CSS. However, this approach has several advantages:

- **Self-explanatory classes**: By reading the classes used, you can quickly understand the type of style being applied. Once you learn the meaning of the classes, it becomes automatic.
- **Copy/paste approach**: Within the HTML, you have everything you need to reuse parts or components of your applications. Simply copy and paste, and you're good to go. This is convenient when finding snippets on the web, allowing you to immediately incorporate them into your project and modify them as needed.
- **Single file component**: With Tailwind, you don't need to jump between files to see what a specific class looks like. You can understand the definition of classes without leaving the file you're working on. After using Tailwind for a few days, you quickly learn that `p-10` indicates `padding: 2.5rem; /* 40px */`, and the same goes for other classes. There's no need to refer to other files, making everything clear and concise.
- **Highly configurable**: Through the Tailwind configuration file, you can customize the framework's behavior to your liking. These changes will be reflected in the final stylesheet. Additionally, Tailwind allows the addition of custom plugins, providing endless possibilities.
- **PurgeCSS by default**: When using traditional CSS, you often create multiple classes for each HTML element. Although you don't reach the same number of classes as with Tailwind, there is a downside. When you need to remove a part of the HTML, you also have to remove the associated classes. This can be challenging, especially if the class is used by multiple elements. Unused classes tend to accumulate over time, reducing the efficiency of the application. Moreover, when applications are handed over from one developer to another, the fear of breaking something in production often prevents the deletion of unused classes. With Tailwind, unused classes can be eliminated during the build phase. Tailwind offers hundreds of utility classes, but only the used ones are included in the final bundle of your application. This process happens by default, but it is possible to achieve the same result using a traditional approach and a custom build process.

It's important to note that when purging CSS, the code of your project is statically analyzed. Therefore, if you use dynamic class names, you may encounter issues. All class names must be explicitly and completely written in your code.

 Let's see an example of incorrect use:

```
export default component$(() => {
  const color = "red";
  return (
    <div class={`bg-${color}-500`}>this won't work</div>
  );
});
```

This example doesn't work because the `bg-red-500` class cannot be statically recomposed. The interpolation is evaluated at runtime. To ensure the system works correctly, you should use the full name of the class.

 Let's see an example of correct use:

```
export default component$(() => {
  const color = "bg-red-500";
  return <div class={`${color}`}>this works</div>;
});
```

To add Tailwind CSS to Qwik, simply use the `pnpm qwik add tailwind` command. The Qwik CLI will add and install the necessary dependencies and create the required configuration files. In the upcoming chapters, you can use Tailwind in the application you create together. Now, let's explore some useful tips based on your years of experience.

Tailwind Tips

There are some tricks you can use to work effectively with Tailwind. While they are not strict rules, these tips can simplify and enhance your development experience.

Firstly, there are different ways to define classes within your Qwik component. You can use a string, an array of strings, or an object that allows for conditional insertion.

```
// string
class="class1 class2 class3"

// array of strings
class={['class1', 'class2', 'class3']}

// object
class={{
  "class1": myCondition1,
  "class2": myCondition2,
  "class3": myCondition3,
}}
```

One of the main disadvantages of Tailwind is the lack of readability when using multiple classes together. It can be challenging for your brain to process all the information when classes are written in this manner. Fortunately, Qwik provides a second option that can help break down the multitude of classes you need to use. Let's take the previous button example and try dividing the classes by defining your class as an array.

```
class={[
  "rounded-full border border-gray-200 bg-white px-5 py-2.5",
  "text-sm font-medium text-gray-900",
  "hover:bg-gray-100 hover:text-blue-700",
  "focus:z-10 focus:outline-none focus:ring-4 focus:ring-gray-200",
  "dark:border-gray-600 dark:bg-gray-800 dark:text-gray-400",
  "dark:hover:bg-gray-700 dark:hover:text-white dark:focus:ring-gray-700",
]}
```

By dividing the classes into blocks, you can improve readability without reducing the number of classes. Additionally, you can group the classes based on their purpose, according to your preferences. In the example, you start with border-related classes, followed by text-related classes, and then hover and focus classes. Finally, you group the classes applied when the theme is dark.

> Nowadays, almost everyone supports the dark and light theme, and for this reason, with Tailwind, we can use the `dark:` syntax to override the style for a given property in a very simple way.

This trick allows you to mitigate the readability issue associated with Tailwind. It's a simple way to enhance readability. If there are only 4 or 5 classes, using the array syntax may not be necessary, and you can use the simple string syntax instead.

You can also benefit from using the VSCode extension Tailwind CSS IntelliSense[36]. This extension improves the developer experience by providing advanced features such as autocomplete, syntax highlighting, and linting. It helps you learn Tailwind and prevents errors. It even warns you if you declare two classes that override each other, such as `text-sm text-lg`. In such cases, it suggests refactoring to eliminate one of the classes and optimize the style.

Another helpful plugin is prettier-plugin-tailwindcss[37]. This plugin can be added to Prettier and automatically reorders the classes used, ensuring they are consistently ordered based on logical concepts. For example, `dark:` classes are always placed at the end, while modifiers like `hover:` and `focus:` are grouped and sorted after plain utilities. This plugin provides a quick overview of related classes, helping you maintain control over your styles.

Additionally, the Tailwind Forms[38] plugin, developed by the official Tailwind CSS team, is very useful. It provides a style reset for form elements, making it easy to override their styles. This plugin is essential when creating custom forms.

Moreover, you can create your own custom CSS classes if you frequently use a set of classes. For example, if you often use `p-10 m-2 font-sm`, you can create your own class like this:

[36] https://marketplace.visualstudio.com/items?itemName=bradlc.vscode-tailwindcss
[37] https://www.npmjs.com/package/prettier-plugin-tailwindcss
[38] https://github.com/tailwindlabs/tailwindcss-forms

```
.p10-m2-fsm {
  /* Using @apply we can use any utility class name from Tailwind */
  @apply p-10 m-2 font-sm;
  /* more additional CSS property here */
}
```

In this way, you use `@apply` to define a CSS class while leveraging the utility classes provided by Tailwind. You can also define other CSS properties if needed. The name of the class is reminiscent of Tailwind's naming convention, allowing you to read your code without constantly switching between component and CSS files. The class name indicates that you are using padding, margin, and a specific font size.

Creating a Component Library

As you have seen, some tricks allow you to make the best use of Tailwind, but usually, when developing medium to large applications, certain elements are repeated throughout the application flow. For example, buttons which are present many times in the application because they allow user interaction. The quickest thing to do if you need to create a new button on your page is to copy and paste the button from another location where it is already present. But by doing so, button by button, technical debt will be created without knowing it. Let's imagine you need to renew the style of your application; you will have to examine all the components one by one to change the style. To avoid this problem, it is a good idea to create a reusable component that you can use at all points where it is required to centralize the logic and optimize any modification interventions.

Let's see an example:

FILE: `button.tsx`

```
export enum ButtonVariant {
    primary = 'primary',
    secondary = 'secondary',
}

export const variantClasses = {
    [ButtonVariant.primary]: 'text-black',
    [ButtonVariant.secondary]: 'text-red',
};

export type ButtonProps = QwikIntrinsicElements['button'] & {
    label: string;
    variant: `${ButtonVariant}`;
    onClick$: [...];
};

export const Button = component$<ButtonProps>(({ variant, label, onClick$ }) => {
```

```
    return (
        <button class={variantClasses[variant]} onClick$={onClick$}>
            {label}
        </button>
    );
});
```

You can use your custom Button like this:

```
[...]
<Button label='My Button' variant={'primary'} onClick$={[...]}/>
[...]
```

In this example, a button component with three properties was created:

- **label**: The label of your button.
- **variant**: You can pass `primary` or `secondary`, and TypeScript helps you avoid mistakes by indicating the possible values. If you pass `primary`, your button will have black text; if you pass `secondary`, the color will be red.
- **onClick$**: This property captures the click event of the button and allows you to apply your logic based on the context of use.

This example illustrates how components can be managed effectively to maintain control over your application. This component library serves as a language from which you can draw to create your application. Having a set of components to rely on reduces the time needed to create repetitive elements (e.g., buttons, inputs, modals, etc.). It establishes a design system that provides a consistent identity to your applications, ensures continuity in a digital ecosystem, and delivers a recognizable brand experience to users.

Consider Amazon as an example. The platform's colors and style are widely recognized because of its design system. Speaking the same language and using the same components and rules increases alignment within the company among developers and other departments such as UI/UX. One of the most famous design systems is Material Design, which Google developed. It has become a trend in web design and is the design system for Android. You can learn more about Material Design on the official website `https://m2.material.io/`.

Managing Data with Qwik

Now that the basics of making applications visually appealing have been covered, the focus can shift to understanding how to read data from external sources. This capability enhances the applications, making them more engaging and providing the user with the most enjoyable experience possible. Consider this example: during a login procedure, interaction with the application changes its state. The application "remembers" the user and allows access to personal information. Specifically, in Qwik, there are several ways to manage the state of the application, but the fundamental idea is to simplify the developer's work. In this chapter, different approaches to managing application states will be explored and reviewed with some examples.

Local State Management

useSignal

The low-level Qwik API that will be most beneficial in managing an app's application state is `useSignal`. It takes an initial value and returns a reactive signal. This reactive signal is an object with a single property `.value`. Changing this property will automatically update any component that depends on it.

Consider this example:

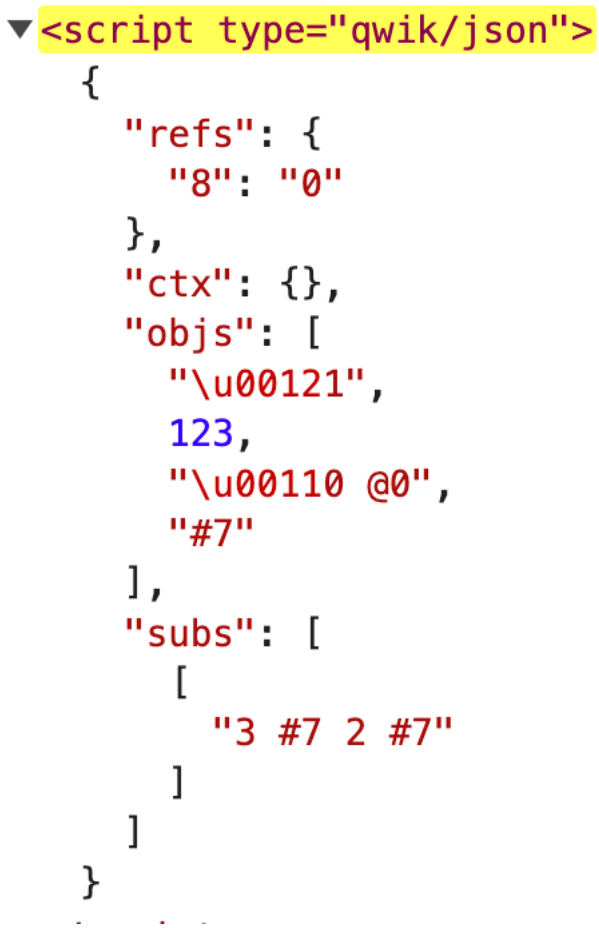
```
export default component$(() => {
  const countSig = useSignal(123);

  return (
    <div>
      Count: {countSig.value}
      <button onClick$={() => countSig.value++}>
        Increment
      </button>
    </div>
  );
});
```

> A naming convention has been defined to use the `Sig` suffix for signals variable names to make the component code more "Scannable".

In this example created via `useSignal`, a reactive variable with an initial value of 123 is established. As soon as it's modified (through the `onClick$` event), it triggers an update to the UI. Signals are highly efficient because the framework can track where that signal is read, allowing it to update things accordingly with a precise change without having to check and re-render the DOM of the components.

Qwik Optimizer

The signal mentioned earlier has an initial value of 123. In previous chapters, it was learned that during the build phase, the Qwik Optimizer packages the application to send it to the browser in anticipation of external events. With external events, the bundles necessary to make the application interactive will be executed. But if the application state needs to be changed and the initial value is 123, where can this information be obtained from? After all, on the first click, the value will have to increase by 1 and, therefore, go to 124. The answer is simple. In the static HTML that the server sends to the browser, the Qwik Optimizer also has the task of serializing the application state. It inserts it directly into the HTML page, and if the HTML sent by the browser is inspected, there is a `type="qwik/json"` script, which contains the initial value, 123.



qwik-json

This value will then be retrieved and injected into the JavaScript, which is necessary to make the counter interactive. By doing so, not only can the JavaScript be executed when necessary, but also the application state can be recovered and resumed from the point where the server had serialized the application.

useStore

In the previous example, something simple was done, but how can more complex states be managed? The `useStore` API can be used. It's very similar to `useSignal`, as it accepts an initial value and, by default, reactively tracks the properties that are nested in it, returning a reactive object. Essentially, it's about many signals that together make up a more complex object, ensuring its reactivity.

Consider this example:

```
export default component$(() => {
  const state = useStore({ count: 0 });

  return (
    <>
      <button onClick$={() => state.count++}>
        Increment
      </button>
      <p>Count: {state.count}</p>
    </>
  );
});
```

This example is very similar to the previous one. `onClick$` is still used, but this time the store is being updated.

> Note that for `useStore` to track all nested properties, many Proxy objects[39] need to be allocated. This can be a performance issue if you have many nested properties; in that case, you can use the `"deep: false"` option to track only top-level properties. When we create a store, by default, the deep option is false.

Global State Management with useContext

In the previous examples, how to declare a local state was discussed. However, with `useContext`, a global state in the application can be declared. The alternative to a global state would be to pass the information, component by component, within the props of each one, generating the so-called `props drilling`. To avoid this, it's good practice to create a global state.

Consider this example:

```
// 1
interface UsersStore {
  users: User[];
}
// 2
export const UsersContext =
  createContextId < UsersStore > "Users";

// 3
export const App = component$(() => {
  useContextProvider(
```

[39] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

```
    UsersContext,
    useStore <
      UsersStore >
      {
        users: [{ firstName: "John", lastName: "Doe" }],
      }
  );

  return <ChildComponent />;
});

// 4
export const ChildComponent = component$(() => {
  const todos = useContext(UsersContext);
  return (
    <ul>
      {todos.items.map((item) => (
        <li>{item}</li>
      ))}
    </ul>
  );
});
```

This example introduces several new concepts, but let's break them down step by step.

- **1**: The TypeScript type for the global state, which will be a list of users, is created.
- **2**: An id with `createContextId` is created, which will allow the global state to be uniquely referenced.
- **3**: In the App component with `useContextProvider`, the id (`UsersContext`), and the initial value, the global state can finally be created.
- **4**: In the child component, `useContext(UsersContext)` can be used to retrieve the list of users from the global state.

  NB. `useContext` will be available from the component where I declare the state with `useContextProvider` and in the descendant components in the tree. So, if I declare it in the root component of my application, it's global; otherwise, it will only belong in a specific part.

Qwik and Virtual DOM (vDOM)

Most of the operations needed to update the DOM are fine-grained, thanks to the reactive objects just discussed. However, there are structural changes that require more in-depth control of the DOM to update the UI. By structural modifications, operations such as dynamic components, components that create and destroy elements, and other operations that change the structure of the DOM are meant. The vDOM is used to calculate the updates to be made due to these structural operations.

> vDOM is a JavaScript representation of our application that resides in memory. It's used to calculate the changes in our UI; in fact, the current vDOM is compared with the previous one, and from the resulting changes, it will be possible to understand what the actual changes are to be applied in our real UI. With other frameworks, we have been used to associating the use of the vDOM with poor performance because the vDOM is an object that includes our entire application. Once again, Qwik amazes us because the vDOM decision is made on a per-component basis, only for components that have structural change and are changing their structure. If a component is structural (vDOM), but no change in structure is detected, then Qwik skips the component.

Exploring routeLoader$

`routeLoader$` is a unique API that allows data to be loaded on the server and injected into a component. For instance, if a list of users needs to be loaded from a database, this is a perfect operation for the server because secret keys and database connections will be needed. These secrets should not be exposed to the client.

> You can only declare a `routeLoader$` in the `index.tsx` and `layout.tsx` files inside the `src/routes` folder, and you need to export them.

Consider this example:

```
export const useUsers = routeLoader$(async () => {
  const users = await // our Database logic
  return users;
});

export default component$(() => {
  const usersSig = useUsers();
  return (
    <div>
      // here we can use usersSig.value to
      // render the user list
    </div>
  );
});
```

If the same list of users is needed in another component, this `routeLoader$` can simply be imported to centralize the logic of the application and maintain clean code. Furthermore, within the same file, multiple `routeLoader$` can be had to load different information. In this case, multiple operations will be performed in parallel to optimize timing.

In special cases, it is possible to use `useVisibleTask$` to perform operations as soon as the component becomes visible in the viewport. Imagine having a component that is outside the viewport because it's far down the page. The code written inside the `useVisibleTask$` hook is not executed at all. Qwik is purist in executing only the necessary JavaScript, and since the component is not visible, that code is considered unnecessary until the component appears on the page.

```
type User = {
    [...]
};

export default component$(() => {
    const usersSig = useSignal<User[]>();

    useVisibleTask$(async () => {
        const response = await fetch('https://..../users');
        usersSig.value = await response.json();
    });

    return <div>We have {usersSig.value?.length} users!</div>;
});
```

Here, by inserting the fetch call inside the `useVisibleTask$` hook, the list of users can be read and then viewed how many there are.

Using and executing code inside `useVisibleTask$` should be considered a last resort because Qwik provides many other methods to avoid eagerly executing JavaScript. There is an ESLint warning rule specifically to make the developer aware of evaluating other options.

Here's what the warning says:

```
Qwik tries very hard to defer client code execution for as long as possible.
This is done to give the end-user the best possible user experience.
Running code eagerly blocks the main thread, which prevents the user from interactin\
g until the task is finished.
"useVisibleTask$" is provided as an escape hatch.

When in doubt, instead of "useVisibleTask$()" use:
- useTask$ -> perform code execution in SSR mode.
- useOn() -> listen to events on the root element of the current component.
- useOnWindow() -> listen to events on the window object.
- useOnDocument() -> listen to events on the document object.

Sometimes it is the only way to achieve the result.
In that case, add:
// eslint-disable-next-line qwik/no-use-visible-task
to the line before "useVisibleTask$" to acknowledge you understand.
```

In addition to `useVisibleTask$`, `useTask$` is seen to perform server-side operations. Furthermore, there is the possibility of listening to events and then carrying out operations. The warning is very clear, but on the other hand, if the executed JavaScript is reduced to a minimum, many benefits will be reaped. So, this best practice should be remembered.

Using Forms with routeAction$

This API allows form submission operations to be performed. For instance, if writing to a database or sending an email is desired, it can be done securely on the server without worrying about exposing secrets to the client. Then, information can be received in response to updating the UI following the operation.

> You can only declare a `routeAction$` in the `index.tsx` and `layout.tsx` files inside the `src/routes` folder, and you need to export them.

Consider this example:

```
export const useAddUser = routeAction$(async (data, requestEvent) => {
  // the data prop is { firstName: "....", lastName: "..." }
  const user = await // our Database logic
  return {
    success: true,
    user,
  };
});

export default component$(() => {
  const action = useAddUser();

  return (
    <div>
      <Form action={action}>
        <input name="firstName" />
        <input name="lastName" />
        <button type="submit">Add user</button>
      </Form>
      {action.value?.success && (
        <p>User {action.value.user.id} added successfully</p>
      )}
    </div>
  );
});
```

In the example it can be seen that the `Form` component is used. It is a special component that wraps the native HTML form and, therefore, allows action even without JavaScript, has been imported. With JavaScript enabled (as in almost all cases), Qwik offers a very smooth experience without reloading, very similar to a single-page application. (see chapter 1) It should be noted that the `routeAction$` code will be executed only when the user submits the form or if the action is called programmatically.

`routeAction$` can be called programmatically like this:

```
[...]
<button
  onClick$={async () => {
    const { value } = await action.submit({ firstName: 'John', lastName: 'Doe' });
    console.log(value.user.id);
  }}
>
  Add user
</button>
[...]
```

If attention has been paid to the example, it will surely be noticed that among the parameters of `routeAction$`, there is `requestEvent`. This object gives access to the HTTP call that the framework executes behind the scenes, allowing communication between the client and server. Thanks to this object, access to the request and the response, and therefore to the HTTP parameters, cookies, and much more, will be had. Of course, the backend API can always be called via a client-side HTTP call. Simply insert the code inside `onClick$`. So when the button is pressed, the modification API will be called, and then the application will be modified based on the response from the server.

```
[...]
<button
  onClick$={async () => {
    const response = await fetch('https://..../addUser', [...])
        // more logic here
  }}
>
  Add user
</button>
[...]
```

globalAction$ vs routeAction$

`globalAction$` can be declared in the `src` folder. If the action is shared on multiple routes its use is recommended because it's globally available. On the other hand, `routeAction$` can only be declared in the `src/routes` folder and is accessible within the route it's declared, so its scope is delimited.

Summary

In this chapter, many points have been covered, starting with an analysis of how to maximize the application flow analysis. It's crucial for you to interview the end users of your solution because, as a developer, your

job is to solve problems. Some useful tricks have been shared to reduce uncertainty when implementing new projects or features.

Continuing in the chapter, a deep dive was taken into the various CSS writing methods available in Qwik. Specifically, the strengths and weaknesses of these methods were examined. After this comparison, the focus shifted to Tailwind, the tool you will use in the application you'll develop in the subsequent chapters.

Some tricks that are used daily to optimize the use of Tailwind have been shared. If a common thread were to be found among these tips, it would be their focus on creating a Tailwind-styled application that is readable and maintainable in the medium to long term.

It was discovered that integrating Tailwind with Qwik is straightforward, thanks to the Qwik CLI. Exploration was also done on how to create a library of components, the benefits of this approach, and why you should consider using this design method for your applications. The advantages are numerous. Having your style and well-defined graphic rules will make your applications unique and will help grow and consolidate your brand identity.

Next, you learned how to manage data within a Qwik application. While creating beautiful applications is important, they must also be functional. To make them interactive, you need to understand how to manage the application state.

The chapter began by examining how to manage a local state in your components using `useSignal` and `useStore`. You also saw how the Qwik optimizer serializes state data into your initial HTML. Then, you learned how to share a global state between multiple components with `useContext`. This method allows you to avoid passing properties back and forth between your components and provides an easy solution to a problem that is difficult to solve in other frameworks.

You delved into how Qwik manages the DOM update behind the scenes with signals and vDOM. At the end of the chapter, you learned how to read data on the server side via `routeLoader$` and on the client side via `useVisibleTask$` to render them in your application.

If you need to modify data and make calls, you can use `routeAction$` or `globalAction$`. Of course, you can always call your API for data modification via fetch.

In the next chapter, you will start building your e-commerce site with Qwik and the help of Supabase, a truly remarkable edge database.

Creating an e-commerce with Qwik and Supabase

Let's Architect Your App

Creating a new application is always a lot of fun because it's one of those moments where imagination can run wild and thoughts about what technologies can be used and how to implement the solutions can be explored. It's usually a great moment of discussion with the team and it is precisely in these moments that a lot of experience is gained. Junior developers have the opportunity to learn new techniques, but even more senior developers find themselves looking around and evaluating whether the technology they've always used is still the one to use. Design meetings are usually organized to discuss the requests together, fully understand the application domain, and try to eliminate any design doubts as much as possible. Each developer will pay particular attention to their area of expertise, so the designer will be more attentive to graphics and user experience while the developers will already be projected to think about how to save data and which algorithms to use to create the best application. As explained in the previous chapters, it is important to clearly understand the functional requirements of the application, what features need to be implemented, and what flow the users will follow to perform the operations. When these decisions are made, also keep in mind the deadline that the project imposes. Let's imagine having to create an application for a fair, the application will be used to allow participants to register for the event, receive a confirmation notification, and receive the ticket via email. Furthermore, there will also be the possibility of displaying a QR code that will allow participants to access the event. In these few lines, a whole series of requirements that are necessary otherwise the application would not make sense have been listed. However, it is implicitly saying that the deadline cannot be changed because if the application is not ready for the event then it can be said that the project has failed. If the job was commissioned to be done in a few weeks, it would be a challenging situation. In this very delicate situation, there is an alternative.

Trade-off Sliders

This exercise can be used whenever a new feature or entire application is to be implemented. This exercise is based on a very simple concept, these four elements are compared:

- **budget**: This is a very delicate topic but in practice, this metric wants to indicate how important it is to stay within the set budget or whether there is room for maneuver.
- **quality**: This element aims to indicate the quality of the application understood as the fact that it is tested correctly and is as free from logic bugs as possible, but also how graphically perfect and free of imperfections it is.
- **scope**: Here an aspect more regarding the features that are included in the application is defined. This tells how important it is that all the required features are present.

- **deadline**: As seen previously this metric is quite obvious. How important is it to deliver the work to the expected deadline? What happens if it is exceeded?

Meet together with all the people who will work on the project and the stakeholders to promote maximum alignment. Together, try to give priority to the elements seen previously, try to give them an order, from the most important to the least important. This exercise will lead to some very interesting discussions. Is it more important to arrive at the deadline on time or is it more important that all the features are present as required?

Taking the example from before: "The QR code is not displayed in the application, but only the email with the ticket, can the delivery of the application be moved?" This type of discussion is very constructive and allows the guiding rules that must guide decisions during the design and architecture of the application to be drawn up. For example, if quality is first and scope last, it is being said that it is necessary to be sure to have a well-tested application with an architecture that scales and is rock-solid. This is the basic rule that must guide the developments, the effort for each functionality must also include the time to write the tests and verify that everything works as required. This attitude will depend on the features present in the application or the fact that the rock-solid architecture may require some extra budget to ground.

It seems like a very simple exercise but in the end, it often happens that everything is important, in fact, managers often struggle to give order to these elements. Looking at each other and saying that a certain feature can be eliminated to reach the expected deadline puts everyone in the same boat and promotes alignment. If you want to try this exercise too, you can follow the instructions on the Atlassian[40].

Golden Hammer

It is clear that if the deadline cannot be changed, the best thing to do is to use tools where there is full knowledge and a feeling of productivity and confidence in being able to estimate and deliver the work with certainty. Using the technologies that are felt confident or strong about, without considering anything else, is usually considered an anti-pattern, however, a non-modifiable deadline is a sufficient restriction to adopt this approach. Many times, however, for no apparent reason, this phrase has been heard: "Let's do what we have always done". This way of thinking is also called: "Golden Hammer" and is a very dangerous anti-pattern, now it will be explained why. This way of doing things can refer to a tool, language, database, or platform that developers feel comfortable and productive with, so they are tempted to use it for whatever problem arises, even if thinking about it wouldn't be the right choice.

> As the saying goes: "If all you have is a hammer, everything will look like a nail."

Perhaps there was satisfaction with a low-code platform in the past and since there is experience with it, it is wanted to be used for the new application. However, it was not taken into account that the business requires a high level of customization, and therefore the choice did not benefit either the result or the end customer.

How can this anti-pattern be avoided?

Evidence and numbers are the best weapon to prove, for example, that in a specific domain Tool A is better than Tool B and it is necessary to be able to convince the people making the decisions why Tool B is a better choice. Experience and use cases deserve a separate chapter because a lot depends on the people that need

[40] https://www.atlassian.com/team-playbook/plays/trade-offs

to be convinced. If they are not technical they will hardly take into consideration metrics based on technical aspects. Unfortunately, the right chord based on the people being dealt with needs to be struck, because perhaps underlining that one solution would cost more rather than another has more appeal. If a serverless approach on the Cloud was to be used, the decision-makers could be provided with the numbers and the economic savings that could be achieved if it were used compared to a more classic server architecture. One way to obtain this useful data is to use the "technological spike methodology".

Technological Spike

A technological spike, in the context of software development, is a minimal initial implementation of the architecture. This procedure allows you to validate and reduce the uncertainty of a system's various functions by evaluating different options. These assessments usually occur within a well-timed period, which can range from a few hours to several days of investigation. In an agile context, it typically never exceeds a sprint. However, if the time is insufficient, you can allocate more time in the next iteration.

Imagine needing to choose the frontend framework for your architecture. First, you need to compile a list of desired features:

- Ready for production
- Integrates well with your services (authentication, infrastructure)
- Intuitive to use
- Has supporting libraries
- Strong community

During your sprint, take time to analyze the various frameworks in detail. This activity isn't necessarily limited to coding but can also include a technical analysis with the recovery of data and metrics to compare the various tools. Once you have the data, share your results for each framework with the team. It's advisable to share these results while also making your idea clear to stimulate a productive discussion. Allocate some predefined time to skim the solutions and narrow down to a few options, for example, two or three frameworks out of the five or six analyzed. The entire team must be aligned and involved in this decision to prevent the person presenting the data from feeling pressured by such an important decision.

With this data, create a basic application with the frameworks that the team has screened. It's crucial that this activity isn't performed solely by the person who extracted the metrics; rather, the entire team should participate. This is a good opportunity to exchange knowledge within the team and help make the most shared decision possible.

The result of this step will be a winner among the compared frameworks, and you can then validate your choice with your managers. As previously mentioned, managers and C-level executives have different parameters compared to the purely technical aspect. Therefore, it would be beneficial to emphasize that a framework allows you to develop faster because it's easy to use, well-integrated, and feature-rich.

If you were to present Qwik to one of your managers, avoid delving into the fact that it sends zero JavaScript by default and uses a service worker. Instead, emphasize that with Qwik, your SEO will be 100%, reducing the need for an external agency to optimize these metrics. This will benefit your budget due to significant cost savings. You could also highlight that the end user will be more satisfied with the user experience and

spend more time in your application, giving you an edge over your competitors. Another aspect to consider is the development time of various functions. Thanks to a fabulous DX, you will be more productive with Qwik and save days of work. Given the time, you could develop the same functionality with the frameworks passed from the first skimming done by the team. This would allow you to present data like: "With Qwik, this simple functionality took me two hours less than this other framework. With Qwik, we will save a lot of money."

This verification method also helps reduce uncertainty and allows you to start developments more confidently or better estimate future developments. If you were to evaluate a new functionality that you have no idea how to handle, this type of spike approach is an excellent way to clarify your ideas.

Non-functional Requirements

You have now discussed the process that can help you decide how to design your application. The example was based on frontend frameworks, but the same approach can be used for choosing the database or any other decision you need to make. By performing the technological spike, there's a risk of focusing only on functional aspects, but you must also consider the non-functional aspects that your domain requires you to guarantee.

Non-Functional Requirements (NFR) are the key ingredient for making informed decisions. Here are some examples:

- **Availability**: What are the operational requirements? Does it need to run 24/7?
- **Scalability**: As needs grow, can the system handle them? For physical installations, this includes spare hardware or space to install it in the future.
- **Supportability**: Is support provided internally, or is remote accessibility required for external resources?
- **Security**: What are the security requirements, both for the physical installation and from an IT perspective?
- **Readability**: The ease with which you can understand the flow and operation of your codebase.

Imagine having this non-functional requirement to meet: "Any team member must be able to read the source code to be able to develop new features".

Readability is crucial because it prevents development blockages. If you don't understand a part of the code, an algorithm, or a procedure, you are effectively tying yourself to the person who wrote it. If this person is no longer part of the company, you are stuck until you understand how it works. Moreover, code that is not very readable also slows down developments and facilitates the introduction of bugs because there is a higher probability of introducing side effects precisely because you do not know the entire codebase well, and perhaps you have underestimated specific cases.

So, if you have readability as a non-functional requirement, keep in mind that you need to use a language that favors this activity. By choosing TypeScript, you have greater readability than other languages, and you can also make it a rule to comment on the code to clarify why certain choices have been made.

While comments in the code can become obsolete quickly, and they are more "code" to maintain and align, they make sense if a certain architectural choice requires an explanation. For example, suppose you perform a workaround for a bug that doesn't depend on you, but on a library you use. In that case, it's good to

indicate why the fix was introduced and perhaps also attach an article or issue that provides evidence of the problem. This will document the code because anyone looking at the fix will understand why it was done. Another good practice if you want to encourage knowledge exchange within the team is reviewing PRs. When implementing a feature, conduct a collaborative review of the changes. This serves two purposes: exchanging information and preventing potential bugs through code review.

So, by keeping in mind the requirements of your application, whether functional or non-functional, you can organize, choose, and implement your work in the best possible way.

Technical Requirements

To actively follow this chapter and create the application, ensure that both Node.js[41] and `npm` are installed on your local computer. Alternatives such as `yarn` and `pnpm` can also be used. `pnpm` might be a preferred choice due to its efficiency and several advantages. In fact, with `pnpm`, when a package is installed, it is stored in a global folder on your computer. Therefore, for each version of a module, there is always only one copy stored on the disk because it creates symbolic links and does not re-download the packages. When using `npm` or `yarn`, for example, if there are 10 projects that use "package A", there will be 10 copies of "package A" on the disk. So, `pnpm` saves a lot of disk space and installation time, let's use it.

> The code snippets included in the chapter are deliberately minimal to focus attention on the fundamental steps, while in the GitHub repository, you can find the complete version.

Supabase for Your Backend

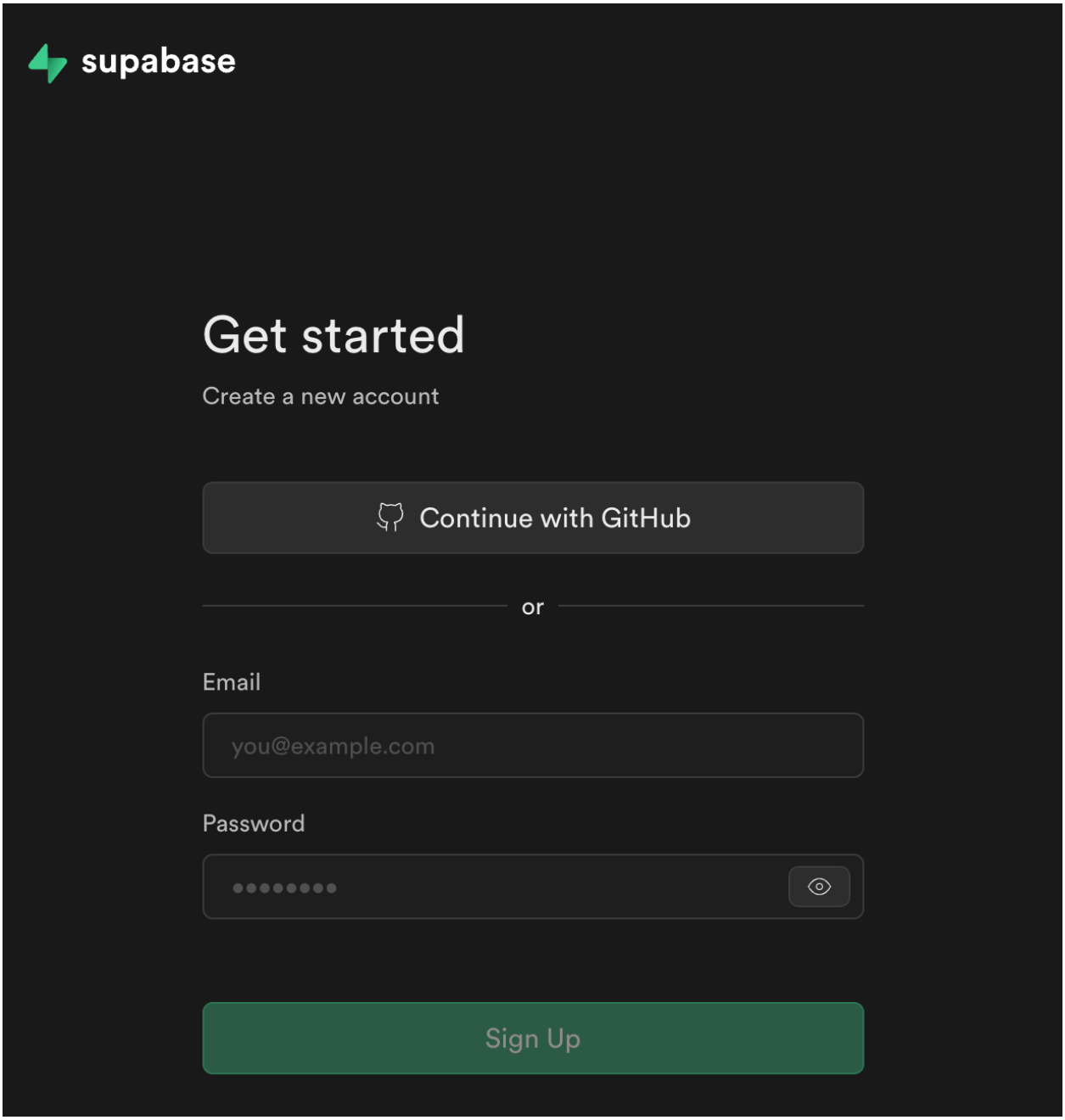
After some evaluations, it's clear that the most interesting database to use for your application is Supabase. Below are the advantages:

- **Highly integrated with Qwik**: In fact, it is included in the official integrations of the framework, so compatibility is guaranteed by the team that oversees the project, as well as by the maintainers and the large community present.
- **Supabase configuration**: It's very simple and fast. In fact, in just a few minutes, you can obtain a working solution. All this is thanks to the UI interface that facilitates operations.
- **Authentication and authorization**: These operations are highly integrated with the ability to securely manage user accounts, roles, and permissions, ensuring safety and security for your data.
- **Postgres support**: This allows you to manage complex data relationships and queries effortlessly. This is a significant advantage over other platforms that often offer limited database capabilities.
- **Supabase SDK**: Managing the connection and all database operations is very simple thanks to the provided SDK.
- **Real-time status**: This supports listening to Postgres database changes, monitoring, and synchronizing shared state between clients or sending temporary messages from client to client with low latency. These features are not obvious for this kind of service, and knowing that you can use them effortlessly is very nice.

[41] https://nodejs.org/en

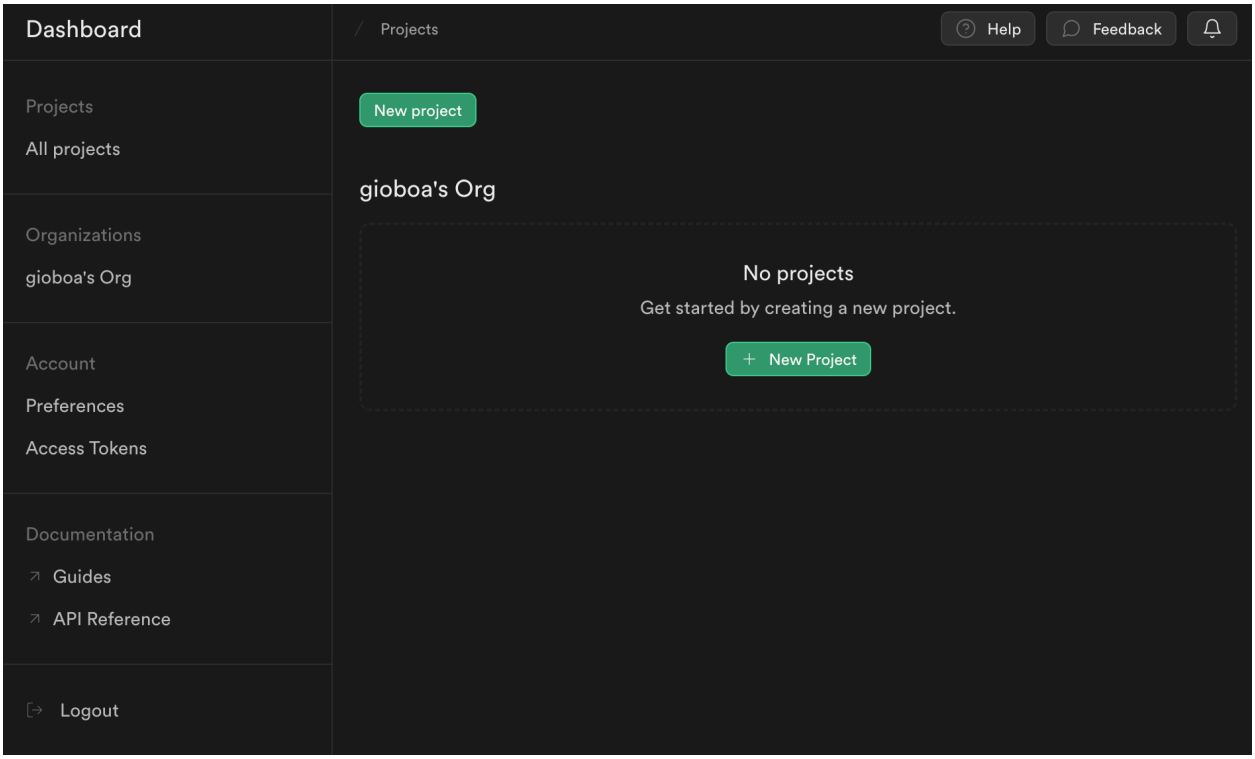
- **Environments**: It is possible to have multiple environments for your use. You can have the production environment, the test environment, and you can create all the environments you need.
- **Migrations**: Having multiple environments and evolving your database feature after feature, it is possible to automatically migrate the database schema. This guarantees consistency and saves you the burden of managing everything manually, avoiding human errors.

To be able to use Supabase's features, you must create an account, and you can do this directly from their site. You can also use your GitHub account to create the user very quickly.

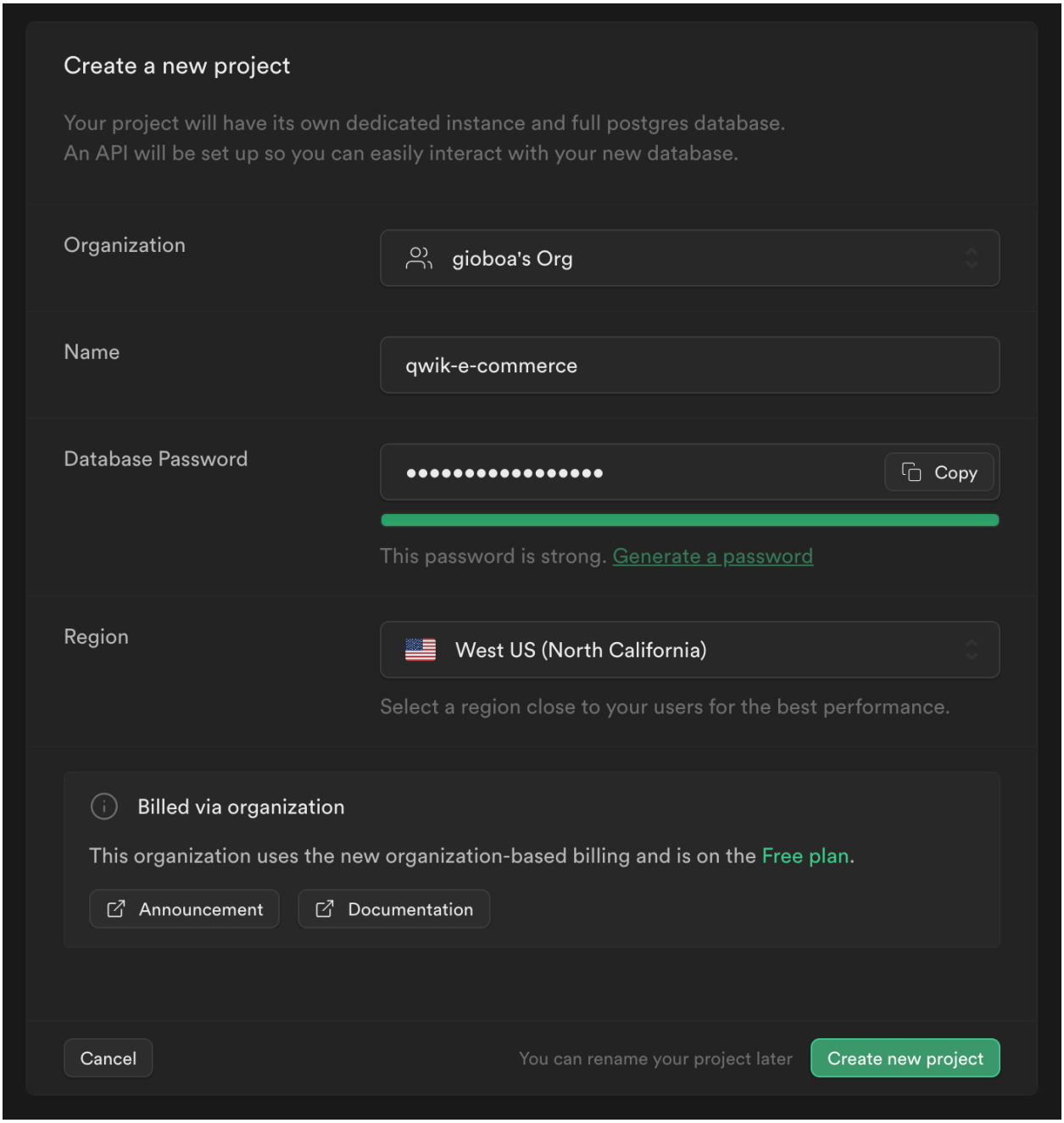


SignUp

Once the user has been created, it will be sufficient to create your project via the dashboard. Call it `qwik-e-commerce`.


Dashboard
Projects
Help
Feedback
Projects
All projects
Organizations
gioboa's Org
Account
Preferences
Access Tokens
Documentation
Guides
API Reference
Logout
New project
gioboa's Org
No projects
Get started by creating a new project.
New Project

NewProject



NewProject2

In addition to the name, you are asked to set a password for your database and the region where to create your database for maximum performance. It is also possible to consult the service's prices, but it doesn't make much sense to list the current prices because this information could change over time. What is important, however, is that you are on the free plan.

Let's create a `test` table and add a new column `test_column`.

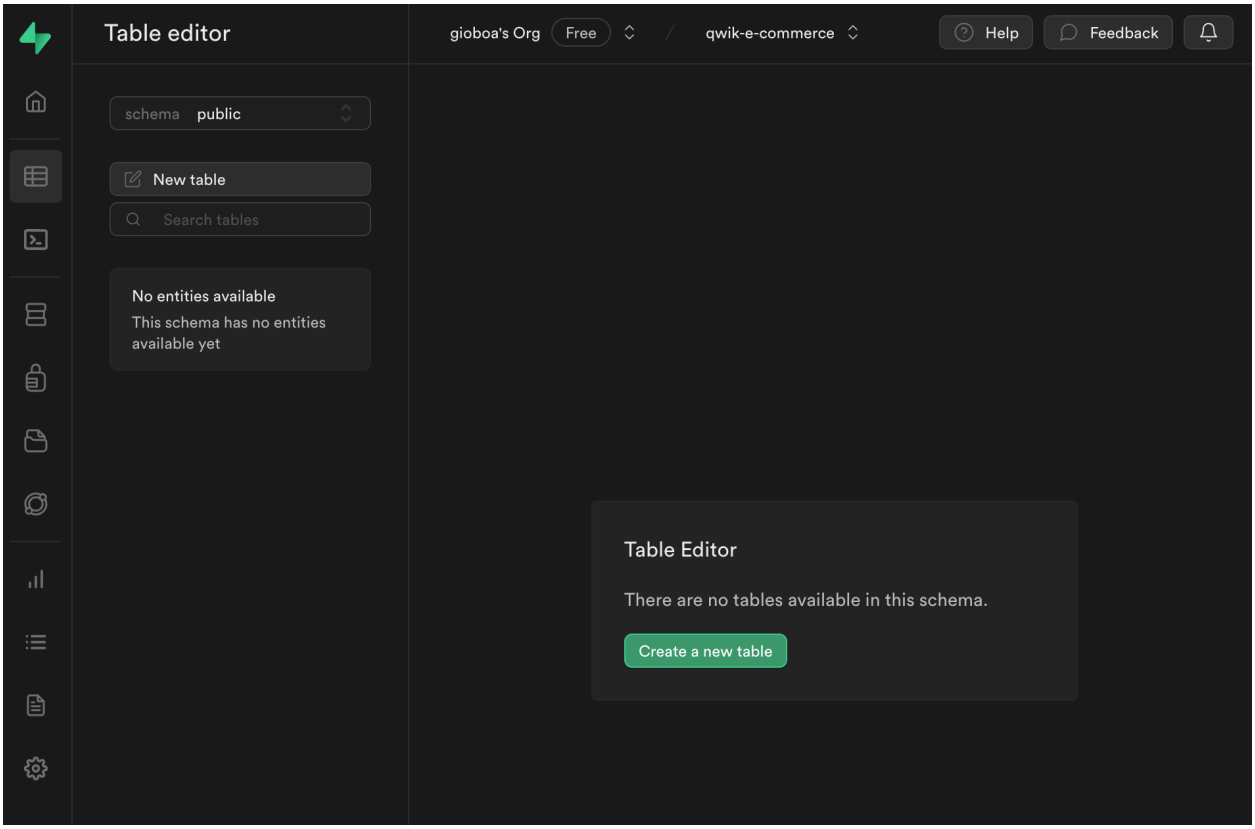
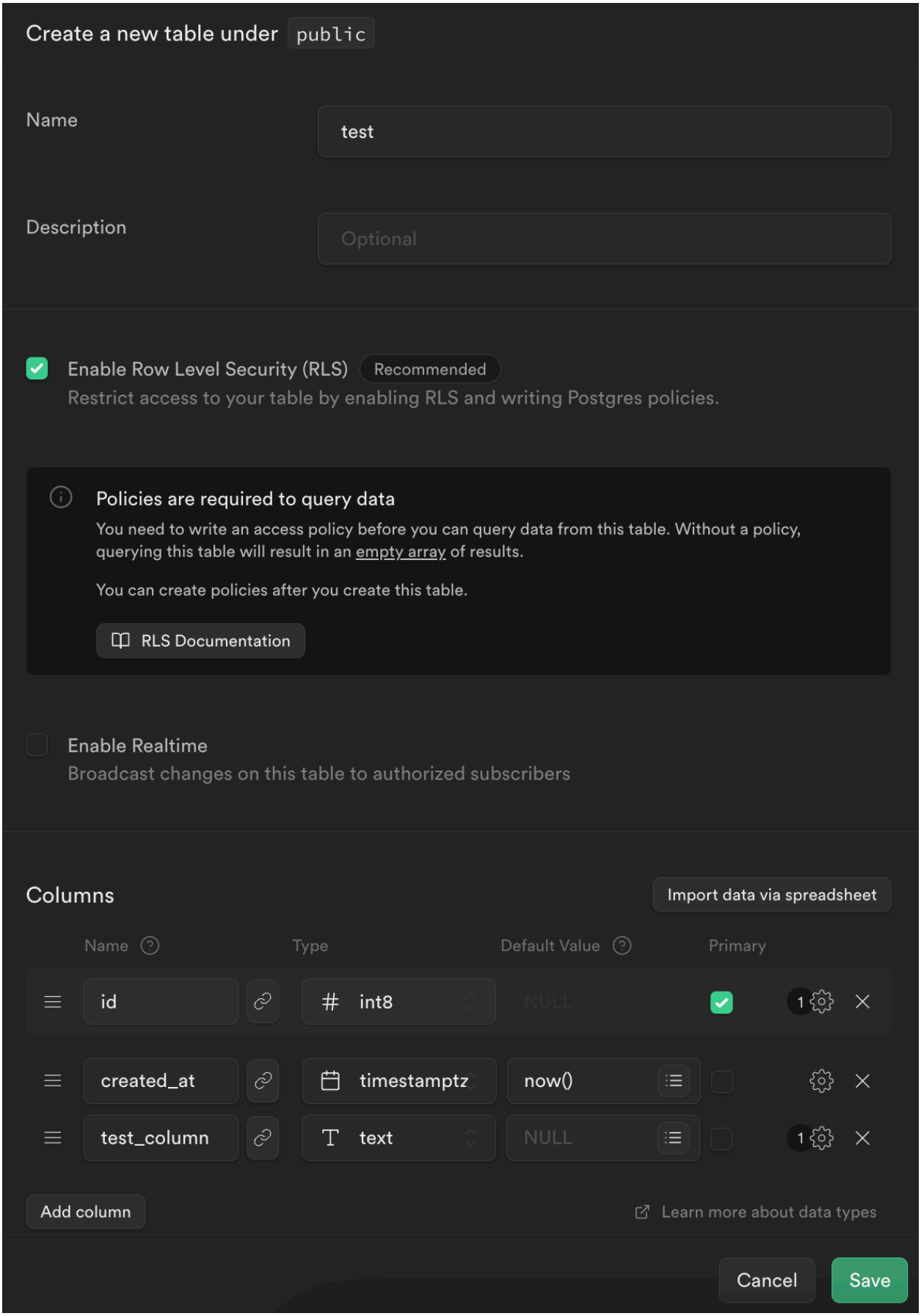


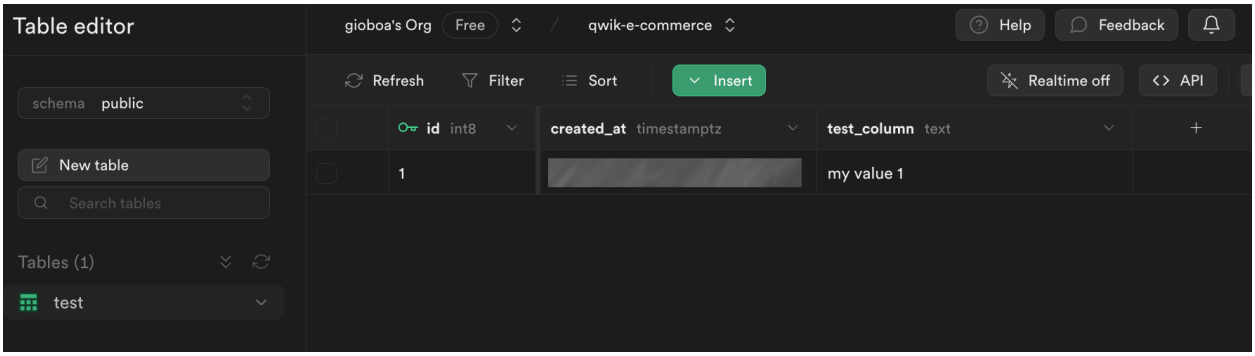
Table editor
gioboa's Org
Free
qwik-e-commerce
Help
Feedback
schema public
New table
Search tables
No entities available
This schema has no entities available yet
Table Editor
There are no tables available in this schema.
Create a new table

NewTable


Create a new table under public
Name
test
Description
Optional
Enable Row Level Security (RLS)
Recommended
Restrict access to your table by enabling RLS and writing Postgres policies.
Policies are required to query data
You need to write an access policy before you can query data from this table. Without a policy, querying this table will result in an empty array of results.
You can create policies after you create this table.
RLS Documentation
Enable Realtime
Broadcast changes on this table to authorized subscribers
Columns
Import data via spreadsheet
Name
Type
Default Value
Primary
id
int8
NULL
created_at
timestamptz
now()
test_column
text
NULL
Add column
Learn more about data types
Cancel
Save

TestTable

After creation, let's create a first example record.



TableEditor

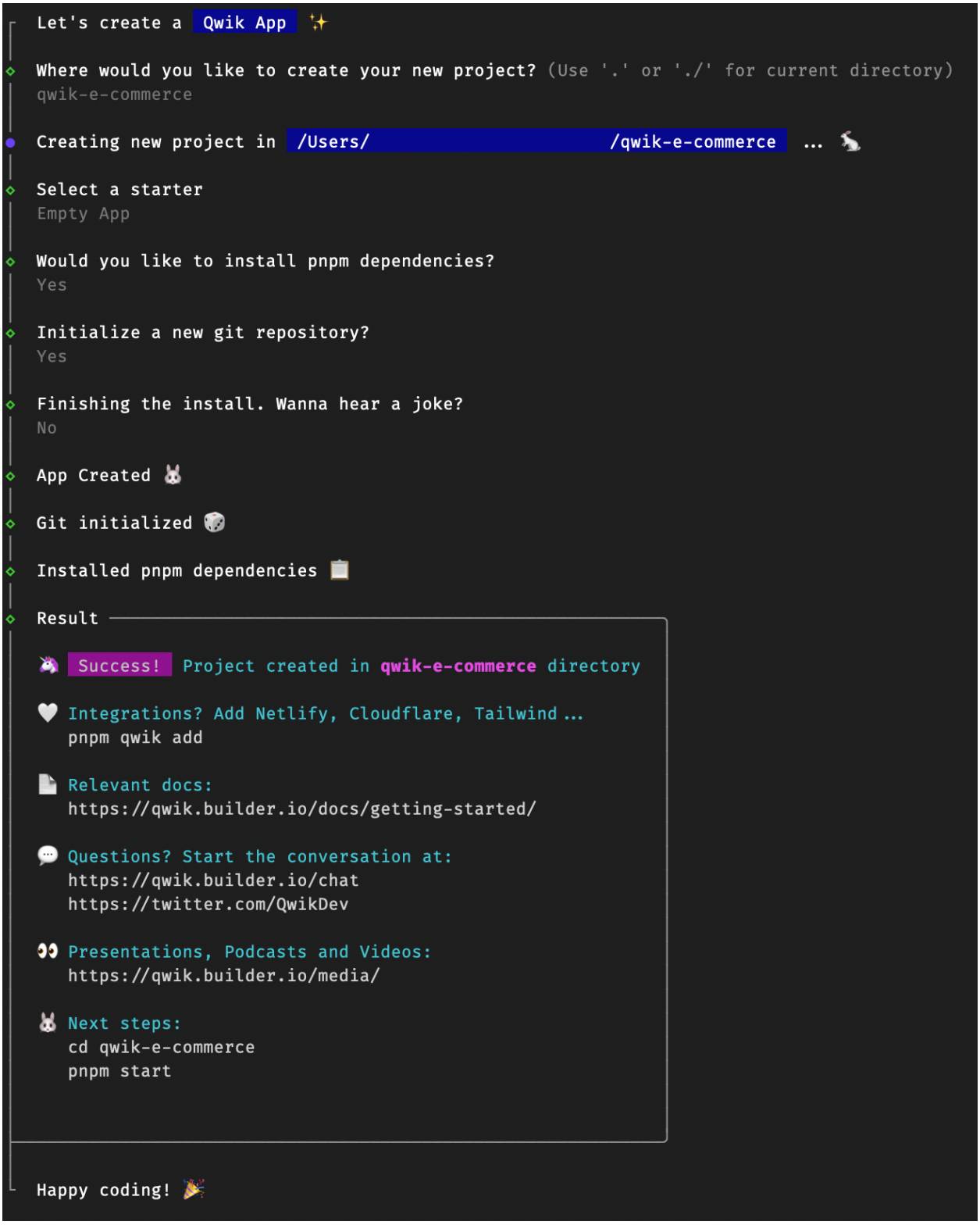
Render Supabase Data

To get started with your app, you can take advantage of the Qwik CLI as seen in the previous chapters. So go to your favorite directory for your projects and type in the terminal:

```
pnpm create qwik@latest
```

> @latest will use the latest version of Qwik

You will be guided with a series of questions to create your project. Let's call the project "qwik-e-commerce" and choose "Empty App" as a starter.


Let's create a Qwik App ✨
Where would you like to create your new project? (Use '.' or './' for current directory)
qwik-e-commerce
Creating new project in /Users/ /qwik-e-commerce ... 🐇
Select a starter
Empty App
Would you like to install pnpm dependencies?
Yes
Initialize a new git repository?
Yes
Finishing the install. Wanna hear a joke?
No
App Created 🐰
Git initialized 🎲
Installed pnpm dependencies 📋
Result
Success! Project created in qwik-e-commerce directory
Integrations? Add Netlify, Cloudflare, Tailwind ...
pnpm qwik add
Relevant docs:
https://qwik.builder.io/docs/getting-started/
Questions? Start the conversation at:
https://qwik.builder.io/chat
https://twitter.com/QwikDev
Presentations, Podcasts and Videos:
https://qwik.builder.io/media/
Next steps:
cd qwik-e-commerce
pnpm start
Happy coding! 🎉

Cli

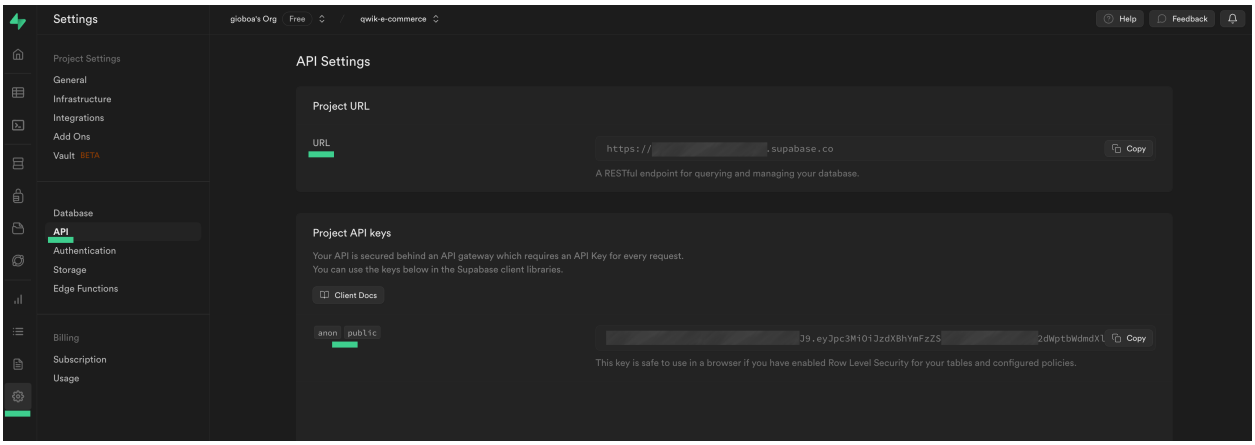
Once the project has been created, you can open the project directory with your favorite editor. The Supabase JS library plays nicely with the server-side APIs from Qwik such as `routeLoader$`, `routeAction$`, or `server$`. You need to install Supabase in the project:

```
pnpm install @supabase/supabase-js supabase-auth-helpers-qwik
```

For this implementation, you will need two environment variables that allow you to communicate with Supabase. These can be copied directly from the Supabase Cloud dashboard, Settings > API area. (see image)

FILE: `.env`

```
VITE_SUPABASE_URL=...
VITE_SUPABASE_ANON_KEY=...
```



EnvVariables

> Note that you need to manage the Supabase Row Level Security (RLS) to allow your application to read and write to the database. Since things may change over time I prefer to refer you to the Supabase documentation[42] where it is explained in depth how to configure these policies.

Let's edit the `src/routes/index.tsx` file as follows.

FILE: `src/routes/index.tsx`

[42] https://supabase.com/docs/guides/auth/row-level-security

```
import { createServerClient } from "supabase-auth-helpers-qwik";

type TestRow = {
  id: number;
  created_at: string;
  test_column: string;
};

export const useTestTable = routeLoader$(
  async (requestEv) => {
    const supabaseClient = createServerClient(
      requestEv.env.get("VITE_SUPABASE_URL")!,
      requestEv.env.get("VITE_SUPABASE_ANON_KEY")!,
      requestEv
    );

    const { data } = await supabaseClient
      .from("test")
      .select("*");

    return data as TestRow[];
  }
);

export default component$(() => {
  const testTableRows = useTestTable();
  return (
    <>
      <h1>Hi from Supabase </h1>
      {testTableRows.value.map((row) => (
        <div key={row.id}>{row.test_column}</div>
      ))}
    </>
  );
});
```

First, you define the type for your records in the test table. Then you use the Qwik server API to read the data from Supabase securely because `routeLoader$` code will never be present in the browser, but is only executed on the server side. Finally, you take the data from the test table and show them on the page. Below, you see the result.

SupabaseResult

Exploring the Authentication Process

Authentication is a fundamental part of every highly dynamic and personalized web application. Authentication refers to the process that identifies a specific user, allowing you to read, write, update, or delete any protected content, depending on your level of authorization. A typical example is a social network, where each user can create personal posts only after authentication, and fortunately, it's not possible to create content on behalf of third parties. This allows for personalizing the experience of using the application for each user and enables you to save configurations and link them to your user profile. Imagine having to save your preference for a dark or light theme; managing authentication makes this possible.

The Authentication Process is Not as Simple as It Seems

The first thing to ask yourself is whether it makes sense to use an external service for authentication or if you should have full control over the process and manage the entire flow with your service. The custom approach has some complexities that are worth considering. First, you have to think about how to safely save your data to prevent external attacks from reading this valuable information. Then, you have to consider the entire flow in its entirety. It's not just about a simple login; there are other features, such as password recovery and many more.

Imagine that initially, your application allows you to log in with an email and password, but subsequently, you're asked to allow authentication via Google. With a custom solution, you would have to invest some effort to cover this functionality. What if you were then asked to log in to Facebook, too? This would be additional unplanned work at the start of the estimates. The possibility of logging in with different providers is the basis of external services, not to mention the sending of emails and notifications that are necessary for a flow of this type. Do you want to reinvent the wheel and create your own service for sending and translating

messages? These problems make it clear why the use of an external service now seems like a solution to at least be carefully considered. In fact, for your e-commerce with Qwik, Supabase Auth will be used.

JWT Token

In the past, web-side authentication was based on the concept of server-side sessions. The process began by filling out a form to log in; the user entered the username and password and sent them to the server. The credentials were checked on the server, and if they were correct, a session ID was created and saved in memory, which was then used to exchange requests between the client and server. This approach, although used for years, is vulnerable to external attacks.

Furthermore, modern applications are designed to run on the cloud to scale horizontally, and this session approach saved on a single instance does not lend itself to these scalability needs. This old approach has been replaced by a more modern one, which is based on JWT tokens. In this new mode, the user sends his username and password to the server. After verifying their validity, the server creates a JWT token that can be used in subsequent calls. The client saves this token in the cookies because it is a safe place after saving it on the browser side, and in future requests, it will be added to the call headers.

The server now only has the task of reading the JWT token from the HTTP call headers, and therefore, there is no need for any search in the local memory of the server instance. It is an atomic operation that, as mentioned, lends itself to a distributed system in the Cloud.

You can think of this process as when you check in at a hotel. You present yourself at the reception, providing your data (your credentials), and once everything is verified, you're given the room key. This key is effectively your JWT token, allowing you to access reserved areas such as the gym, the spa, and finally, your room, which can be compared to the reserved area of your application. It should be noted that with the room key, you cannot access the rooms of other hotel guests, and once your stay is over, it will no longer be valid. Just like the room key, the JWT token also has an expiration mechanism, and once it expires, it will no longer allow you to be authenticated. You will need to log in again with your credentials. It may have happened to you that you logged in to your favorite social media and then never logged in again for months, if not years. This can happen because, through a refresh token, which is exchanged with the server, you have the possibility on the client side to extend the expiration of the token itself and allow you to have very long sessions without logging in again with the username and password.

But let's look in detail at how a JWT token is constructed.

```
// JWT Encoded

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9\
lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

// JWT Decoded

// Header: algorithm & token type

{
```

```
  "alg": "HS256",
  "typ": "JWT"
}

// Payload

{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}

// Verify signature

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

You can see that the encoded JWT token is constructed in three parts separated by `.` and looks like the following `xxxxx.yyyyy.zzzzz`.

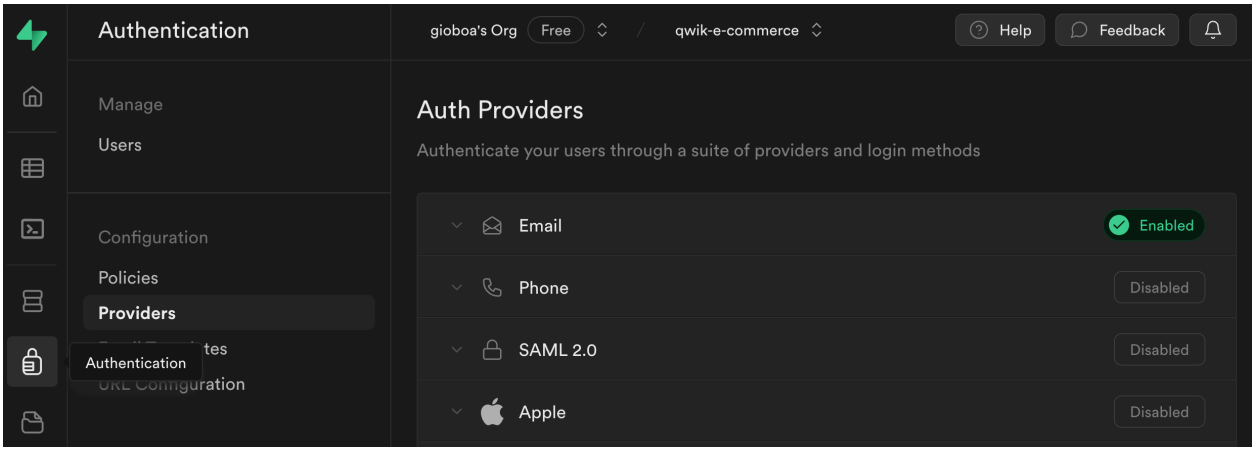
Once these three parts have been decoded, you can read their contents. They contain very precise data and are respectively:

- **header**: Here, you can see two properties: the token type, which is JWT, and the signature algorithm used (HMAC SHA256 or RSA).
- **payload**: The second part of the token is the payload and generally contains the user and some information that you want to pass; for example, you can use the token to save the user role or its permissions. It also contains the expiration time and other data relating to the token issuer. The information in the payload is readable by anyone, so it's a good idea not to put any secret data inside unless it's encrypted.
- **signature**: The third part of the signature is created with the encoded header, the encoded payload, a secret, and the algorithm specified in the header and signed. This part is very important because it is used to check if the message has been modified fraudulently. If this part does not match, then there has been tampering with the package. If a private key was used for the signature, then it is also possible to check whether the sender of the JWT is correct or not.

This JWT token is effectively a message that is passed back and forth in HTTP calls and, by its nature, is very simple to use. Nowadays, all authentication methods are based on this system, and even the system that you are going to use is based on this procedure and exchange of information. In fact, Supabase, being a modern system based on the Cloud, is certainly no different.

Supabase Auth is Really Good

Once the Supabase account and the `qwik-e-commerce` project are created, you can move on to the authentication part. Once you enter your project, check that in the `Authentication/Providers` section the email provider is enabled.



auth_providers

On the application side, you will create two routes, `sign-in` and `sign-up`, inside the `routes` folder. Here is a detail of the structure that you will find in the project.

```
qwik-e-commerce
├── src
|   └── routes
|   ├── sign-in
|   |   └── index.tsx
|   ├── sign-up
|   |   └── index.tsx
```

Create Supabase Client

Before defining the various routes, let's create the method that creates the Supabase client with the two environment variables created previously.

FILE: `src/utils/supabase.ts`

```
import { createClient } from "@supabase/supabase-js";

export const supabaseClient = createClient(
  import.meta.env.VITE_SUPABASE_URL,
  import.meta.env.VITE_SUPABASE_ANON_KEY
);
```

Sign Up

You define the `sign-up` route; this specific page is used to register as a user in the application.

FILE: `src/routes/sign-up/index.tsx`

```
import { supabaseClient } from "~/utils/supabase";

export default component$(() => {
  const emailSig = useSignal("");
  const passwordSig = useSignal("");
  const messageSig = useSignal("");
  return (
    <div>
      <label for="email" class="...">
        Your email
      </label>
      <input
        type="email"
        id="email"
        value={emailSig.value}
        onInput$={(_, el) => {
          emailSig.value = el.value;
        }}
      />

      <label for="password" class="...">
        Password
      </label>
      <input
        type="password"
        id="password"
        bind:value={passwordSig}
      />

      {!!messageSig.value && <div>{messageSig.value}</div>}

      <button
        onClick$={async () => {
          const { error } =
            await supabaseClient.auth.signUp({
              email: emailSig.value,
              password: passwordSig.value,
            });
          messageSig.value = error
            ? "Error"
            : "Success. Please check your email/spam folder";
        }}
```

```
      >
        Sign up
      </button>
    </div>
  );
});
```

Two inputs, `email` and `password`, are created for entry, and several `useSignal$` to manage the state of the component. In the first input, you defined the value of the input `value={emailSig.value}` and you subscribed to its change to update the internal status relating to the email `onChange$={(event) => { emailSig. value = event.target.value; }}`.

For the second input, that of the password, you used a special Qwik `bind:value` syntax, which allows you to write less code and guarantees a truly rewarding Developer Experience. Finally, you have the button that executes the call to Supabase, which requests user registration. In this call, you pass `email` plus `password`, and you will update the message based on Supabase's response.

Following registration, an email is sent with a link to confirm registration. In this phase, the user status is `Waiting for verification..`, but as soon as you press the link received via email, you will be redirected to your app, and the user status will be confirmed. The code is executed on the client side, but it is possible to perform the operations on the server side. In fact, for the `sign-in` route, the operation will be performed via `routeAction$` and, therefore, on the server side.

Sign In

FILE: `src/routes/sign-in/index.tsx`

```
import { supabaseClient } from "~/utils/supabase";

export const useSignInAction = routeAction$(
  async ({ email, password }, requestEv) => {
    const { data } =
      await supabaseClient.auth.signInWithPassword({
        email,
        password,
      });
    if (data.session) {
      requestEv.cookie.set(
        "supabase_access_token",
        data.session.access_token,
        {
          path: "/",
        }
      );
```

```
      throw requestEv.redirect(308, "/");
    }
    return { success: false };
  },
  zod$({
    email: z.string(),
    password: z.string(),
  })
);

export default component$(() => {
  const signInAction = useSignInAction();
  return (
    <Form action={signInAction}>
      <div>
        <label for="email">Your email</label>
        <input type="email" name="email" id="email" />
      </div>
      <div>
        <label for="password">Password</label>
        <input
          type="password"
          name="password"
          id="password"
        />
      </div>
      <button type="submit">Sign in</button>
    </Form>
  );
});
```

Here, with the `Form` component, you send the data inserted in the `email` and `password` inputs and use them on the server side to log in with the `signInWithPassword` method of Supabase. If the credentials are correct, the operation will be successful. Here, you take the `access_token` from the user session and save it in the cookies. It is important to note that you are going to set the `path` option to be able to read the cookie in the whole application. Finally, you perform a redirect to the homepage of your app.

Homepage

FILE: `src/routes/index.tsx`

```
import { supabaseClient } from "~/utils/supabase";

export const useUser = routeLoader$(async (requestEv) => {
  const supabaseAccessToken = requestEv.cookie.get(
    "supabase_access_token"
  );
  if (!supabaseAccessToken) {
    return null;
  }
  const { data, error } = await supabaseClient.auth.getUser(
    supabaseAccessToken.value
  );
  return error ? null : data.user;
});

export default component$(() => {
  const userSig = useUser();
  return (
    <h1>
      Welcome{" "}
      {userSig.value ? userSig.value.email : "guest"} 
    </h1>
  );
});
```

Here is the homepage; for now, it is very simple, but on the server side, you are going to use the cookie to recover the user, and if the operation is successful, you are going to display the user's email on the page. Here, you have customized the homepage based on whether the user is logged in or not.

Now you have everything configured correctly. To perform the complete tour, you can start from the `sign-up` page and enter your email and password. Once you have entered this data and pressed the button to confirm, you can move to the `sign-in` page to log in to the application. In fact, by entering the email and password used for registration, you will be redirected from the main screen, where you will see a personalized message with your email.

Summary

During this chapter, a lot of ground was covered. The discussion began by exploring how to architect applications, what elements to consider, how to manage deadlines effectively, and how to view the deadline as a constraint without ignoring its existence. It's important for you to lay all the variables on the table, understand what they are, and prioritize them to avoid promising outcomes that can't be achieved due to constraints. If you're working with a tight deadline, it's better to use tools that provide more security and allow you to deliver value in the shortest time possible. However, it's been learned that sticking to what's

always been used isn't a good practice and doesn't allow for approaching application development with an open mind. It's worth noting that it's not always necessary to write code to complete a task.

To illustrate this point, consider an example. Suppose a manager needs to export data from a database table. After discussing it with the team, it's realized the task is quite demanding. So, a crucial question is asked: How often will this export be needed? The answer is just once. In this case, the task was solved without writing any scheduled data export code. A simple automatic extraction in CSV format was enough to achieve the desired result. Some strategies for communicating effectively with managers and persuading non-technical people to consider your needs were also shared.

In addition to functional constraints, there are also non-functional ones. Some examples were provided to clarify that not only the technical aspects need to be considered but also some metrics that don't strictly pertain to the code. This approach helps you work more efficiently and organize your tasks better. The reasons why Supabase is an excellent choice for the application you want to create were then explored. By examining its features, you can see that it's not an inferior alternative to a physically installed database, but rather an optimal solution for managing your data in the Cloud. The first table was created in a straightforward manner and this data was read via the Qwik application.

The Supabase SDK makes everything quite simple. It was learned that creating an authentication system from scratch is not easy, with data security being the most challenging aspect. Therefore, reliance on services that offer this as a product is recommended. These products use the JWT token, a complex yet simple and ingenious system for managing client authentication. The token is encrypted with a secret, allowing you to verify that the communication between the client and server is secure. You can also include some extra information in the token. Once the token is obtained after login, it's ready to use in all HTTP calls.

The JWT process can be likened to hotel check-in. You can dine at the hotel restaurant without being guests or staying overnight, but if you want to access certain areas like the spa, gym, or a room, you need to authenticate at the reception and receive the hotel room key, which is your JWT token.

Finally, authentication within the application was developed by creating the Sign-up, Sign-in, and Homepage pages. The homepage displays different content based on whether the user is authenticated or not. These pages execute the logic on the backend side, which allows for easy use of API keys. If support for authentication with another vendor was desired, it wouldn't be a problem because the method of the SDK used would just need to be changed. Therefore, supporting authentication via Google is not an issue.

In the next chapter, you will add products to the database and display them in your application. You will also add a search bar to make it easier to find the products to buy.

Rendering products with Orama full text search

Supabase CLI

In the previous chapter, the Supabase table was manually defined within the cloud dashboard to verify the reading and integration with Supabase. However, for the subsequent steps of the application, the most appropriate method to carry out this kind of procedure will be used. In everyday development, a local environment is needed that allows the writing and testing of code by simulating the production or staging environment as closely as possible. In the short term, manually modifying and creating the various data tables is not a good practice because it is prone to errors. If a column needs to be added to the table, it is unthinkable to delegate the activity to a person. With a thousand steps and tasks to perform, the risk of error is too high, and the changes made to the database would not be versioned. Let's imagine a new feature needs to be added to the user profile, such as showing the profile photo within the personal area. The front end will need all the data to display the image and ensure that it is correctly visible in all resolutions. This work is purely related to the Qwik code, but the image information from the database is needed. Therefore, along with this functionality, this new field on the Supabase table is also needed. Ideally, the database should be modified only when the functionality is available. So, remember to update the database only when the functionality is ready and released. If it's an addition of a column, it's a trivial activity. But what if the user table data needs to be changed to split the user name into two separate fields, first name, and last name? If the code expects a single field and the data is modified, the application will have problems because it will be inconsistent. Here, using an automated and versioned migration system saves a lot of problems and makes it easier to release features.

Managing the entire process with Supabase is very simple because the tool's CLI is available, which allows all operations to be performed very quickly and securely.

To use the Supabase CLI, install the dependency within the project with the following command:

```
// Supabase CLI as dev dependency via pnpm:
pnpm install -D supabase
```

Inside the package.json, a dedicated script for Supabase can be created so it can be used like this: `pnpm supabase <command>`:

FILE: `package.json`

```
  "scripts": {
    [...]
    "supabase": "supabase"
  },
```

So far, so good. Now, first of all, execute the following command to initialize the Supabase database locally.

```
pnpm supabase init

## Supabase init steps
#
## pnpm supabase init
## Generate VS Code workspace settings? [y/N] y
## Open the qwik-e-commerce.code-workspace file in VS Code.
## Finished supabase init.
```

Now, there is a new Supabase folder in the project. Then, link the local project with the Supabase instance present on their cloud platform. Using these commands, execute the project linking process, where `$PROJECT_ID` is the project ID that can be found in the URL `https://supabase.com/dashboard/project/<project-id>`

```
pnpm supabase login
pnpm supabase link --project-ref $PROJECT_ID
```

Once the project is connected, install Docker[43] locally to manage the local development stack.

> **Docker**: it is a tool that allows us to create and manage virtual environments that perform better than the Virtual Machines seen in the previous chapters. It is a more optimized approach because they do not have a graphical interface; you can only interact with the terminal. To fully understand Docker, we would need a dedicated book; we just need to know that Supabase needs an environment where we can install and manage our database, and it is synchronized with the Supabase cloud.

Once this tool has been installed, use the following command to create the local Docker environment that will host the local data.

```
pnpm supabase start
```

Finally, to synchronize the local database with the remote one, the one that has already been created in the previous chapters, execute this command:

[43] https://docs.docker.com/desktop/

```
pnpm supabase db pull
```

Excellent! Now everything is configured and synchronized locally, and it's time to create the table to host the products that will be shown in the e-commerce.

Switch to the Local Supabase Instance

Once the `pnpm supabase start` command process is finished, new API KEYs will be provided to access the local database instead of pointing to the remote one present in Supabase Cloud. Create a `.env.local` file in the root of the project to work locally in total freedom, disconnected from the main database. To retrieve this information at a later time, use the `pnpm supabase status` command, which will show these values again.

```
pnpm supabase status

#Output
#
## API URL: http://127.0.0.1:54321
## [...]
## anon key: XXXX
## [...]
```

Take these values and insert them into the file, as shown below.

FILE: `.env.local`

```
VITE_SUPABASE_URL=...
VITE_SUPABASE_ANON_KEY=...
```

Supabase db Migrations

Firstly, you'll want to delete the `test` table that was created, and then create a product table. This can be done in a precise and punctual manner, thanks to the automatic Supabase tool, to avoid carrying out operations manually. To create a new migration, initiate everything from the terminal using the command:

```
pnpm supabase migration new products
```

A file will be created in the migration folder:

```
qwik-e-commerce
├── supabase
|   ├── migrations
|   |   └── <timestamp>_products.sql
```

There are other files managed independently by Supabase, but specifically, this file is the focus. In the first part of the `<timestamp>_products.sql` file, the date of its creation is noted, while in the second <timestamp>_-`products.sql`, the name that was previously assigned with the command in the terminal is present. In this file, SQL can be defined, which will be executed to modify the database. Let's define the table:

```
drop table public.test;

create table public.products (
  id bigint generated by default as identity,
  name text null,
  description text null,
  price double precision null,
  image text null,
  constraint products_pkey primary key (id)
) tablespace pg_default;
```

Here, the fields of the table are being defined, and it can be seen that the id field is incremental, so there's no need to value it; it will be done automatically. Also, the other fields in the table can be noticed. The information regarding the product needed is:

- product name
- description
- price
- a beautiful image that encourages the customer to buy

So, without wasting time, populate the database with this information.

Populate the `seed.sql` file, which is inside the `supabase` folder. This file is executed after all migrations. A good practice is not to add schema statements to your seed file, only data.

```
insert into public.products
  (name, description, price, image)
values
  ('Spiky Cactus', 'A spiky yet elegant house cactus - perfect for the home or offic\
e. Origin and habitat: Probably native only to the Andes of Peru', 20.95,  'spiky_ca\
ctus.webp'),
  ('Tulip Pot', 'Bright crimson red species tulip with black centers, the poppy-like\
 flowers will open up in full sun. Ideal for rock gardens, pots and border edging.',\
 10.95, 'tulip_pot.webp'),
  ('Aloe Vera', 'Decorative Aloe vera makes a lovely house plant. A really trendy pl\
ant, Aloe vera is just so easy to care for.', 10.45, 'aloe_vera.webp'),
  ('Fern Blechnum Gibbum', 'Create a tropical feel in your home with this lush green\
 tree fern, it has decorative leaves and will develop a short slender trunk in time.\
', 12.95, 'fern_blechnum_gibbum.webp'),
  ('Assorted Indoor Succulents', 'These assorted succulents come in a variety of dif\
ferent shapes and colours - each with their own unique personality.', 42.35, 'assort\
ed_indoor_succulents.webp'),
  ('Orchid', 'Gloriously elegant. It can go along with any interior as it is a neutr\
al color and the most popular Phalaenopsis overall.', 30.75, 'orchid.webp');
```

Here, the database products have been defined. Now, all that's left to do is launch this command:

```
pnpm supabase db reset
```

This command recreates the local database from scratch. It runs all the migration scripts present in the `supabase/migrations` directory to recreate the optimal and clean situation. Furthermore, as seen before, it will also insert the data by reading and executing the `seed.sql` file. This automatic process has the advantage of being replicable, and therefore, all developers on the team may be able to set up their environment without problems so they can be productive immediately.

With the `pnpm supabase start` command, a local copy of the environment present in the Supabase cloud has been recreated. There's also the possibility of accessing a dashboard completely identical to the one available in Supabase, but with the advantage that it is local. Therefore, generating the migrations starting from the local dashboard and then generating the migration via the Supabase CLI is also possible. With this `diff` command, a new migration script from changes already applied to the local database can be created.

```
pnpm supabase db diff -f new_migration_name
```

This makes the process even smoother and faster. These migration and seeding files will be part of the project and the versioned code in all respects.

Align Remote Database

By executing the following command, the database situation can be checked to understand if there are misalignments.

```
pnpm supabase migration list

## Output
#
## Connecting to remote database...
#
##       LOCAL       |     REMOTE     |     TIME (UTC)
## ------------------|----------------|---------------------
##   migration-123   | migration-123  |   last_update_time
##   migration-456   |                |   last_update_time
```

Here, the situation of the databases can be seen, and it can be noticed that the remote one is not aligned with what is locally present. This is because the `test` table was removed and the `products` one was created, and these changes are not available in the remote DB. To align the remote database on the Supabase cloud, use this command:

```
pnpm supabase db push
```

To carry out an alignment and perform the migrations, at the end of the operation, the `products` table will also be found in the remote database. But in fact, with this operation, the functionality has not been tied to database modification. It has been seen previously that this is a good practice, and this level of process and alignment between functionality and database can be obtained thanks to the use of GitHub Actions.

GitHub Action

GitHub offers a feature called GitHub Actions, which gives the ability to launch tasks of any type when a GitHub event occurs. For example, when new code is pushed to the repository according to a pre-established schedule or when an external event occurs using the repository submission webhook. Often, these tasks are set up in repositories to verify PRs and perform a series of checks before maintainers can even evaluate changes made to the project.

It is possible to use these flows to deploy packages or release projects, too. (e.g., every push on the main branch, the public site relating to the project documentation can be updated) This GitHub functionality will be used to perform migrations of the various environments available in Supabase. By creating configuration files inside the `.github/workflows` folder, automated procedures can be created, and a real Continuous Integration[44] can be established.

So, let's analyze this file:

FILE: `.github/workflows/production.yaml`

[44] https://en.wikipedia.org/wiki/Continuous_integration

```yaml
name: Deploy Migrations to Production

on:
  push:
    branches:
      - main
  workflow_dispatch:

jobs:
  deploy:
    runs-on: ubuntu-latest

    env:
      SUPABASE_ACCESS_TOKEN: ${{ secrets.SUPABASE_ACCESS_TOKEN }}
      SUPABASE_DB_PASSWORD: ${{ secrets.PRODUCTION_DB_PASSWORD }}
      SUPABASE_PROJECT_ID: ${{ secrets.PRODUCTION_PROJECT_ID }}

    steps:
      - uses: actions/checkout@v3

      - uses: supabase/setup-cli@v1
        with:
          version: latest

      - run: supabase link --project-ref $SUPABASE_PROJECT_ID
      - run: supabase db push
```

By creating the file inside the previously mentioned folder, the workflow can be defined. Here, it is being said that every time a push is made on the `main` branch, a series of steps with the files that are present inside the project are also performed. The files are checked out, and the project is practically downloaded. With the Supabase CLI, the link operations to the project present in the Supabase cloud and the database push are performed. These are the same operations that were done step by step in the previous section, and the workflow behaves in the same way. This allows the database migration to be performed only at the correct moment in time, only when the application code needs it.

If this process needs to be brought to other environments, too, all that's needed is to create a specific file for the environment and modify the variables and the branch that are included in the configuration.

> **Secrets variables**: It is good practice to save secrets within your GitHub and then use them in configuration files. This allows us to avoid revealing these variables that should not be made public. There is a special section within GitHub where you can create your variables, which can be made visible in different ways within the organization; you can share a variable between multiple repositories or limit its visibility to a single project.

Displaying Products on the Homepage

Now that the products are in the database, the next step is to display them on video. This will allow users to purchase them and start seeing the first orders come in. In the previous chapters, the process of reading and displaying data from Supabase was learned. There's no need to invent anything new here; the knowledge is already there. All that's needed is to write the code to generate a beautiful list of products.

FILE: `src/routes/index.tsx`

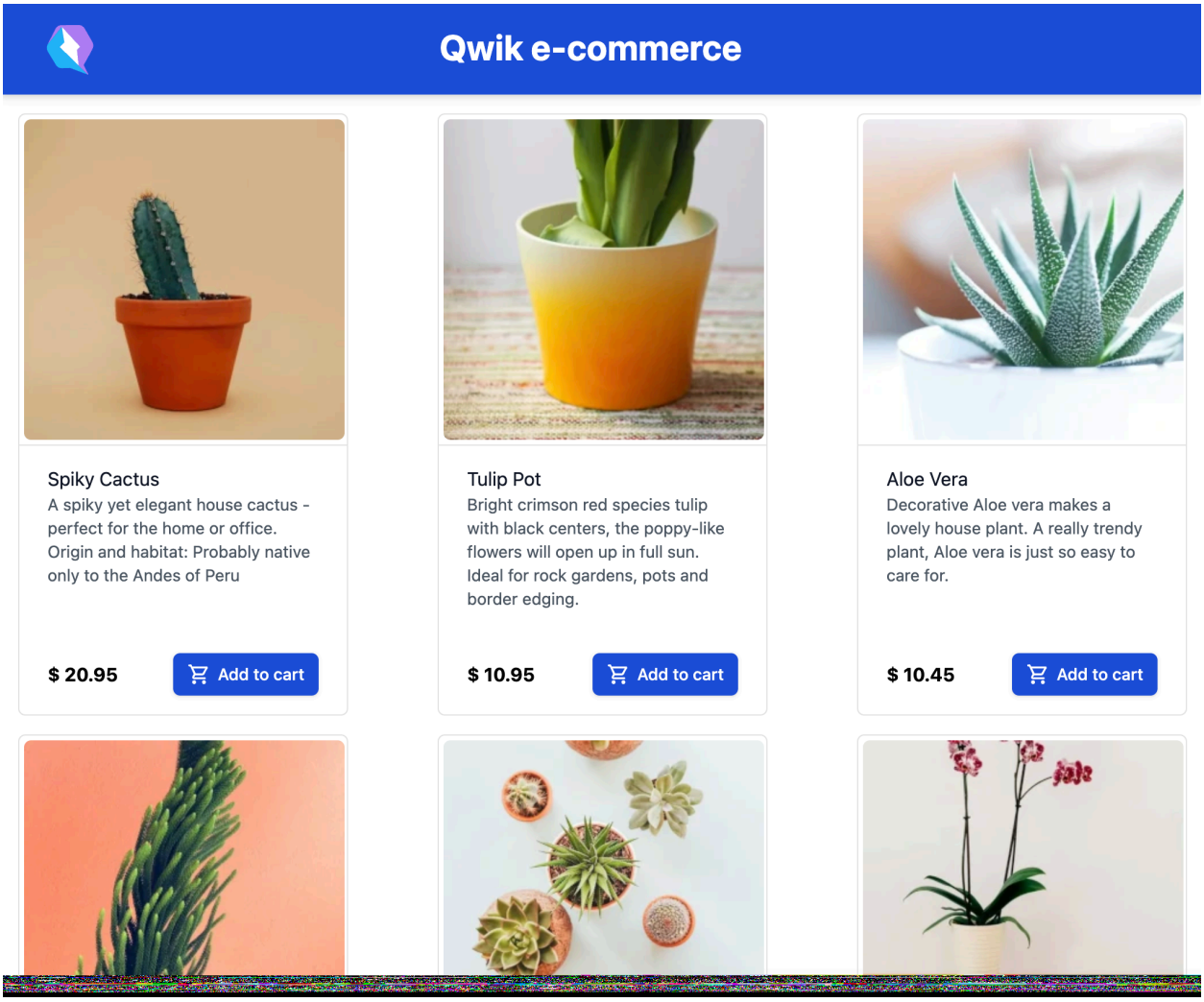
```
import { IconShoppingCart } from "~/components/IconShoppingCart";
import { supabaseClient } from "~/utils/supabase";

export const useUser = routeLoader$(async (requestEv) => {
  const supabaseAccessToken = requestEv.cookie.get(
    "supabase_access_token"
  );
  if (!supabaseAccessToken) {
    return null;
  }
  const { data, error } = await supabaseClient.auth.getUser(
    supabaseAccessToken.value
  );
  return error ? null : data.user;
});

type Product = {
  id: number;
  name: string;
  description: string;
  price: string;
  image: string;
};

export const useProducts = routeLoader$(async () => {
  const { data } = await supabaseClient
    .from("products")
    .select("*");
  return data as Product[];
});

export default component$(() => {
  const userSig = useUser();
  const productsSig = useProducts();
```

```
  const navigate = useNavigate();

  return (
    <div class="...">
      {productsSig.value.map((product) => (
        <div key={product.id} class="...">
          <div class="...">
            <div class="...">
              <img
                loading="eager"
                width={280}
                height={280}
                class="..."
                src={`/images/${product.image}`}
                alt={product.name}
              />
            </div>
          </div>
          <div class="...">
            <span class="...">{product.name}</span>
            <div class="...">{product.description}</div>
            <div class="...">
              <span class="...">$ {product.price}</span>
              {userSig.value ? (
                <button
                  type="button"
                  class="..."
                  onClick$={() =>
                    console.log("Add to cart!")
                  }
                >
                  <IconShoppingCart />
                  Add to cart
                </button>
              ) : (
                <button
                  type="button"
                  class="..."
                  onClick$={() => navigate("/sign-in")}
                >
                  Sign In
                </button>
```

```
77                )}
78              </div>
79            </div>
80          </div>
81        ))}
82      </div>
83    );
84  });
```

Here is the implementation of the homepage to display products. There are two `routeLoader$`. The first one, already familiar from the previous chapter, allows the determination of whether the user is signed in. The second `routeLoader$`, which is needed to read from the `products` Supabase table, has also been seen before. Once the data is read, it's rendered graphically on the homepage. With `key={product.id}`, a unique key is assigned to each element in the list, and then the image and information related to the product are displayed. One last note: the button changes depending on whether the user is signed in. If signed in, the product is added to the cart (for this first step, it's a console.log). Otherwise, a button is displayed that, thanks to the `navigate` function, directs to the sign-in page.

Here is the result.



products

Orama Full-Text Search

Orama is an excellent library that enables searching within complex texts and more. It's written in TypeScript and designed for use in cloud environments, highly optimized for speed and low latency. It offers advanced search capabilities, including the `Type tolerance` feature. Let's say users want to search for `Aloe`, but they mistakenly type `Aleo`. A simple string comparison wouldn't yield a potential result, but the powerful algorithms used by Orama make this possible. Results can be suggested to users, enhancing e-commerce performance and increasing earnings. To add Orama to Qwik, simply use the

```
pnpm qwik add orama
```

command. The Qwik CLI will add and install the dependencies and create the necessary configuration files for its operation.

Let's see how the code has changed.

FILE: src/routes/index.tsx

```
import type { Orama } from "@orama/orama";
import { create, insert, search } from "@orama/orama";
import { IconShoppingCart } from "~/components/IconShoppingCart";
import { supabaseClient } from "~/utils/supabase";

export const useUser = routeLoader$(async (requestEv) => {
  const supabaseAccessToken = requestEv.cookie.get(
    "supabase_access_token"
  );
  if (!supabaseAccessToken) {
    return null;
  }
  const { data, error } = await supabaseClient.auth.getUser(
    supabaseAccessToken.value
  );
  return error ? null : data.user;
});

type Product = {
  id: number;
  name: string;
  description: string;
  price: number;
  image: string;
};

let oramaDb: Orama;

export const useProducts = routeLoader$(async () => {
  const { data } = await supabaseClient
    .from("products")
    .select("*");
  oramaDb = await create({
    schema: {
      id: "string",
      name: "string",
      description: "string",
      price: "number",
      image: "string",
```

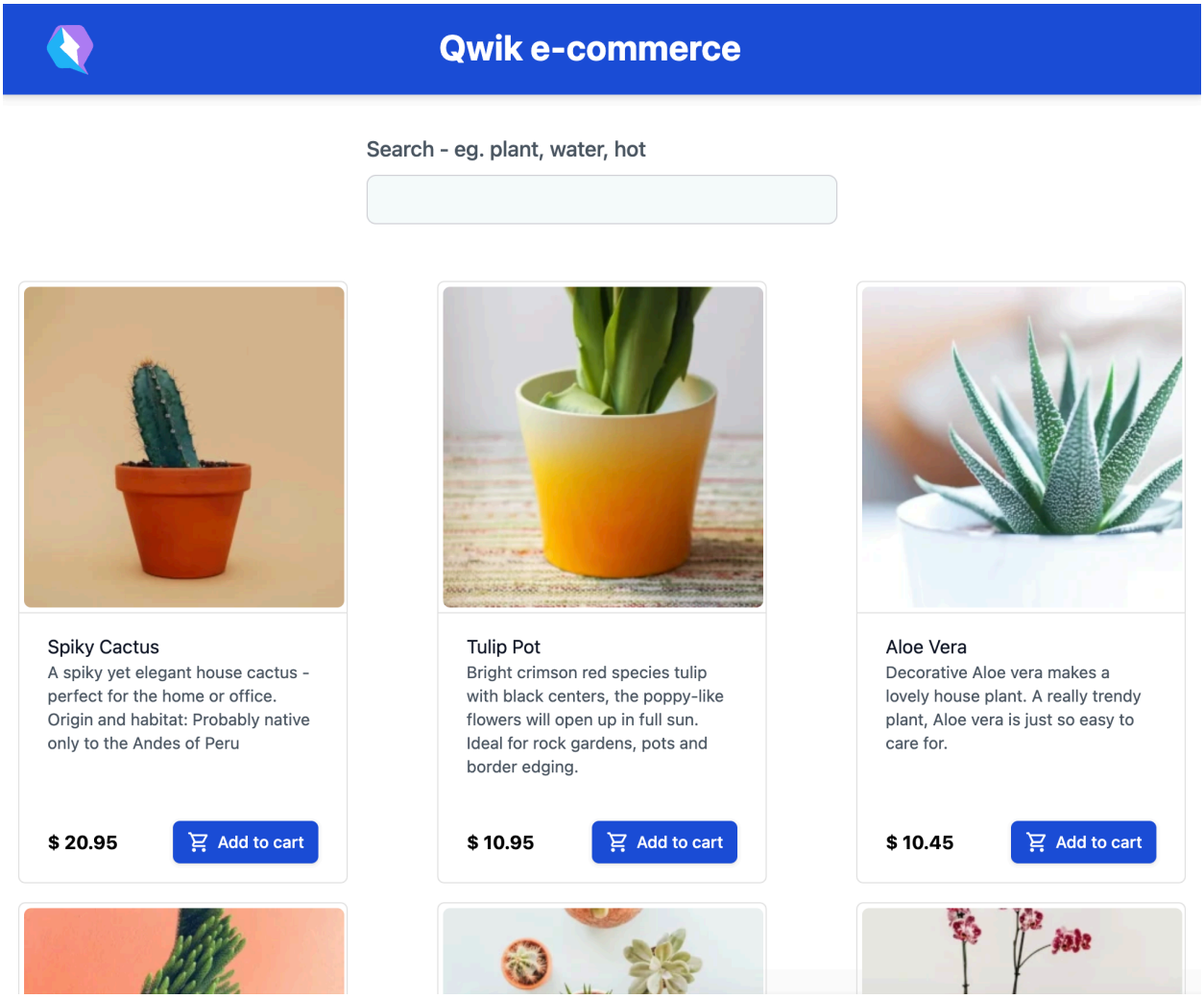
```
    },
  });
  if (data) {
    data.map(
      async (product: Product) =>
        await insert(oramaDb, {
          ...product,
          id: product.id.toString(),
        })
    );
  }
  return data as Product[];
});

export const execSearch = server$(async (term: string) => {
  const response = await search(oramaDb, {
    term,
    properties: "*",
    boost: { name: 1.5 },
    tolerance: 2,
  });
  return response;
});

export default component$(() => {
  const userSig = useUser();
  const termSig = useSignal("");
  const navigate = useNavigate();
  const productsSig = useProducts();
  const resultsSig = useSignal<Product[]>(
    productsSig.value
  );

  const onSearch = $(async (term: string) => {
    if (term === "") {
      resultsSig.value = productsSig.value;
      return;
    }

    const response = await execSearch(term);
    resultsSig.value = response.hits.map(
      (hit) => hit.document as unknown as Product
```

```
    );
  });

  return (
    <div class="...">
      <label class="...">
        Search - eg. plant, water, hot
      </label>
      <input
        type="text"
        class="..."
        bind:value={termSig}
        onKeyDown$={(e) => {
          if (e.key === "Enter") {
            onSearch(termSig.value);
          }
        }}
      />
      {resultsSig.value.map((product) => (
        <div key={product.id} class="...">
          <div class="...">
            <div class="...">
              <img
                loading="eager"
                width={280}
                height={280}
                class="..."
                src={`/images/${product.image}`}
                alt={product.name}
              />
            </div>
          </div>
          <div class="...">
            <span class="...">{product.name}</span>
            <div class="...">{product.description}</div>
            <div class="...">
              <span class="...">$ {product.price}</span>
              {userSig.value ? (
                <button
                  type="button"
                  class="..."
                  onClick$={() =>
```

```
                    console.log("Add to cart!")
                  }
                >
                  <IconShoppingCart />
                  Add to cart
                </button>
              ) : (
                <button
                  type="button"
                  class="..."
                  onClick$={() => navigate("/sign-in")}
                >
                  Sign In
                </button>
              )}
            </div>
          </div>
        </div>
      ))}
    </div>
  );
});
```

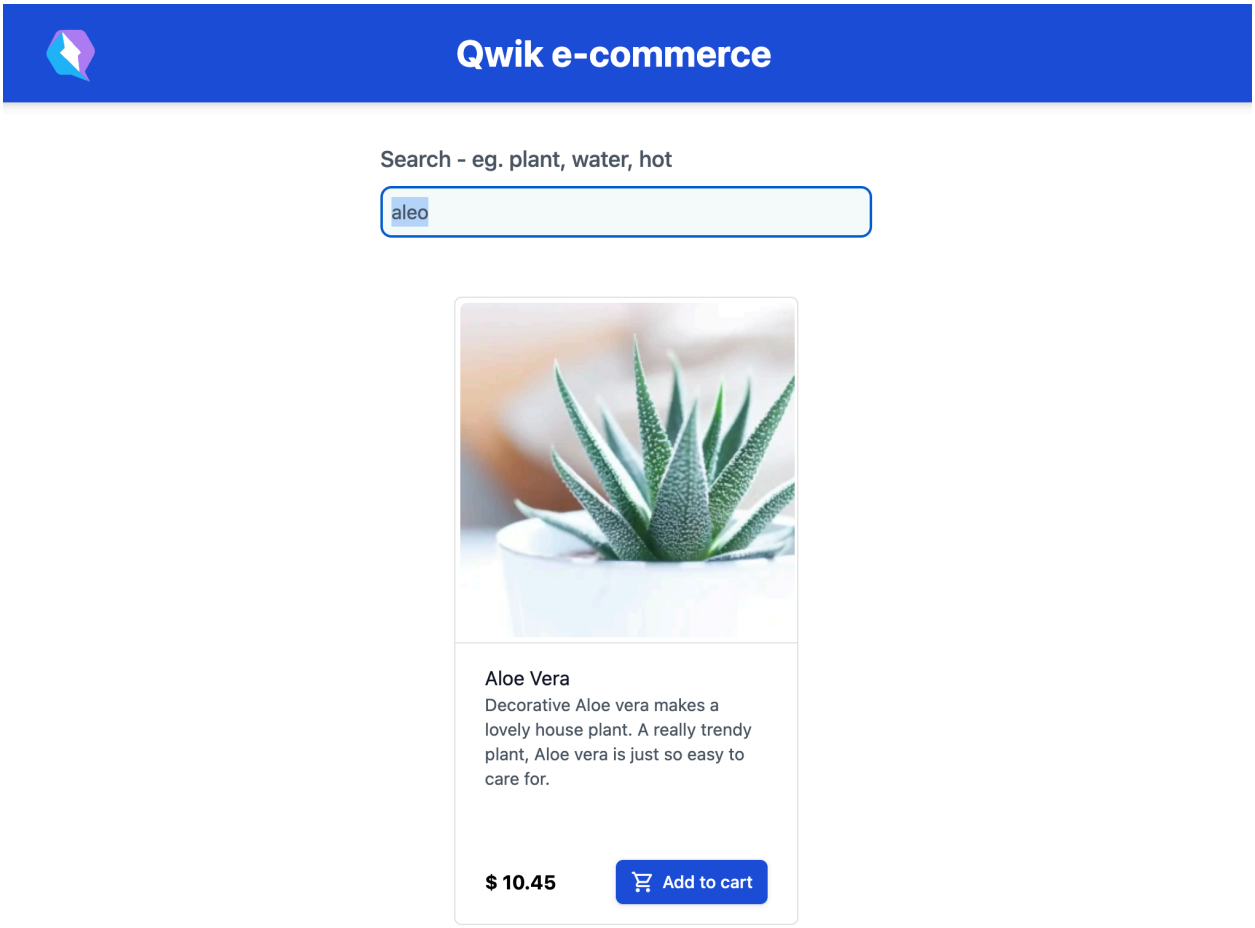
A server-side Orama instance was created and seeded with the products read from Supabase. Above the product list, a search input was created. User input is tracked using `termSig`, and when the `Enter` key is pressed, the search is performed. This operation only occurs if the input is populated. If it's empty, all the products are returned. When a search needs to be conducted, the function enclosed with `server$` is executed. The calculation will then be performed on the server, and the products that match the search filter will be returned. In the query executed by Orama in the `execSearch` function, `boost` was used to give more importance to the product name and the `tolerance` property to offer suggestions in case of typos.

Here is the result.



search

Here's an example of a typo.



search_filtered

Product Detail

Now that we have our home page where we show the catalog we are very happy because we are seeing our application take shape. As we have seen, it is very simple to create our pages and the integration with Supabase is truly sensational for how simple and fast it is. But now we want to implement a detail page for our products, this allows us to add more useful information to the user because we can, for example, show product reviews left by other buyers. Furthermore, another functionality that we can implement is to show the stock of the product itself on the page. Let's see how we can create a dedicated route for the detail part. First, we need to create a route to view the details of the article so let's create this structure:

```
qwik-e-commerce
├── src
|   ├── routes
|   |   ├── detail
|   |   |   ├── [slug]
|   |   |   |   └── index.tsx
```

Let's analyze the newly created structure in detail, under the `routes` folder we have the `detail` folder, and inside we have a `[slug]` folder. Here we define the component that will render the detail of our product. So with this structure, we have defined a new application route that will be `https://my-website.com/detail/<product_slug>/` Here we can see that we are referring to a field called slug which is not currently present in our application. So as we said before we can exploit Supabase migrations to create the new field.

Let's create the file where we can write our migration:

```
pnpm supabase migration new add_product_slug_field
```

And inside the new file `<timestamp>_add_product_slug_field.sql` we are going to insert our migration script.

```
alter table "public"."products" add column "slug" text;
```

Let's update our `seed.sql` file with the new field: `slug`.

```
insert into public.products
  (name, description, price, image, slug)
values
  ('Spiky Cactus', 'A spiky yet elegant house cactus - perfect for the home or offic\
e. Origin and habitat: Probably native only to the Andes of Peru', 20.95,  'spiky_ca\
ctus.webp', 'spiky-cactus'),
  ('Tulip Pot', 'Bright crimson red species tulip with black centers, the poppy-like\
 flowers will open up in full sun. Ideal for rock gardens, pots and border edging.',\
 10.95, 'tulip_pot.webp', 'tulip-pot'),
  ('Aloe Vera', 'Decorative Aloe vera makes a lovely house plant. A really trendy pl\
ant, Aloe vera is just so easy to care for.', 10.45, 'aloe_vera.webp', 'aloe-vera'),
  ('Fern Blechnum Gibbum', 'Create a tropical feel in your home with this lush green\
 tree fern, it has decorative leaves and will develop a short slender trunk in time.\
', 12.95, 'fern_blechnum_gibbum.webp', 'fern-blechnum-gibbum'),
  ('Assorted Indoor Succulents', 'These assorted succulents come in a variety of dif\
ferent shapes and colours - each with their own unique personality.', 42.35, 'assort\
ed_indoor_succulents.webp', 'assorted-indoor-succulents'),
  ('Orchid', 'Gloriously elegant. It can go along with any interior as it is a neutr\
```

```
al color and the most popular Phalaenopsis overall.', 30.75, 'orchid.webp', 'orchid'\
);
```

And we launch the following command to apply the changes to our local database.

```
pnpm supabase db reset
```

Once the migration is complete we are ready to render the details of our articles. So let's implement the detail page.

FILE: `src/routes/detail/[slug]/index.tsx`

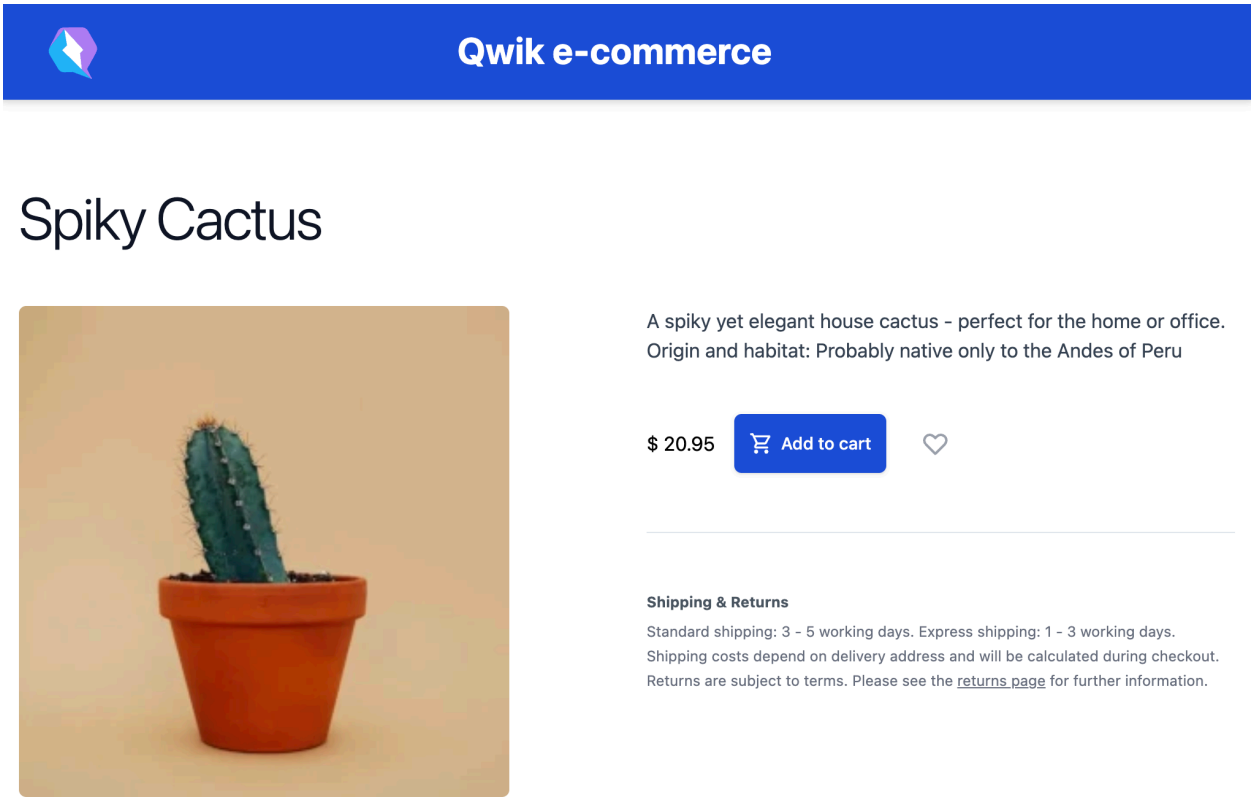
```tsx
import type { PostgrestSingleResponse } from "@supabase/supabase-js";
import { HeartIcon } from "~/components/HeartIcon";
import { IconShoppingCart } from "~/components/ShoppingCartIcon";
import { useUser } from "~/routes/layout";
import type { Product } from "~/utils/store";
import { supabaseClient } from "~/utils/supabase";

export const useProductDetail = routeLoader$(
  async ({ params, status }) => {
    const slug = params.slug;
    const { data }: PostgrestSingleResponse<Product[]> =
      await supabaseClient
        .from("products")
        .select("*")
        .eq("slug", slug);

    if (!data) {
      status(404);
    }

    return data ? data[0] : null;
  }
);

export default component$(() => {
  const userSig = useUser();
  const navigate = useNavigate();
  const productDetail = useProductDetail();

  if (!productDetail.value) {
    return (
```

```
      <div>
        Sorry, looks like we don't have this product.
      </div>
    );
  }

  return (
    <div>
      <div class="...">
        <div>
          <h2 class="...">{productDetail.value.name}</h2>
          <div class="...">
            <div class="...">
              <span class="...">
                <div class="...">
                  <img
                    loading="eager"
                    width={400}
                    height={400}
                    class="..."
                    src={`/images/${productDetail.value.image}`}
                    alt={productDetail.value.name}
                  />
                </div>
              </span>
            </div>
            <div class="...">
              <div class="...">
                <h3 class="...">Description</h3>
                <div class="...">
                  {productDetail.value.description}
                  <div />
                </div>
                <div class="mt-10 flex flex-col sm:flex-row sm:items-center">
                  $ {productDetail.value.price}
                  <div class="flex sm:flex-col1 align-baseline px-4">
                    {userSig.value ? (
                      <button
                        type="button"
                        class="..."
                        onClick$={() =>
                          console.log("Add to cart!")
```

```
                        }
                      >
                        <IconShoppingCart />
                        Add to cart
                      </button>
                    ) : (
                      <button
                        type="button"
                        class="..."
                        onClick$={() =>
                          navigate("/sign-in")
                        }
                      >
                        Sign In
                      </button>
                    )}
                    <button type="button" class="...">
                      <HeartIcon />
                      <span class="...">
                        Add to favorites
                      </span>
                    </button>
                  </div>
                </div>

                <section class="...">
                  <h3 class="...">Shipping & Returns</h3>
                  <div class="...">
                    <p>
                      Standard shipping: 3 - 5 working days.
                      Express shipping: 1 - 3 working days.
                    </p>
                    <p>
                      Shipping costs depend on delivery
                      address and will be calculated during
                      checkout.
                    </p>
                    <p>
                      Returns are subject to terms. Please
                      see the{" "}
                      <span class="...">returns page</span>{" "}
                      for further information.
```

```
                    </p>
                  </div>
                </section>
              </div>
            </div>
          </div>
        </div>
      </div>
    );
});
```

Here is the result:



product_detail

Let's analyze this implementation. On the server side, inside the routeLoader$, we are reading the parameters, and retrieving the slug typed in the address bar. With this value, we are going to execute a query on Supabase to retrieve the product from our database. If the product does not exist we will display a fallback message indicating that there are no products available for that specific slug. If we have managed to find a product then we are going to render it on the page and if the user is logged in we are going to show the button to add the product to the cart. As on the homepage, if we are not logged in then the button will have a different label

"Sign In" which will take us to the sign-in page. Furthermore, other extra information is shown that explains the shipping conditions. Here we were able to create our detail page quickly and easily, we performed the reading on the server side and we did not perform any unnecessary JavaScript operations.

Cache-control headers

We have implemented our pages and I would say we have done a good job. Now once the site is online everyone will have the opportunity to see our products. It will happen that when a user requests a certain detail page for one of our articles, our server will execute the routeLoader$ to connect to Supabase and execute a query, with the resulting data it will generate the HTML which will then be returned to the browser. The user will then see the details of our product in the browser. However, this behavior is not optimized because, for each request from any user, our server performs the work of reading the data and generates the HTML to send to the browser. This becomes a problem because the more our server is busy and works, the more we are going to spend. The working time of our server and the cost of our infrastructure are two strongly linked things. The detail page of our products or the list of our items for sale are always the same, they do not change frequently, in fact, new products are added, but only once a month, and once we have published a new item and checked its description, we will no longer need to change the description because it always remains the same. There may be some changes in prices but it is something that does not happen frequently. In short, we are telling ourselves that if our server generates the page for each request, it will often generate the same page. To overcome this type of problem, CDNs are used. By putting a CDN in place we can dramatically improve the response time and reduce our infrastructure and management costs. Let's see how things change with a CDN in place. The first visitor requests a page, this request arrives at the CDN which checks whether it has the requested page in its cache. Being the first to request this page, the CDN does not have this resource so the request is passed to our server which downloads the data from Supabase, generates the HTML page, and responds to the CDN which will forward the response to the user who requested the page. However, if configured correctly, the CDN saves the HTML page in the cache and subsequent requests for that specific page will be satisfied with the page present in the cache to avoid reaching our server and reduce costs because we eliminate the server-side computation. The first user had to wait for the execution of the entire roundtrip described above, but the second, third, millionth user will wait much less because he will receive the page present in the cache of our CDN. Great problem solved, we create a page once and then always serve that to all users. But what happens if we want to change the prices of our items? If we have no way to invalidate the cache of our CDN we cannot show the user the new price, because the user will always see the page created by the famous first user who made the first request. Here, to solve this problem we can instruct the CDN with HTTP headers that are designed precisely for this reason.

> Using cache is a good practice, it is not suitable for all pages. If there are personalized pages per user, for example, profile pages or order history reserved for a user, it is not good practice to store them in the cache. On the other hand, it is very suitable for storing high-traffic pages such as the homepage or other pages that appear the same for everyone. This will help you achieve better performance but above all reduce costs.

With Qwik, we can define these headers within our internal pages. Let's see an example:

FILE: `src/routes/detail/[slug]/index.tsx`

```
export const onGet: RequestHandler = async ({
  cacheControl,
}) => {
  cacheControl({
    maxAge: 60, // In the standard HTTP is max-age
    sMaxAge: 60, // In the standard HTTP is s-maxage
    staleWhileRevalidate: 120, //In the standard HTTP is stale-while-revalidate
  });
};

// Our component and logic here
```

Thanks to `onGet` middleware we can respond to the request by also adding the Cache-control headers. Here we are defining very precise rules. With `maxAge: 60` we are saying: "Dear CDN this page that you are keeping in cache is valid from now and for the next 60 seconds, if you receive calls for the same page within 60 seconds you can use the copy you have in cache." Let's analyze this scenario:

- CDN saves a `/abc` page in the cache
- the developer, update the information on the `/abc` page
- the CDN receives a request for the `/abc` page 50 seconds after it saved the page, in this case, the cached copy will be served
- 60 seconds have passed, the cache has expired for the `/abc` page and the new request, therefore, reaches our server to generate the newly updated page. This page will take the place of the previous one in our CDN cache.

Therefore the cache time is always respected and until the cache is invalidated, users will see outdated content. Nothing prevents us from putting logic inside our `onGet` and changing the value of `maxAge` according to a certain logic. This is a parameter that must be refined over time and we can vary it, not only based on our needs, but also on the context. In fact `maxAge` is a cache that occurs on the browser side where we have no control to invalidate it, but there is another header called `sMaxAge` which is instead at the CDN level. If both parameters are present `sMaxAge` wins. Using a value on the CDN side allows us to purge this data. In the previous scenario, once we modified the data in the `/abc` page we had to wait for the cache to expire, using `sMaxAge` instead we can invalidate the cache in the CDN and so as soon as our content changes we can immediately serve an updated page. It requires more time because you have to perform a purge directly by querying the CDN, but if we can automate this procedure it is an optimal approach. So the strategy is to set a short `maxAge` because we have no control over the browser cache and set a longer `sMaxAge` where instead, through the possibility of performing an automated purge of our CDN cache, we have full control. Let's analyze `staleWhileRevalidate: 120` which allows us to obtain this type of behavior. Let's imagine having a `maxAge` of 60 seconds and receiving a call after this time, say 65 seconds, in this case `staleWhileRevalidate` comes into play because it is inside the 120 seconds we defined. To respond to the request as soon as possible, the CDN responds with the copy it has in the cache even if it has expired. Behind the scenes, the CDN makes a call to our server to generate an updated version of our page to replace the old one that is in the cache. Without `staleWhileRevalidate`, the request arriving after 65 seconds would have made the entire round trip to our server and the user would not have received such a quick response. `staleWhileRevalidate`

therefore defines the period during which even if the cache has expired, to provide a fast response, the cache resource is used and behind the scenes, as mentioned, the updated resource is requested to be used as a new cache in CDN. I wanted to close this section with an analogy. Let's imagine we are managers of a bar and we want to improve the service at peak times. In the morning we know that many people are in a hurry and want to drink a coffee quickly and then go to work. It often happens that customers have to wait to receive their coffee because they all arrive at the same time and it takes time to prepare them all. Knowing that customers start arriving at around 8 in the morning, I prepare a coffee in advance (my cache), when the first request arrives I serve them the coffee immediately because I already have it ready. In the meantime, I'm already preparing another one, so the second customer will also receive their coffee very quickly. What happens though, if I prepare the coffee and within 60 seconds (my maxAge) no one comes to order, I can consider that the coffee is getting cold, it is no longer excellent, but thinking about it I can wait up to 120 seconds (my staleWhileRevalidate). If any customer orders coffee in this period (between 60-120 seconds) then I will serve the coffee I already have and prepare a new one for a future customer. However, if no one orders the coffee, I will be forced to throw it away because it is now cold and no longer good to be served (my cache is expired).

Summary

This chapter was full of content, we immediately saw how to use the Supabase CLI, which dependencies are necessary, and what the commands are to recreate the remote database even locally. The advantages are undeniable because it allows us to work in peace, freeing us from the remote database. Together we modified the database created in the previous chapter and created migration files to modify the database automatically. We also created a seed to initialize the database with useful data for local development. Thanks to Docker and the Supabase CLI this process is very smooth. We have seen how we can update our remote database via the command line, but we also discovered that the most correct process is to tie the implemented functionality and the database modification together. All this is possible thanks to GitHub actions, by doing so the database will only change when necessary, when our code requests new data from the database. By doing so, many errors are avoided and there is no possibility of creating inconsistent situations. In our Qwik application via `routeLoader$`, we read our products from the newly created table. We rendered the various products on the homepage, and we inserted the image, and the data relating to the product (name, description, price). Finally, again using a `routeLoader$`, we checked whether there was a logged-in user or not. If the user is logged in he will see the "Add to cart" button otherwise he will see a button which behind the scenes, thanks to the navigate API, will direct the user to the Sign in page. We have created a search bar to allow our users to search within our catalog. Thanks to the interesting Orama tolerance functionality we can also correct any typing errors made by the user and equally recommend a product to purchase. With the `server$` API, we were able to execute our server-side code in a completely safe and fluid manner without having to worry. At the end of the chapter, we also implemented a detail page for our products, together we analyzed how to manage dynamic routes and we modified our database by adding the slug field to the products table. Finally, we have analyzed in detail the Cache-control headers which allow us to save a lot of money in the management of our infrastructure, because they take the load off the server with the intelligent use of the cache, both at the browser and CDN level. In the next chapter, we will add new features to our application such as adding our products to the cart and managing the checkout process with Stripe.

Adding cart and checkout process with Stripe

Add a product to your favorites list

We have displayed the list of products in our application and it would be nice to be able to add the ability to mark as favorite. It is a very common feature in e-commerce applications that of being able to mark a product as a favorite so that it can then be retrieved at a later time to continue with the purchase or go and repurchase it. In the previous chapter, you may have noticed that there is a heart-shaped icon, to implement this new functionality we can use this graphic element to mark a single article as a favorite. This functionality also offers us the excuse to see together how we can write inside our tables with Supabase because up to now we have only read information, but usually in applications many more operations are performed. Usually to interact with an entity the term CRUD is used to refer to all the operations that can be performed: create, read, update, and delete.

So we are telling ourselves that to implement this functionality we must save in our database that the user has expressed his desire to save that specific product. Thinking about how to structure our database table we can simply think of creating a table with two columns, one with the user ID and one with the product ID these are the two pieces of information that are necessary to track the choices of each user. It can be implemented in a thousand ways, but I think this is the simplest for our context.

Let's create a new migration to create the table that will contain this information.

```
pnpm supabase migration new favorites
```

Let's write the SQL that will generate our table to contain the information we need.

```
create table public.favorites (
  id bigint generated by default as identity,
  user_id uuid not null,
  product_id bigint not null,
  constraint favorites_pkey primary key (id),
  constraint favorites_id_key unique (id),
  constraint favorites_product_id_fkey foreign key (product_id) references products \
(id)
) tablespace pg_default;
```

Now we can go and apply our changes to the database with the following command:

```
pnpm supabase db reset
```

Once the changes have been applied we can modify the code in our application as follows. First of all, we have to modify the products on our homepage to insert a link that allows us to reach the detail directly. Here it will be enough to modify the image of our products to accept the user's click and consequently redirect the user to the product detail page.

FILE: `src/routes/index.tsx`

```
export type Product = {
  id: number;
  [...]
  slug: string; // <- We need to add the slug property
};

export default component$(() => {
  return (
    <div class='...'>
      <div class='...'>
        {resultsSig.value.map((product) => (
          <div key={product.id}>
            [...]
            <img
              loading='eager'
              width={280}
              height={280}
              class='...'
              src={`/images/${product.image}`}
              alt={product.name}
              onClick$={() => {
                navigate(`/detail/${product.slug}`);
              }}
            />
            [...]
          </div>
        ))}
        [...]
      </div>
    </div>
  );
});
```

Here we see that we have added the `slug` property to our Product type so that we can use it in our component. We have added the `onClick$` event to the image so when the user clicks on the image he will be able to go

and see the detail. Precisely on this detail page, we will add the logic to show or not whether the product is among our favorites. But first let's modify the icon to be able to assume the two states, when it is active or not.

FILE: `src/components/HeartIcon/index.tsx`

```
type Props = {
  active: boolean,
  onClick$?: PropFunction<() => void>,
};

export const HeartIcon =
  component$ <
  Props >
  (({ active, onClick$ }) => {
    return (
      <svg
        xmlns="http://www.w3.org/2000/svg"
        class="h-6 w-6 flex-shrink-0"
        fill={active ? "red" : "none"}
        viewBox="0 0 24 24"
        stroke="currentColor"
        stroke-width="2"
        onClick$={() => {
          onClick$ && onClick$();
        }}
      >
        <path
          stroke-linecap="round"
          stroke-linejoin="round"
          d="..."
        />
      </svg>
    );
  });
```

Here we have an additional property `active`, in fact now we can go and set whether we want the icon active or not. Furthermore, we have defined the `onClick$` event to manage the user's click and change the status. This prepares the ground for modifying the product detail page.

FILE: `src/routes/detail/[slug]/index.tsx`

```
import type { PostgrestSingleResponse } from "@supabase/supabase-js";
import { HeartIcon } from "~/components/HeartIcon";
import { IconShoppingCart } from "~/components/IconShoppingCart";
import type { Product } from "~/routes";
import { useUser } from "~/routes/layout";
import { supabaseClient } from "~/utils/supabase";

export const useProductDetail = routeLoader$(
  async ({ params, resolveValue }) => {
    const slug = params.slug;
    const { data }: PostgrestSingleResponse<Product[]> =
      await supabaseClient
        .from("products")
        .select("*")
        .eq("slug", slug);

    if (!data || data.length === 0) {
      return { product: null, isFavorite: false };
    }

    let isFavorite = false;
    const user = await resolveValue(useUser);
    if (user) {
      const favoritesResponse = await supabaseClient
        .from("favorites")
        .select("*")
        .match({
          user_id: user.id,
          product_id: data[0].id,
        });
      isFavorite =
        !!favoritesResponse.data &&
        favoritesResponse.data.length > 0;
    }
    return { product: data[0], isFavorite };
  }
);

export const changeFavorite = server$(
  async (
    userId: string,
    productId: number,
```

```
    isFavorite: boolean
  ) => {
    if (isFavorite) {
      await supabaseClient
        .from("favorites")
        .insert({ user_id: userId, product_id: productId });
    } else {
      await supabaseClient
        .from("favorites")
        .delete()
        .match({ user_id: userId, product_id: productId });
    }
  }
);

export default component$(() => {
  const userSig = useUser();
  const navigate = useNavigate();
  const productDetail = useSignal(useProductDetail().value);

  if (!productDetail.value.product) {
    return (
      <div>
        Sorry, looks like we don't have this product.
      </div>
    );
  }

  return (
    <div>
      <div class="...">
        <div>
          <h2 class="...">
            {productDetail.value.product.name}
          </h2>
          <div class="...">
            <div class="...">
              <span class="...">
                <div class="...">
                  <img
                    loading="eager"
                    width={400}
```

```
                    height={400}
                    class="..."
                    src={`/images/${productDetail.value.product.image}`}
                    alt={productDetail.value.product.name}
                  />
                </div>
              </span>
            </div>
            <div class="...">
              <div>
                <h3 class="...">Description</h3>
                <div
                  class="..."
                  dangerouslySetInnerHTML={
                    productDetail.value.product.description
                  }
                />
              </div>
              <div class="...">
                $ {productDetail.value.product.price}
                <div class="...">
                  {userSig.value ? (
                    <button
                      type="button"
                      class="..."
                      onClick$={() =>
                        console.log("Add to cart!")
                      }
                    >
                      <IconShoppingCart />
                      Add to cart
                    </button>
                  ) : (
                    <button
                      type="button"
                      class="..."
                      onClick$={() => navigate("/sign-in")}
                    >
                      Sign In
                    </button>
                  )}
                  <button type="button" class="...">
```

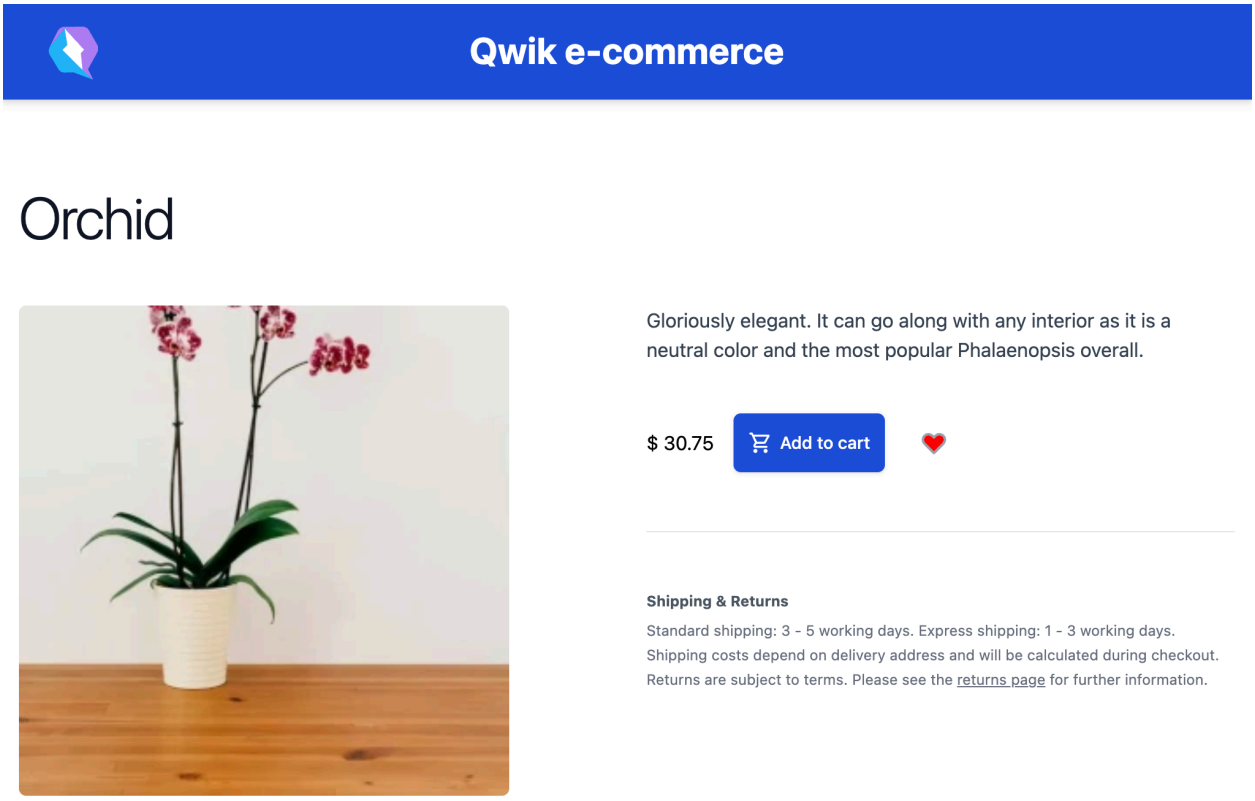
```
                    <HeartIcon
                      active={
                        productDetail.value.isFavorite
                      }
                      onClick$={async () => {
                        if (userSig.value) {
                          await changeFavorite(
                            userSig.value.id,
                            productDetail.value.product!.id,
                            !productDetail.value.isFavorite
                          );
                          productDetail.value = {
                            ...productDetail.value,
                            isFavorite:
                              !productDetail.value
                                .isFavorite,
                          };
                        }
                      }}
                    />
                    <span class="...">
                      Add to favorites
                    </span>
                  </button>
                </div>
              </div>
              [...]
            </div>
          </div>
        </div>
      </div>
    </div>
  );
});
```

So far so good, we made several changes to add this functionality, but let's start from the beginning. Let's start from the initial `routeLoader$`, we immediately read the detail of the product which has the slug that corresponds to the one present in our URL. If there is no product in our database we return `product` equal to null and `isFavorite` equal to false. Continuing we used the `resolveValue` parameter which we can invoked within `routeLoader$` to retrieve information relating to the user and to understand whether the user is logged in or not.

With `resolveValue` we will not re-execute the logic and code, but we will recover the value

previously calculated in the current request. In practice, for each request there is a cache system that allows you to read the data calculated in the case of nested requests, all this to have maximum optimization.

We continue with the analysis of the code and see that if the user is logged in we read in our Supabase table to understand if we have an associated record for this specific product and user. If a record exists then it means that our user has marked the product as favorite. We take all the information and use it as the return value of our loader. Here we see that compared to before we are returning an object composed of two parts, the product and the property that tells us whether it is preferred or not. Scrolling through the file we see that we have defined a `server$` function and therefore that it is executed on the server side in complete safety way. This function receives several parameters, the user ID, the product ID, and a boolean that tells us what operation we need to perform. If we have to mark the product as favorite we add a record to the `favorites` table otherwise, we will delete the record that has the indicated user ID and product ID. Let's then move to the changes we made to our component. We changed `productDetail` to make it a full-fledged signal by giving it a default value, the result of the logic just seen together. If we hadn't done this we would have only had a read-only signal but we also want to be able to go and modify it. Continuing with the changes we see that in the `onClick$` event of our heart icon, we have few lines with some logic. Here in these lines, we are saying: "If the user is logged in then we are going to call our server-side function to change our database and optimistically we are going to modify our `productDetail` signal to invert the value of `isFavorite`". By doing so we will be able to view our modification on the screen and we will have implemented the desired feature.



favorite

Another section that could be implemented would be the page that shows the favorite products. It will be enough to declare a new application route and then using a `routeLoader$`. We can go and read all the products of a specific user. This functionality contains all the concepts already addressed in the previous chapters and is therefore quite trivial, I challenge you to complete it before continuing to read the chapter.

Stripe

We have added many features to our application but we are missing the most important one, our core feature, the possibility to order and complete the payment. This type of process is very delicate because money, sensitive data regarding credit cards, and personal information are involved. This is why it is best to use an external service to avoid exposing our users to security problems. Stripe is a service that allows you to carry out financial operations and is one of the best on the market for simplicity and truly excellent documentation. It offers an SDK for almost all languages and of course also for Javascript. The first step to perform is to open an account on https://stripe.com and retrieve the API KEYS `Publishable key` and `Secret key` via the Dashboard https://dashboard.stripe.com/apikeys, these will be used to perform integration with the service.

FILE: `.env`

```
VITE_STRIPE_PUBLISHABLE_KEY=...
STRIPE_SECRET_KEY=...
```

We also need to install the JavaScript SDKs `@stripe/stripe-js` & `stripe` to be able to use the service APIs and proceed with the integration as best as possible.

```
pnpm install @stripe/stripe-js stripe
```

Add products to cart

In the previous chapter we got to the point of adding items to the cart, let's go through the logic it implements. Let's create a file that contains the types related to our global state.

FILE: `src/utils/store.ts`

```ts
export type Store = {
  cart: Cart;
};

type Cart = {
  products: CartProduct[];
};

export type Product = {
  id: number;
  name: string;
  description: string;
  price: string;
  image: string;
};

export type CartProduct = Product & {
  quantity: number;
};
```

Let's create a global state in our `layout` file

FILE: `src/routes/layout.tsx`

```
import { NavbarTop } from "~/components/NavbarTop";
import type { Store } from "~/utils/store";
import { supabaseClient } from "~/utils/supabase";

export const STORE_CONTEXT =
  createContextId < Store > "STORE_CONTEXT";

const initialData: Store = {
  cart: {
    products: [],
  },
};

export default component$(() => {
  const store =
    useStore <
    Store >
    (initialData,
    {
      deep: true,
    });
  useContextProvider(STORE_CONTEXT, store);

  return (
    <div>
      <NavbarTop />
      <Slot />
    </div>
  );
});
```

We created a global state with `useContextProvider` to be able to collect the items added to the cart and we also created a button to trigger the checkout process in our `NavbarTop` component.

FILE: `src/components/NavbarTop/index.tsx`

```
const store = useContext(STORE_CONTEXT);
const cartQuantitySig = useComputed$(() =>
  getCartQuantity(store.cart)
);

export const getCartQuantity = (cart: Store["cart"]) =>
  cart.products.reduce(
    (total, item) => total + item.quantity,
    0
  );
```

This is the part of logic that uses `useComputed$` through the `getCartQuantity` method to always keep the number of calculations present in our cart synchronized and the updated data will always be available through the `cartQuantitySig` variable.

Let's modify the logic present in our products on the homepage, but we must remember to modify the product detail page in the same way. The ideal would be to centralize these logics in a component to avoid repeating the code in multiple places. I also leave this activity to you as a possible extra task to increase your mastery.

FILE: `src/routes/index.tsx`

```
export default component$(() => {
  const store = useContext(STORE_CONTEXT);

  return (
    <button
      type="button"
      onClick$={() => {
        const cartProduct = [...store.cart.products].find(
          (p) => p.id === product.id
        );
        if (cartProduct) {
          cartProduct.quantity += 1;
          store.cart.products = [...store.cart.products];
        } else {
          store.cart.products = [
            ...store.cart.products,
            { ...product, quantity: 1 },
          ];
        }
      }}
    >
      Add to cart
    </button>
```

```
  );
});
```

With the following code we are modifying the global state of our application and by doing this all the components that use the global state will be kept updated. By doing so the counter seen previously will also be updated, showing us the number of correct products. Great we can see in real time that the item counter is increasing.

Stripe session

To initialize a Stripe session we must make a call with the `Stripe Secret key` and it must be used on the server side so we will create an API to execute this logic. This way, Stripe will create a beautiful and secure checkout page to redirect our users by entering their payment and shipping details. Once users place orders, they will be redirected to a landing page of our choosing, but we'll look at that later in this section.

FILE: `src/routes/api/process-payment/index.ts`

```
import Stripe from "stripe";
import type { CartProduct } from "~/utils/store";

let stripe: Stripe;

export const onPost: RequestHandler = async ({
  request,
  json,
  env,
}) => {
  const body = await request.json();
  if (!stripe) {
    stripe = new Stripe(env.get("STRIPE_SECRET_KEY") || "");
  }
  const stripeLineItems = body.products.map(
    (product: CartProduct) => ({
      price_data: {
        currency: "usd",
        product_data: {
          name: product.name,
          images: [
            `${import.meta.env.VITE_APP_URL}/images/${
              product.image
            }`,
          ],
```

```
        },
        unit_amount: parseFloat(product.price) * 100,
      },
      quantity: product.quantity,
    })
  );
  const session = await stripe.checkout.sessions.create({
    line_items: stripeLineItems,
    mode: "payment",
    success_url: `${
      import.meta.env.VITE_APP_URL
    }/order/success`,
    cancel_url: `${
      import.meta.env.VITE_APP_URL
    }/order/cancel`,
    shipping_address_collection: {
      allowed_countries: ["US"],
    },
    shipping_options: [
      {
        shipping_rate_data: {
          type: "fixed_amount",
          fixed_amount: { amount: 0, currency: "usd" },
          display_name: "Standard Shipping",
          delivery_estimate: {
            minimum: { unit: "business_day", value: 3 },
            maximum: { unit: "business_day", value: 5 },
          },
        },
      },
      {
        shipping_rate_data: {
          type: "fixed_amount",
          fixed_amount: { amount: 1500, currency: "usd" },
          display_name: "Express shipping",
          delivery_estimate: {
            minimum: { unit: "business_day", value: 1 },
            maximum: { unit: "business_day", value: 3 },
          },
        },
      },
    ],
```

```
  });
  json(200, { session });
};
```

With this file, we are exposing an API `/api/process-payment` that accepts a POST request with the items in the cart. Here we take these items and remap them to the structure that Stripe expects in the `line_items` object. We also set the `payment` mode and the two callback URLs for managing the case of success or error. Finally, in the `shipping_address_collection` variables we set the countries where we would like to ship to and with `shipping_options` shipping costs, we set a free one and a faster paid one. All these choice options can then be filled in by the user during the checkout phase. The API will then return the generated session which we will need to redirect to the Stripe page.

> `VITE_APP_URL` This environment variable is the URL of our site, it is used for images, but above all to compose the callback URLs: `success_url` and `cancel_url`.

Checkout process

In the `NavbarTop` component we created the icon with the counter of products added to the cart. By pressing on this icon we start the checkout procedure.

FILE: `src/components/CartIcon/index.tsx`

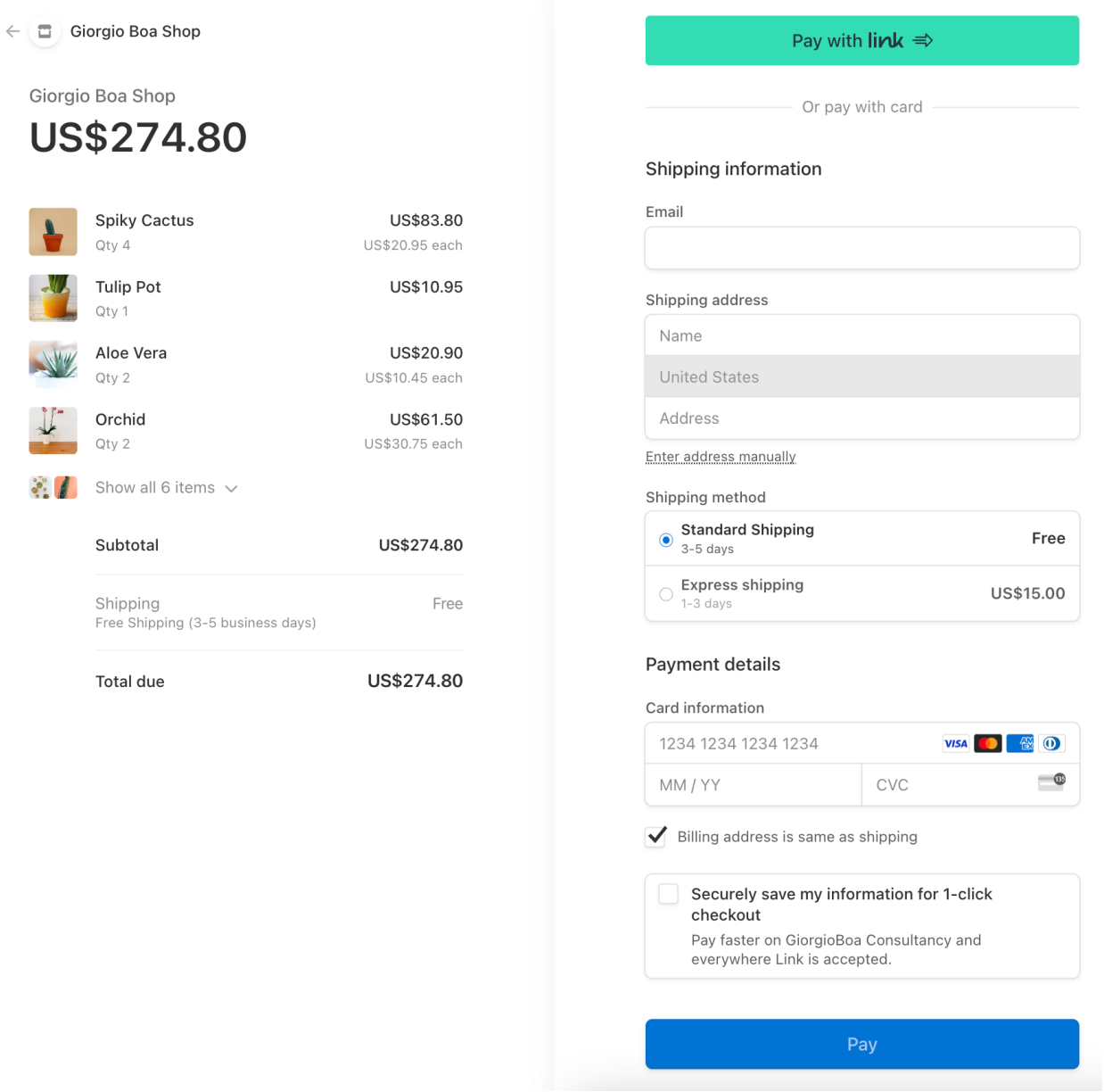
```
import { loadStripe } from "@stripe/stripe-js";
import { STORE_CONTEXT, useUser } from "~/routes/layout";

export const CartIcon = component$(() => {
  const userSig = useUser();
  const store = useContext(STORE_CONTEXT);
  const cartQuantitySig = useComputed$(() =>
    store.cart.products.reduce(
      (total, item) => total + item.quantity,
      0
    )
  );
  return (
    <nav class="...">
      <button
        class="..."
        data-testid="button"
        aria-label="cart icon"
        onClick$={async () => {
          if (userSig.value) {
            const response = await fetch(
```

```
              "/api/process-payment",
              {
                method: "POST",
                body: JSON.stringify({
                  products: store.cart.products,
                }),
              }
            );
            const { session } = await response.json();
            const key =
              import.meta.env.VITE_STRIPE_PUBLISHABLE_KEY ||
              "";
            const stripe = await loadStripe(key);
            if (stripe) {
              await stripe.redirectToCheckout({
                sessionId: session.id,
              });
            }
          }
        }}
      >
        {cartQuantitySig.value}
      </button>
    </nav>
  );
});
```

When we press the button we will call our newly created POST API `/api/process-payment` to which we will pass the items from our cart. We will take the newly created session and using `loadStripe` we will create the object that will allow us to execute the `redirectToCheckout`, we must pass the `session.id` to this function.

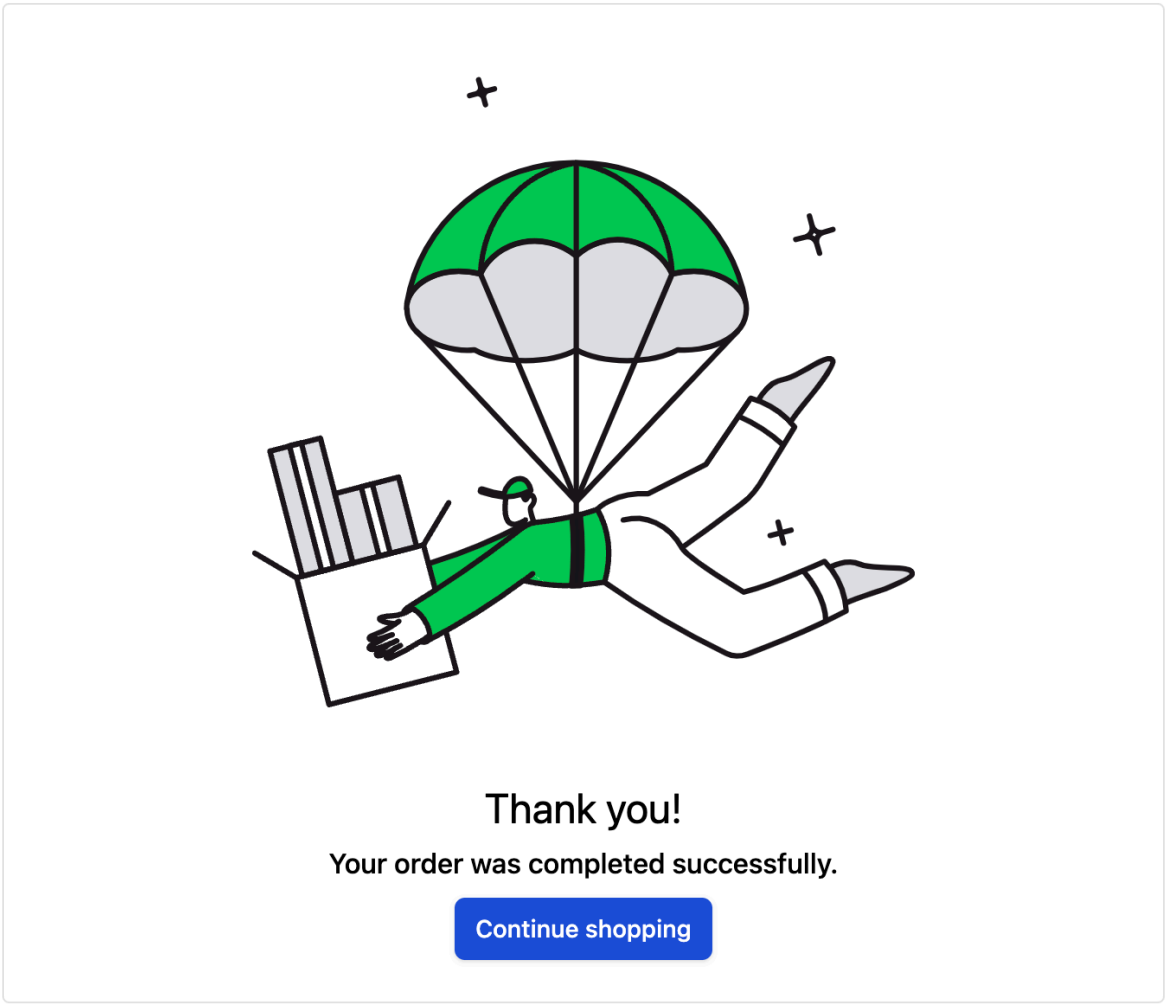
Let's test our checkout

We are finally ready to test our payment flow! We can add a couple of products to the cart, and then click on the icon to continue to checkout. We should find ourselves on this beautiful Stripe-based payment page, where we can enter our shipping information and choose the desired payment method.



stripe_checkout

In the Stripe dashboard you can change the name of the shop, in my case, it is Giorgio Boa Shop. Once the purchase is completed we will be redirected to the success order page.



success

It is also possible to configure Stripe so that when the payment is completed correctly we can receive a notification. You can enable this flow by configuring the Stripe webhook. It ensures that we process the payment confirmation securely. We need to set this configuration directly in the Stripe dashboard and then it is possible to expose an endpoint to receive this notification and perform operations following this. For example, we can think of saving the processed orders internally in our database to keep track of them, or another common scenario is to send notification emails to the users to thank them for the purchase made.

Page views tracking

Excellent, well done. Now another nice feature that we can add to our application is to be able to track how many times one of our pages is viewed and to do this we will take advantage of some advanced features of Supabase. First of all, however, let's create the table that will host our data. Let's create the file to write our SQL.

```
pnpm supabase migration new page_views
```

Let's write the SQL that will generate our table to contain the information we need.

```
create table public.page_views (
  id bigint generated by default as identity,
  page text not null,
  views bigint null default '1'::bigint,
  last_update timestamp without time zone not null default now(),
  constraint page_views_pkey primary key (id),
  constraint page_views_page_key unique (page)
) tablespace pg_default;

set check_function_bodies = off;

CREATE OR REPLACE FUNCTION public.increment_views(page_slug text)
 RETURNS void
 LANGUAGE plpgsql
AS $function$BEGIN
    IF EXISTS (SELECT FROM page_views WHERE page=page_slug) THEN
        UPDATE page_views
        SET views = views + 1,
            last_update = now()
        WHERE page = page_slug;
    ELSE
        INSERT into page_views(page) VALUES (page_slug);
    END IF;
END;$function$
;
```

Additionally, we can see that we have defined a Supabase database function that allows us to do several things and also add some logic to our procedure. We can see that this function accepts a `page_slug` parameter and then a conditional check is performed. If a record already exists for this specific page then we are going to increase the number of views for the specific page and we are also going to update the last update of the page. However, if the searched record is not present then one is created, the default value of `views` will be `1` and the current date will be inserted for the `last_update` field. Now we have everything configured to be able to track the views of our pages.

Now we can go and apply our changes to the database with the following command:

```
pnpm supabase db reset
```

Let's now move on to our Qwik application, and insert the logic to be able to call our function at the right time.

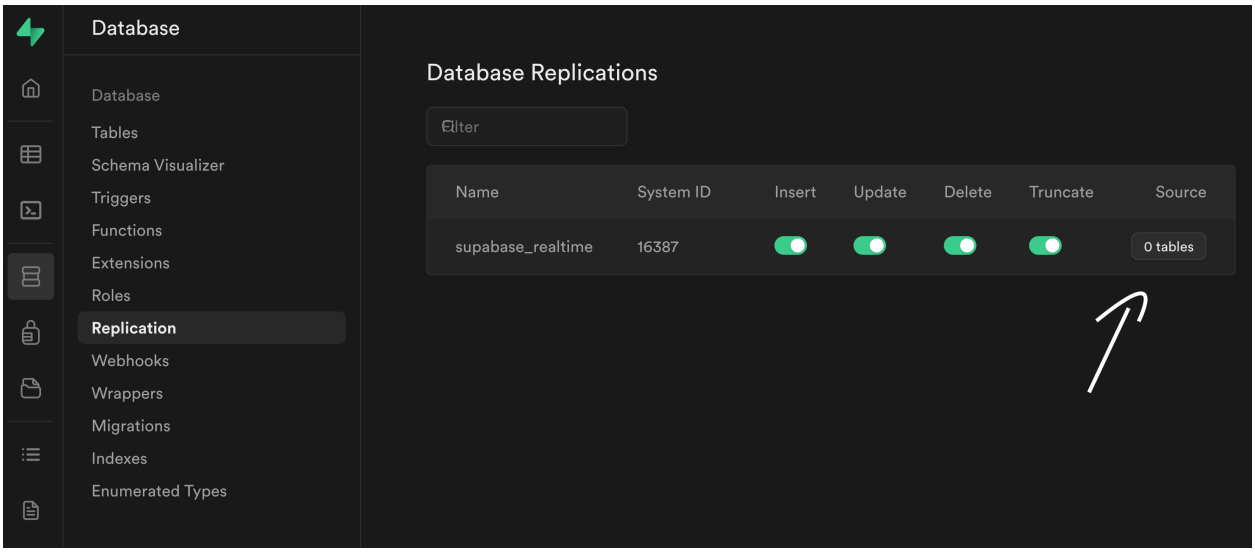
FILE: src/routes/detail/[slug]/index.tsx

```
import { supabaseClient } from "~/utils/supabase";

export const onGet: RequestHandler = async ({ params, next }) => {
  await supabaseClient.rpc('increment_views', { page_slug: params.slug });
  await next();
}

export const useProductDetail = routeLoader$(async ({ params, resolveValue }) => {
  [...]
});

export default component$(() => {
  [...]

  return (
    <div>
      [...]
    </div>
  );
});
```

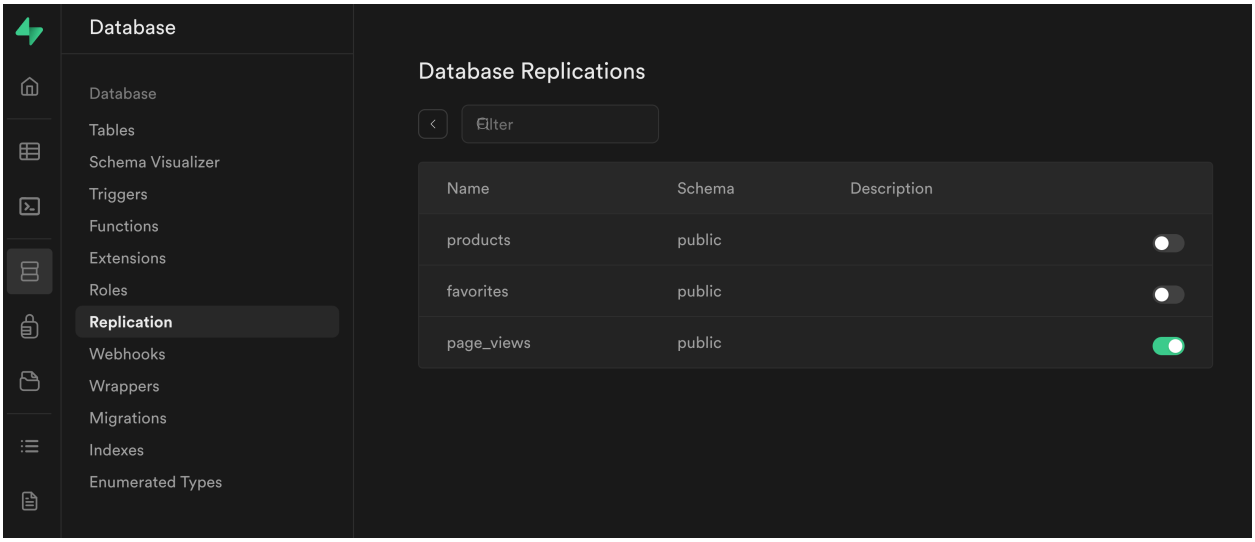
We have added an onGet middleware to the top of our src/routes/detail/[slug]/index.tsx page, which shows the item detail. This allows us to execute code when our page is called in GET and in fact when we go to visit a specific URL with our browser we are executing an HTTP GET operation. In this method, we are performing server-side operations, and specifically, we are going to call the database function that we defined in our database. We can see that we are also passing the parameter that is needed by the function to execute the logic correctly. We could have used the methods seen previously to write and read our data from the Supabase database, but I thought it was a great way to show you how to use database functions. If we think about it carefully, we are moving the logic to the database side because the computation and the logic are not inside our application, but are moved to the database. It is an interesting way to be able to remove logic from our frontend application although it is always better to make a very precise choice on how to manage these operations, either all of them in the frontend part or moving all the logic to the database. With this implementation, as soon as we load our page, our database is immediately updated. But usually, this is not what we want because we would be more interested in understanding whether a page has been viewed or not. Let's imagine we want to say: "My product has been seen and therefore I increase visits only if the end user has seen the image". With Qwik, performing this type of operation is very simple because we can create a component and then connect the backend logic. For example, when the user hovers over our product image or when a specific component is displayed. Thanks to the useOnDocument API we can listen to the mousemove event and check when the mouse position is above our component. We could also have used the useVisibleTask$ API to execute logic when our component is visible, but as explained in the previous chapters it is necessary to limit the eager execution of JavaScript as much as possible to keep our application fast and performing.

Real-time database

Supabase offers us a further very interesting functionality, we can create real-time applications because it provides us with a series of very simple methods to use. If we wanted to implement a chat or a social media it is clear that this functionality can be really useful, but it doesn't often happen that we have to implement such applications. I wanted to share a real application case that I had to develop. I implemented an interactive map that showed the movement of a specific vehicle in real-time. On the vehicle side, its position was sent every second, so I received the coordinates of the car and saved them in the database. On the frontend side, however, I had to show the position on the map and as the database changed, my application had to react accordingly by showing the new position to the end user. The result was very satisfactory because you could see the path of the vehicle, and I remember with pleasure that when we showed this functionality to customers, we always managed to obtain a "wow effect." We can do the same with Supabase because we have the real-time functionality available. To activate the function we must go to the Database > Replication menu and in this section click on "0 table." A new screen will open where we can activate real-time for our `page_views` table. By enabling this option we will be able to implement our new functionality. Together we will be able to show in real-time how many views a specific page has.



1_real_time



2_real_time

Ok, now let's write our frontend code to receive and display the data. Let's implement the functionality within the article detail page, but we could create a footer component and show the counter on all pages. We will need to add the middleware that saves the views on the pages we want to monitor.

FILE: `src/routes/detail/[slug]/index.tsx`

```
import type { PostgrestSingleResponse } from "@supabase/supabase-js";
import { HeartIcon } from "~/components/HeartIcon";
import { IconShoppingCart } from "~/components/IconShoppingCart";
import type { Product } from "~/routes";
import { STORE_CONTEXT, useUser } from "~/routes/layout";
import { supabaseClient } from "~/utils/supabase";

export const onGet: RequestHandler = async ({
  params,
  next,
}) => {
  await supabaseClient.rpc("increment_views", {
    page_slug: params.slug,
  });
  await next();
};

export const useProductDetail = routeLoader$(
  async ({ params, resolveValue }) => {
    const slug = params.slug;
    const { data }: PostgrestSingleResponse<Product[]> =
      await supabaseClient
```

```
        .from("products")
        .select("*")
        .eq("slug", slug);

    if (!data || data.length === 0) {
      return { product: null, isFavorite: false };
    }

    let isFavorite = false;
    const user = await resolveValue(useUser);
    if (user) {
      const favoritesResponse = await supabaseClient
        .from("favorites")
        .select("*")
        .match({
          user_id: user.id,
          product_id: data[0].id,
        });
      isFavorite =
        !!favoritesResponse.data &&
        favoritesResponse.data.length > 0;
    }
    return { product: data[0], isFavorite };
  }
);

export const useCurentViews = routeLoader$(
  async ({ params }) => {
    const { data } = await supabaseClient
      .from("page_views")
      .select("views")
      .eq("page", params.slug);
    return data && data[0] ? data[0].views : 1;
  }
);

export const changeFavorite = server$(
  async (
    userId: string,
    productId: number,
    isFavorite: boolean
  ) => {
```

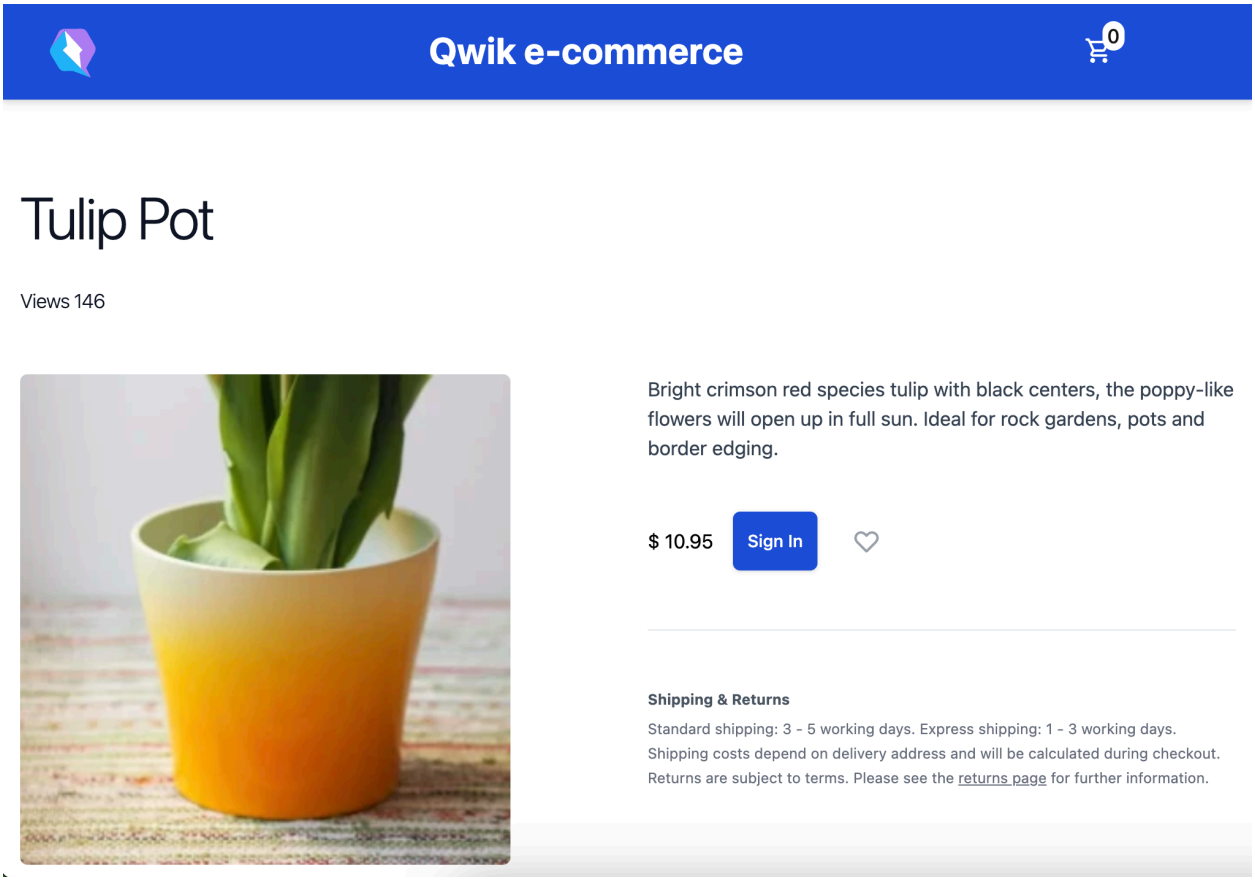
```
    if (isFavorite) {
      await supabaseClient
        .from("favorites")
        .insert({ user_id: userId, product_id: productId });
    } else {
      await supabaseClient
        .from("favorites")
        .delete()
        .match({ user_id: userId, product_id: productId });
    }
  }
);

export default component$(() => {
  const userSig = useUser();
  const viewsSig = useSignal(useCurentViews().value);
  const navigate = useNavigate();
  const location = useLocation();
  const store = useContext(STORE_CONTEXT);
  const productDetail = useSignal(useProductDetail().value);

  if (!productDetail.value.product) {
    return (
      <div>
        Sorry, looks like we don't have this product.
      </div>
    );
  }

  useVisibleTask$(({ cleanup }) => {
    const sub = supabaseClient
      .channel("custom-all-channel")
      .on(
        "postgres_changes",
        {
          event: "*",
          schema: "public",
          table: "page_views",
        },
        async (payload) => {
          if (
            payload.eventType === "UPDATE" &&
```

```
            payload.new.page === location.params.slug
          ) {
            const newViews = payload.new.views;
            viewsSig.value = newViews;
          }
        }
      )
      .subscribe();

    cleanup(() => {
      sub.unsubscribe();
    });
  });

  return (
    <div>
      <div class="...">
        <div>
          <h2 class="...">
            {productDetail.value.product.name}
          </h2>
          <span class="...">
            Views {viewsSig.value.toString()}
          </span>
          <div class="...">
            <div class="...">
              <span class="...">
                <div class="...">
                  <img
                    loading="eager"
                    width={400}
                    height={400}
                    class="..."
                    src={`/images/${productDetail.value.product.image}`}
                    alt={productDetail.value.product.name}
                  />
                </div>
              </span>
            </div>
            <div class="...">
              <div class="...">
                <h3 class="...">Description</h3>
```

```
149                   <div
150                     class="..."
151                     dangerouslySetInnerHTML={
152                       productDetail.value.product.description
153                     }
154                   />
155                 </div>
156                 <div class="...">
157                   $ {productDetail.value.product.price}
158                   <div class="...">
159                     {userSig.value ? (
160                       <button
161                         type="button"
162                         class="..."
163                         onClick$={() => {
164                           const cartProduct = [
165                             ...store.cart.products,
166                           ].find(
167                             (p) =>
168                               p.id ===
169                               productDetail.value.product!.id
170                           );
171                           if (cartProduct) {
172                             cartProduct.quantity += 1;
173                             store.cart.products = [
174                               ...store.cart.products,
175                             ];
176                           } else {
177                             store.cart.products = [
178                               ...store.cart.products,
179                               {
180                                 ...productDetail.value
181                                   .product!,
182                                 quantity: 1,
183                               },
184                             ];
185                           }
186                         }}
187                       >
188                         <IconShoppingCart />
189                         Add to cart
190                       </button>
```

```
                ) : (
                  <button
                    type="button"
                    class="..."
                    onClick$={() => navigate("/sign-in")}
                  >
                    Sign In
                  </button>
                )}
                <button type="button" class="...">
                  <HeartIcon
                    active={
                      productDetail.value.isFavorite
                    }
                    onClick$={async () => {
                      if (userSig.value) {
                        await changeFavorite(
                          userSig.value.id,
                          productDetail.value.product!.id,
                          !productDetail.value.isFavorite
                        );
                        productDetail.value = {
                          ...productDetail.value,
                          isFavorite:
                            !productDetail.value
                              .isFavorite,
                        };
                      }
                    }}
                  />
                  <span class="...">
                    Add to favorites
                  </span>
                </button>
              </div>
            </div>

            <section class="...">
              <h3 class="...">Shipping & Returns</h3>
              <div class="...">
                <p>
                  Standard shipping: 3 - 5 working days.
```

```
                    Express shipping: 1 - 3 working days.
                  </p>
                  <p>
                    Shipping costs depend on delivery
                    address and will be calculated during
                    checkout.
                  </p>
                  <p>
                    Returns are subject to terms. Please see
                    the
                    <span class="...">
                      returns page
                    </span> for further information.
                  </p>
                </div>
              </section>
            </div>
          </div>
        </div>
      </div>
    );
});
```



detail_views

Great, here's our finished detail page. We have implemented the functionality to display in real-time how many views the page has, but this is a good time to make a summary of the entire file because there is a lot of logic implemented. At the beginning of the file we have an `onGet` middleware that allows us to update the views of our page in the database, every time the page is rendered the counter in our database is also updated. Here we are invoking a database function that behind the scenes will execute the data update or insertion logic. We then move on to the first `routeLoader$` present, here we read the product detail from the `products` table, and then we check whether the user is logged in or not thanks to the `resolveValue` method that we saw previously. If the user is logged in we check whether the article is among the user's favorites or not. We then pass all this information to the component to display it. Continuing we have another `routeLoader$` which allows us to read how many views the page we are viewing has because we are going to execute a query in our database, this will help us to have the initial value to show to the user. Then we find a `server$` function that allows us to perform the update on our database to save whether the article is favorited or not for the current user. Below we have our component, if we don't find the product we display a fallback message for the user otherwise we show the details of our product. We have several signals, some of which receive default values that have been calculated on the server side. We also have `useNavigate` which allows us to redirect the user in case they need to Sign In and we have `useLocation` which allows us to read the slug from the page URL. To handle the update of the `viewsSig` signal with the updated hit counter we used a `useVisibleTask$` to run the code in an `eagerly` when the component becomes visible in the viewport. The Qwik team wants to make sure that we are using `useVisibleTask$` with awareness and therefore decided to

add a warning if we use it. You can disable it globally by changing the eslint configuration or with a specific eslint comment. Inside the `useVisibleTask$` we have inserted our logic to react to changes made to the database. We subscribe to changing events in our database and react accordingly. We see that if the event is of type `UPDATE` and the change concerns the page we are viewing then we update the value of `viewsSig` to show the new data. We also included `cleanup` at the end of the function, which is a very useful method because it allows us to remove the subscription to database changes when it's not necessary anymore. We can think of `cleanup` as a method that is called when our component is removed from the viewport or when a new task is triggered. It is not invoked when the task is completed. In the rest of the components we have the details of our product and then every time we add the product to the cart we update the application state to show the update of the counter of the product we want to purchase. As fully explained in the previous sections, the last peculiarity is that if we are not logged in we are asked to sign in.

Summary

In this chapter, we added other features to our e-commerce that make it a truly usable application in production. It will then be our task to carry out a deployment in production. In the previous chapters, however, we have seen that it is very simple to deploy our Qwik application because there are many integrations to perform the deployment and specifically we have analyzed the deployment with Vercel. We added the ability to mark products as favorites and being able to see the list of favorite items would be a nice feature to implement as a homework assignment. We have implemented a global state to keep track of our cart and therefore with each added product we have the updated situation. We have created a cart button that will allow us to trigger the stripe flow and redirect the user to the checkout on the Stripe platform. In this way, we have freed ourselves from payment management and everything happens safely in the Stripe environment; the user then, following the procedure, will be redirected to the route of our application both in the event of success and in the event of an error. Together we went to see how easy it is with Supabase to create intelligent functions that allow us to execute more or less complex logic. Thanks to this function we have created a view counter for our pages. Then we saw the real-time database feature which is a really powerful feature and allowed us to view page views in real-time. In an e-commerce it might make sense to show how many active users are seeing a certain product, this can increase our earnings by bringing more orders. Our users may feel they have a unique opportunity to purchase the product they are viewing. It is necessary to carry out tests to validate and understand whether the functionality brings the advantages we have just indicated. There are many features to implement, starting with the possibility of eliminating or modifying the quantities of an item or creating a user area where you can view order history or change your credentials. I'll leave these ideas to you as homework, try implementing some functionality before moving on to the next chapter. Speaking of the most common best practices in writing our applications, we cannot fail to mention the fact of trying to keep the code as clean as possible and assigning meaningful names to the variables to avoid extra cognitive efforts to understand the code being written. It often happens that over time even the same person who wrote the code struggles to immediately understand the application logic, so pay attention to the names of the variables and more. Nowadays there are tools to perform automatic linting of our code and I must say that it is quite a convenience. They allow us to standardize the written code, avoiding going crazy from the eternal dilemma… spaces versus tabs or more code styling things. Which one do you like the most? Honestly, you just need to reach an agreement within the team and avoid returning to the issue every code review. Another aspect to be considered as absolute best practice is the writing of tests. Dedicating time to writing tests is certainly one of the most valuable practices. We will see together how to test our Qwik application to ensure you have a rock-solid solution.

Adding tests to our Qwik application

Overview of application testing

How many times have we written a new piece of code and, due to a side effect, ended up breaking another feature we weren't expecting? These types of problems are commonly called regression errors. It often happens that if our code is strongly coupled and shares business logic, changing the logic connected to it also affects other parts of the application. How can we be sure that our changes or evolutions do not create problems in production? A good test suite is able to prevent this kind of problems. Another benefit that should not be underestimated regarding testing is certainly that of being able to detect errors in advance directly during the development phase. So even before integrating the code into your final bundle, you may realize that something doesn't work as it should. This guarantees you save time and greatly reduces errors that can occur in production. In fact, if a problem arises a few weeks after our code has reached production, it will be more complex to identify and solve the problem. Having a good test suite also allows us to facilitate refactoring activities, i.e., the process of fixing existing code to make it more manageable, and scalable but not only. Refactoring is an essential part of software development that allows us to evolve and improve the code of our application and reduce technical debt. Tests act as checkers that allow us to check whether the refactoring of our code has introduced errors or created regressions. Having clean code in the long term allows you to be more productive and therefore save money and time. The most important advantage, however, is the knowledge that we are following the implementation of high-quality code. Whether we are working on a small application, or even more so on a large one, we are aware that what we have tested works correctly, and therefore we have not changed the behavior which until now has allowed us to have an application that works correctly. To write a good test suite, however, it is important to dedicate effort and resources to be able to derive the benefits we have mentioned. Precisely because it takes time, many companies that I have had the opportunity to meet in my career have never fully believed in testing, a practice that was born many years ago and has evolved over time. With the advent of software, we have had an explosion of innovation and nowadays there are many programs to perform tests, both manual and automatic. Automatic ones were created precisely to limit the time investment that was mentioned before, and to speed up all those repetitive tasks that do not require a person to carry them out. However, various types of tests touch on different areas of our application, because they are focused on different scopes. Let's look at this distinction together.

Unit tests

Unit tests are very important because they focus on the logic part of our application. They allow us to write robust and reliable code. This type of test focuses on a single functionality, and therefore we can only focus on a few lines of code. The functions that are tested are pure functions; i.e., they do not have side effects within them. The most classic example is testing the sum function, given a certain input (two numbers) I provide an output. Let's imagine inserting a logic within the function according to which, on Monday, the sum works differently compared to the other days of the week. In this case, we are adding unpredictability within our function and making our code very difficult to test. Pure functions give us a big advantage in terms of stability and predictability of our code because we can always have the situation under control.

We know with certainty that our business algorithms do not change. If we have discount policies that are based on the total amount of the goods our user is purchasing, we can verify the logic with a specific test and verify that, following the evolutions of our software, our algorithm is not affected. This algorithm will be covered with various tests that verify the logic with different amounts and will be put under stress with edge cases that perhaps will bring out some cases not considered. A striking case could be that of having an amount which, net of promotions, entitles you to 20% of the total. Does this rule also apply if my user has already applied a 50% promotional discount for the spring sales? Which of the two discounts should I keep? If we think about it carefully, when we go to design an algorithm, these are questions that we already ask ourselves, but with the help of tests it is easier for us to define them. We have the possibility of isolating the scope leaving out the rest of the application to concentrate on the single piece of code. This approach therefore also helps us to better design our applications by focusing on real cases and avoiding predicting edge cases upfront. I know you are thinking that these are logical problems most of the time, and usually they are in the back-end. So, I'll give you another example, which is purely frontend. Let's take the product detail page seen before in our application. A unit test that can be developed is to verify that if I am inside the detail of a product, the name of the product I requested is present in the viewport. This guarantees us that our page is correct because by checking that all the information is in the right place, we can make the code that renders the detail page robust. The low cost in terms of effort and their simplicity make these types of tests an essential aid to our application development and the future evolutions we will make. This type of test also implicitly creates a sort of documentation of how the application should work and makes clear a whole series of cases that the application takes into consideration. By looking at a correctly written test suite we will be able to quickly understand the logic behind a certain piece of code, this aspect is very powerful. Another factor to consider is the fact of structuring our code with functions that are as atomic as possible to make any type of operation trivial. There should be no surprises in the code. This is a good practice not only for testing, but programming in general. If you have never read Clean Code: A Handbook of Agile Software Craftsmanship[45] by Robert C. Martin, I recommend you do so because you will find many interesting ideas on best practices for writing quality code.

Visual regression tests

As we can understand from the name, Visual regression tests are tests where a particular aspect of our frontend application is examined. This type of testing is based on user interface comparison and specifically compares two versions to ensure that there are no unwanted changes. As mentioned, two versions of our interface are compared, the current version with our changes and a previous version that has been appropriately saved. Thanks to specific tools it is possible to take a "photograph" of the current version which will then be compared with the one to be published. This type of testing helps ensure graphical consistency across different devices, operating systems, and browsers. When we go to make changes to our application it is very expensive in terms of time to test on all these combinations, and it is not even possible to have all the supported devices available to check that everything is as desired. Furthermore, it is very complex to identify the graphic differences between two interfaces. Have you ever played "Spot the Difference"? Well, it's a very similar experience to the one you can have when playing this game. However, when we have to perform a release it is anything but a game because the pressure to solve the problem is high. Automated solutions, on the other hand, make this verification cycle smooth which can save you from bugs in production. These tools are very precise because they analytically check pixel by pixel to find defects. Making a catastrophic hypothesis we can think of the

[45] https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882

fact that some buttons have become too small to be tapped from mobile, and therefore, our mobile users will not be able to use our application.

Integration tests

Integration tests are particular types of tests that focus on the iteration between two or more parts of the application, and are slightly more expensive than unit tests because more parts of the application need to be coordinated together. If we were to explain the integration tests with a simple example we could take the case study of sending emails following registration. The user registers, and we send a greeting email. This integration test, though, is very backend-oriented, but it's easy to think of this flow. As done previously, let's try to see these tests from the frontend side, things are slightly different. We can imagine having a login form for our application, a good integration test could be to test if we can log in to the application. So thanks to the tools that we will later exam, we can go and render our login page, we can fill in the fields automatically, and then go and log in within the application. All this involves various integration steps, validation of the forms to verify that we have entered current data, and when we press the submit button we also verify that the call is executed. Once we receive the response we will also proceed to the application homepage. Often, the login calls, or more generally, are simulated so as not to include backend management in these tests. When we develop, we don't have both up-and-running environments on our local machine, and logging into staging or production environments is not the wisest thing. After all, we want to run tests and not create confusion. As long as we talk about login it can be ok, but when we have to try to close an order or create a user it is not something to do. Writing this type of test requires time, but the advantages are many. First of all, we are going to add value and security to our application because we are going to guarantee that the specific functionality continues to work during releases. Another aspect that should not be underestimated is that in this case too we are implicitly documenting how the flows of our application must work, and how the systems must interact. These tests are very useful and help us lay the foundation for end-to-end testing.

End-to-end tests

These types of tests were born with the idea of being able to control the flow of our application from start to the end. These types of controls are much more expensive to maintain in the long run because our interface continually changes with the implementation of new features. An example of an end-to-end flow could be to start by logging into the application, continue with the test by entering the order list, and finish with the download and verification of the order selected by us. Already from this description, we can see that it is truly an all-round test because we are testing the entire flow. If we think about it, we are putting together many integration tests to test a more complex one. Writing these types of tests is very burdensome and some tools have implemented a user action recording functionality for some time now. Let me explain better. In practice, these tools allow us to record all the actions we do in our application and then convert them into code that can be used to reproduce the sequence whenever we want. This approach allows you to significantly reduce the writing times of these tests and partially solve one of the weak points of end-to-end tests. The other problem present is that of maintaining the tests in the long term because it is easy for the position of our components to change, the same can be said for the styles and much more. Various conditions certify whether a test is correct or not, but the most used are based on the presence of DOM elements or CSS classes, this makes end-to-end tests very fragile indeed. On the other hand, they give us a complete picture of the flow that the user can complete from start to finish and manage to find problems that are not evident with other types of

tests. If we don't already have a visual regression test suite, an end-to-end flow for certain browsers might have compatibility problems. Perhaps features that are not cross-browser compatible have been used, and therefore, automatically testing the flows is a really good thing to give a positive user experience regardless of the browser they use. We can simulate access to the platform by different users, with different roles and with different views. In short, if you have time to expand these tests, there is a lot to do. In my experience, what I have seen done most is to concentrate most of the effort on testing the core flow and then possibly covering the less priority flows. It's an excellent way not to give up these very important tests, but also calibrate the effort they unfortunately require.

Accessibility tests

Accessibility in modern applications has become fundamental and is also required at a legislative level, so we cannot hide and pretend it doesn't exist. The basic principle is to be able to allow everyone to use the application regardless of their ability or disability. There are various ways to check if our application is accessible, the most common is to test our application using screen readers or other assistive technologies, but it is possible to run tests with users with different disabilities and verify compliance with accessibility standards. Some automatic tools analyze and interact with our application to verify that it complies with W3C standards[46] to guarantee equal opportunities for everyone.

How much testing is enough? When do I write tests?

Being able to understand when and what to test is very important to optimize our efforts and maximize profits. The famous graph `3X Explore/Expand/Extract` created by Kent Beck[47] can be used to understand in which stage our application is. At each stage, it is necessary to behave differently to avoid unnecessary waste. Let's analyze the three phases in detail:

- **explore**: we are launching our idea on the market, and we don't know if it will bring benefits to our business, it makes no sense to invest in testing our features because things could change quickly. At this stage, however, it makes more sense to invest in validating the business idea and refining the UI/UX by interviewing stakeholders.
- **expand**: we already have a user base and we need to scale to increase our business, we need to balance manual and automatic testing without losing focus on the market and develop new features to attract additional users. Often in this phase we are always looking to improve and optimize from many points of view, both at the code level, but also at the level of services and way of presenting ourselves to new customers. This type of approach is aimed at attracting as many customers as possible and trying to take away the market from competitors. Most applications are in this phase. You experiment and try to consolidate at the same time, it's a difficult balance to find, but precisely because it proceeds in waves it's nice to be able to experience.
- **extract**: our position in the market is established and acquiring new users becomes increasingly complex because we have already reached most of them. Dedicating ourselves to automatic testing is the optimal solution because we need to consolidate our processes without creating bugs in production. Introducing

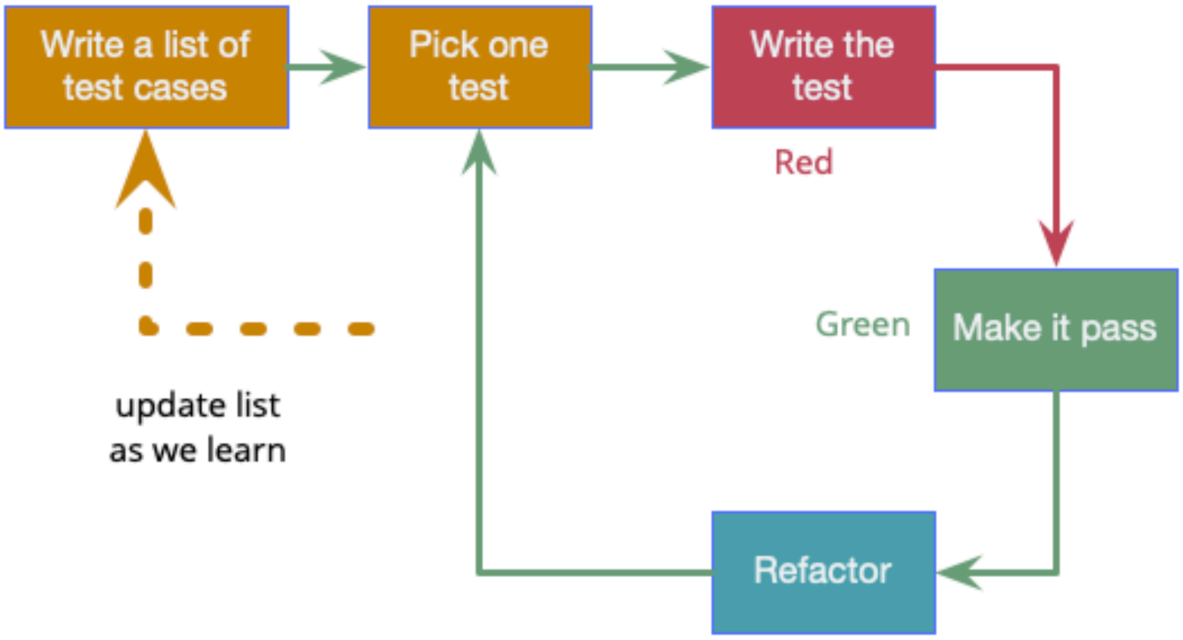
[46] https://www.w3.org/WAI/standards-guidelines/
[47] https://medium.com/@kentbeck_7670/the-product-development-triathlon-6464e2763c46

> new features step by step will make our users even more satisfied. So we need to think carefully about which tests to write and if and when it makes sense to test our code, but for sure the main features of our application are the ones that will be tested first. If our application is an e-commerce, it certainly makes more sense to protect against bugs in the order closing process rather than making sure that the newsletter subscription works. So, since writing and maintaining tests costs valuable time in the long run, we think carefully about what and how to test our application.

There is an interesting analogy with triathlon, which lends itself well to better focusing on what has been said so far. A triathlon race is made up of different phases: a first swimming phase, a second cycling phase, and a third running phase. These phases have different distances, and it is important to tackle them in the right way to reach the end of a race of this difficulty. During the race, we must be very focused on the discipline we are facing because if we tackle the swimming phase with the bike, we will certainly have problems. We can apply the same thing to testing. If, during the expansion phase, we dedicate ourselves to testing instead of interviewing users and understanding what can distinguish us from other competitors, our business will never be able to emerge, and we will fail before we even realize it. The same thing can be thought of for the other phases seen previously. Therefore it is very important to stop and reflect if we have the right attitude to maximize profits.

Test-Driven Development (TDD)

We have seen what tests are for and what types exist, but we have not yet said how to put them into practice with dedication. TDD is a software development method that aims to be lean, and where tests are written before the actual code. The idea is that before even writing the application code for a solution, we think about the first possible cases and variants of the functionality that we need to implement. This approach ensures alignment between the requirements and the code we are going to write for our application. I deliberately said “first possible case studies” because with this methodology, the requirements are transformed into test cases step by step. The problem is divided into small parts and each requirement is introduced one by one, thus avoiding writing superfluous code upfront. We start by writing the first test, therefore the first requirement, and we make it fail, after all, we haven’t written any logic yet, but only the test. How could this work? At this point, the developer writes the minimum code necessary to pass the test. Once the test(s) have passed, we continue with a refactoring phase, which must not break the tests but continue to produce valid code. Once the duplications have been removed, and the code has been improved and aligned with the project standards, we can continue by inserting another requirement and then write a new test starting again with the cycle just described. Requirement after requirement we develop our code. Once we understand the mechanism, using this methodology, we would reduce the development time and increase the quality of our code because in each cycle there is the possibility of refactoring our code. If not adopted in this way, refactoring risks is one of those activities that are difficult to include in everyday activities because there is always something more important to do. With TDD, it is a specific phase of the methodology, and therefore, it is very difficult not to be able to dedicate time to this refactoring activity.



TDD

Image from https://martinfowler.com/bliki/TestDrivenDevelopment.html

To get the most out of this methodology, you need to focus on one feature at a time because it is an iterative method; it doesn't pretend to solve everything straight away. We need to start with a test that fails because this means that something is missing in our code. When, on the other hand, we see the green test, it means that we are continuing in the right direction. Don't write the code right away otherwise the tests, even if inserted later, will not help you design well-written logic. It is also good practice to write down good names for our tests because when they fail the names can help us determine what is wrong. Another golden rule that I can share with you is not to create tests that depend on each other because we cannot execute them in parallel, but only in precise order. This greatly slows down the flow of execution of the tests.

Let's add some tests with Cypress

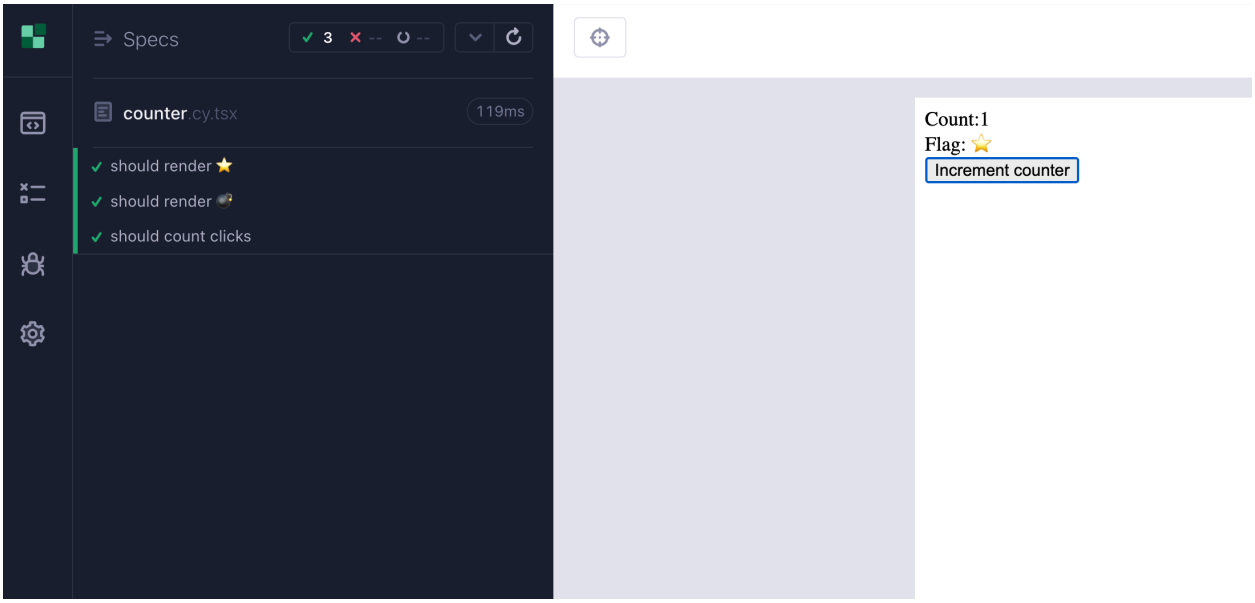
There are many tools for running tests with Qwik and via the Qwik CLI, you can integrate them into your project very quickly. I would like to see together how easy it is to use `Cypress` to run e2e tests of our application. To add Cypress to Qwik, simply use the

```
pnpm run qwik add cypress
```

command, and the Qwik CLI will add and install the dependencies for us and create the configuration files necessary for its operation. This command, in addition to all the files necessary for the configuration, will also add two scripts in our `package.json`

```
"cypress.open": "cypress open --component",
"cypress.run": "cypress run --component",
```

Both commands launch Cypress to run the tests, with open the tool will open to show us the tests graphically in the browser while with run the same process will be executed but in headless mode. This allows us to run tests even in environments such as pipelines where the graphical interface is not present. If instead, we decide to launch the graphical interface we will also be able to navigate our tests, check the results, and re-run them at will.



cypress

Among the various files created for us by the Qwik integration, we also have this first example base file. We have a component that renders a counter and maintains an internal state using the useStore API.

```
export const Counter = component$(
  (props: { flag: boolean }) => {
    const state = useStore({ counter: 0 });
    return (
      <>
        <span>Count:{state.counter}</span>
        <div class="icon">
          Flag: {props.flag ? "" : ""}
        </div>
        <button
          class="btn-counter"
          onClick$={() => state.counter++}
        >
          Increment counter
```

```
        </button>
      </>
    );
  }
);
```

Here is the file relating to the tests.

```
it("should render □", () => {
  cy.mount(<Counter flag={true} />);
  cy.get("div.icon").should("contain.text", "□");
});

it("should render □", () => {
  cy.mount(<Counter flag={false} />);
  cy.get("div.icon").should("contain.text", "□");
});

it("should count clicks", () => {
  cy.mount(<Counter flag={true} />);
  cy.get("span").should("contain.text", "Count:0");
  cy.get("button").click();
  cy.get("span").should("contain.text", "Count:1");
});
```

Here we have some tests, in the first and second by varying the `flag` property we are going to verify that the `div` with the `icon` class contains the text that we have decided to show. The third test instead performs a more complex check, clicks on the button, and verifies that the displayed data containing `state.counter` is as we expect. With this test, we can go and do the first refactor without fear of creating bugs because Cypress ran the test, and our component is correct. Let's replace `useStore` with `useSignal` because in any case we only need to keep one value and `useSignal` is the API that best suits this need.

```
export const Counter = component$(
  (props: { flag: boolean }) => {
    const counterSig = useSignal(0);
    return (
      <>
        <span>Count:{counterSig.value}</span>
        <div class="icon">
          Flag: {props.flag ? "□" : "□"}
        </div>
        <button
```

```
            class="btn-counter"
            onClick$={() => counterSig.value++}
          >
            Increment counter
          </button>
        </>
      );
    }
);
```

So as per Test-Driven Development practice, we went to improve our component code to use `useSignal`. By launching the test suite we see that our test completes successfully and therefore we can say that we have completed our TDD cycle. If we have other requirements to include in the tests, now is the right time because our code passes the tests correctly, and we have also performed our refactoring, which has allowed us to increase the quality of our code. Well done!

This is a first example which you can then complicate as you wish. This is a test directly on the component and not an end-to-end test that touches the entire application flow. In fact, with Cypress, we can completely control our web page. We can simulate user events, here are some examples: `click`, `dblclick`, `rightclick`, `scrollTo`, `select`, `type`. It is very clear what these events simulate, and it is possible to integrate them very quickly into our tests.

```
cy.get("span").should("contain.text", "Count:0");
cy.get("button").click();
```

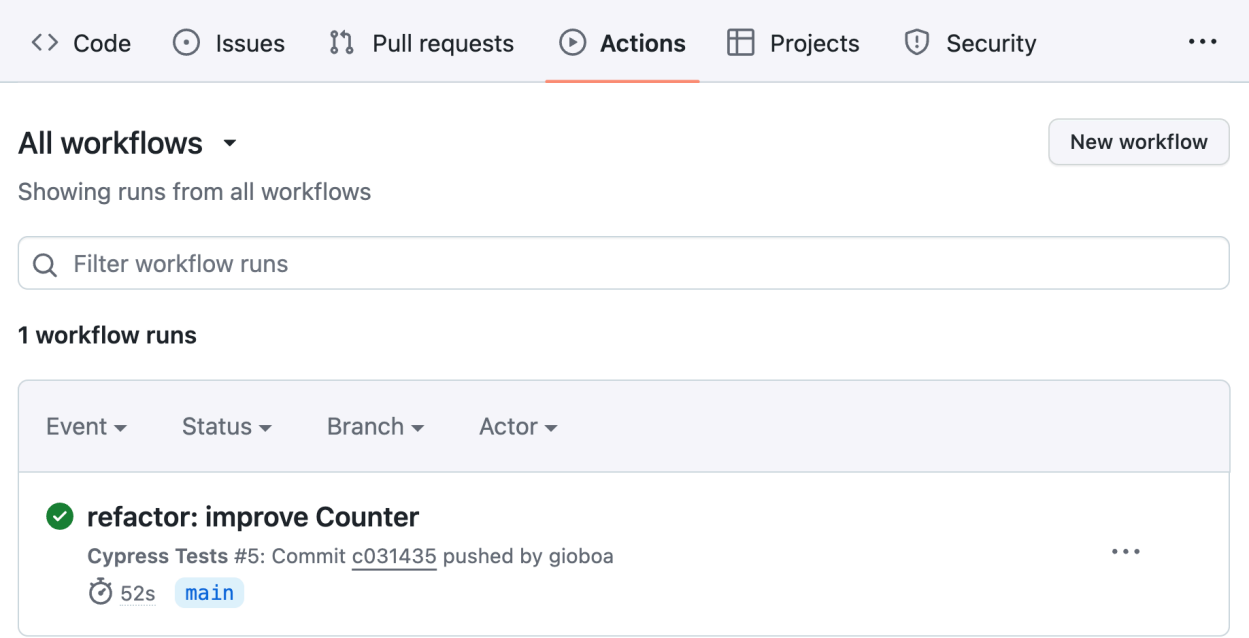
Analyzing the previous example we had already seen the `click` event. So once we have managed to recover the element we can simulate a certain event in a truly natural way. It is also possible to control the browser by changing the history to move around the URL, but not only that. It is possible to access cookies to read, modify, and delete them. An example is `getAllCookies` which allows us to read cookies. We can also command the browser's `focus` to move the cursor on input, for example, and with the `type` event we can fill in the fields of a form. In short, we have full control to interact in depth with the browser and therefore writing our tests is quite easy. With Cypress, it is then possible to record the tests instead of writing them, and as mentioned before this allows us to reduce the time for creating and maintaining our tests.

Add Cypress to our GitHub actions

As we have seen in the previous chapters, with GitHub Action we can perform automatic tasks to guarantee quality in our way of working. To configure our GitHub repository to run the Cypress task on each commit, simply go to this page `https://github.com/<your_github_user>/<your_repo>/actions` and create a new task with this code.

```yaml
name: Cypress Tests

on: push

jobs:
  cypress-run:
    runs-on: ubuntu-22.04
    steps:
      - name: Checkout
        uses: actions/checkout@v4
      # Install NPM dependencies, cache them correctly
      # and run all Cypress tests
      - name: Cypress run
        uses: cypress-io/github-action@v6
        with:
          component: true
          build: npm run build
          start: npm start
```

As we can see, GitHub Action has the advantage of simplicity because in just a few minutes you can create something truly valid and advanced. This task installs the dependencies and through the action provided by Cypress `cypress-io/github-action` we can execute the task to verify our commits. Note that here we are running a specific test for a component, and not for a more complete flow we are using the `component: true` flag. Here is the check for our refactor commit. This flow allows us to verify that our code is always correct and allows us to prevent many bugs from reaching production. The configuration can be as complicated as desired to take into consideration many other variables. This is the basis for automation that nowadays is truly within everyone's reach, and there is no longer any reason not to consider it a necessary practice in everyday development.



GitHub_action

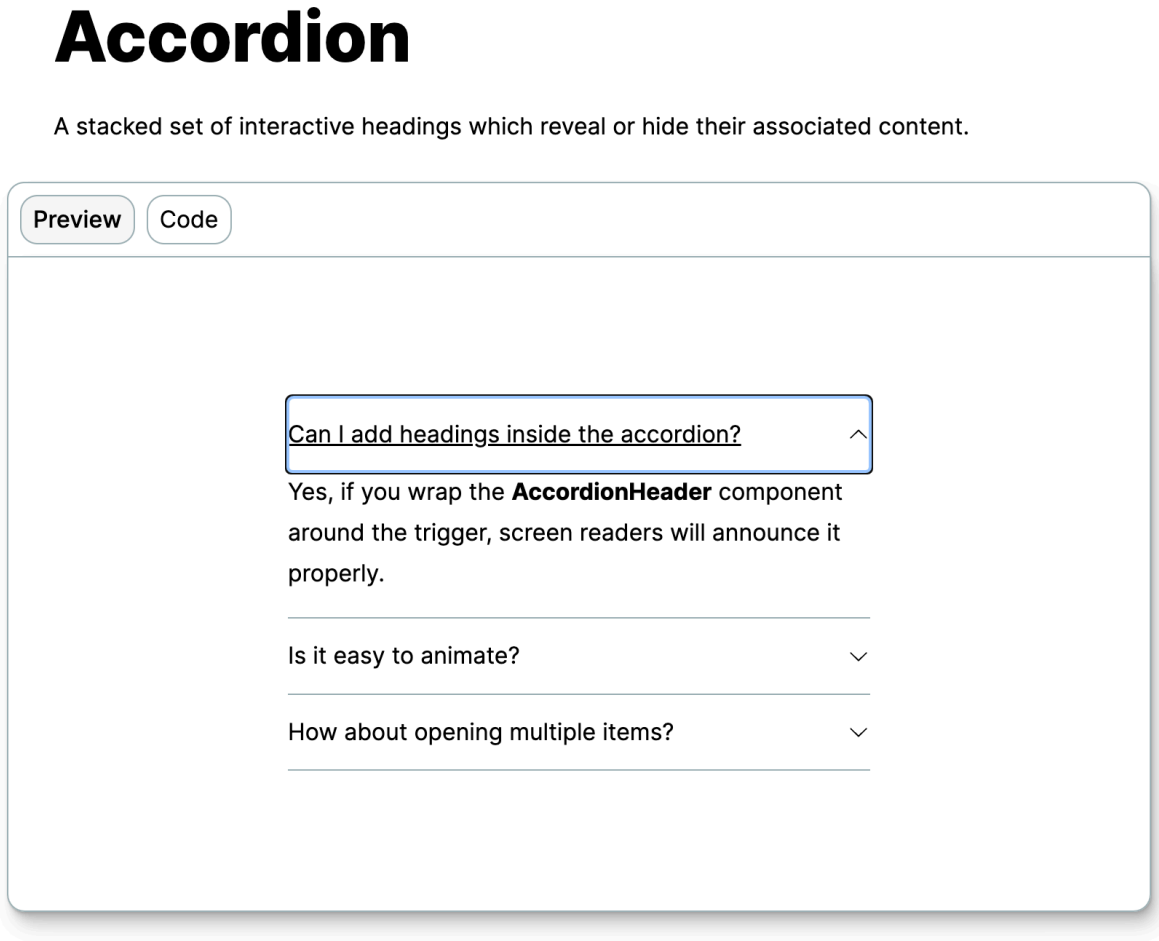
Qwik UI

At the end of 2022, thanks to the community, the Qwik UI[48] project was born, a library of components dedicated to and implemented on Qwik. To use the Qwik API and define and experiment with best practices in using Qwik. The last few years have allowed us to significantly improve the framework's codebase. By developing a series of low-level components, we touch on borderline cases and also use the various library integrations. This helped and still helps the ecosystem. The idea is that of a library of components that are provided in the form of a Headless Kit, therefore a series of bare-bones components where we can insert our styles independently. The library is still in the beta phase and other kits will almost certainly be released shortly, this will allow end users to not worry about defining the style for our components. So for example, the kit that Tailwind uses behind the scenes could come out and you just need to import the component. Let's see an example of components to understand how this can be implemented. Already in the previous chapters, with the creation of a button, we have seen an example of creating an internal library of components. Here it is very similar, except that a series of properties must be designed in advance to be useful to everyone.

Accordion component

Let's see the `Accordion` component together.

[48] https://qwikui.com/



accordion

```
import {
  AccordionContent,
  AccordionHeader,
  AccordionItem,
  AccordionRoot,
  AccordionTrigger,
} from "@qwik-ui/headless";
import SVG from "./svg";

export default component$(() => {
  return (
    <>
      <div class="flex w-full justify-center">
```

```
14          <AccordionRoot
15            animated
16            enhance={true}
17            class="w-[min(400px,_100%)]"
18          >
19            <AccordionItem class="border-b">
20              <AccordionHeader as="h3">
21                <AccordionTrigger class="group flex w-full items-center justify-betwee\
22 n rounded-t-sm py-4 text-left hover:underline">
23                  <span>
24                    Can I add headings inside the accordion?
25                  </span>
26                  <span class="pl-2">
27                    <SVG class="ease transition-transform duration-500 group-aria-expa\
28 nded:rotate-180 group-aria-expanded:transform" />
29                  </span>
30                </AccordionTrigger>
31              </AccordionHeader>
32              <AccordionContent class="accordion-animation-1 overflow-hidden">
33                <p class="pb-4">
34                  Yes, if you wrap the{" "}
35                  <strong>AccordionHeader</strong> component
36                  around the trigger, screen readers will
37                  announce it properly.
38                </p>
39              </AccordionContent>
40            </AccordionItem>
41            <AccordionItem class="border-b">
42              <AccordionHeader as="h3">
43                <AccordionTrigger class="group flex w-full items-center justify-betwee\
44 n py-4 text-left hover:underline">
45                  <span>Is it easy to animate?</span>
46                  <span class="pl-2">
47                    <SVG class="ease transition-transform duration-500 group-aria-expa\
48 nded:rotate-180 group-aria-expanded:transform" />
49                  </span>
50                </AccordionTrigger>
51              </AccordionHeader>
52              <AccordionContent class="accordion-animation-1 overflow-hidden">
53                <p class="pb-4">
54                  Yup! You can even use animations or CSS
55                  transitions using the{" "}
```

```
                <strong>animated</strong> prop on the
                accordion root!
              </p>
            </AccordionContent>
          </AccordionItem>
          <AccordionItem class="border-b">
            <AccordionHeader as="h3">
              <AccordionTrigger class="group flex w-full items-center justify-betwee\
n py-4 text-left hover:underline aria-expanded:rounded-none">
                <span>
                  How about opening multiple items?
                </span>
                <span class="pl-2">
                  <SVG class="ease transition-transform duration-500 group-aria-expa\
nded:rotate-180 group-aria-expanded:transform" />
                </span>
              </AccordionTrigger>
            </AccordionHeader>
            <AccordionContent class="accordion-animation-1 overflow-hidden">
              <p class="pb-4">
                You can do that by setting the{" "}
                <strong>behavior</strong> prop to "multi" on
                the Accordion
              </p>
            </AccordionContent>
          </AccordionItem>
        </AccordionRoot>
      </div>
    </>
  );
});
```

Here we see that, to give maximum customization to the end user who will use the component, various components have been defined. We see them listed in the initial imports, and together they make up the final result. These components are in fact `AccordionContent`, `AccordionHeader`, `AccordionItem`, `AccordionRoot`, and `AccordionTrigger`. Tailwind was used to add style to these headless components, but let's analyze them to understand how you can go into such detail to create something complex.

AccordionRoot

```
import { accordionRootContextId } from "./accordion-context-id";
import { type AccordionRootContext } from "./accordion-context.type";

export type AccordionRootProps = {

} & QwikIntrinsicElements["div"];

export type AccordionItemProps = {
  behavior?: "single" | "multi";
  animated?: boolean;
  enhance?: boolean;
  collapsible?: boolean;
  onSelectedIndexChange$?: PropFunction<
    (index: number) => void
  >;
  onFocusIndexChange$?: PropFunction<
    (index: number) => void
  >;
} & QwikIntrinsicElements["div"];

export const AccordionRoot = component$(
  ({
    collapsible = true,
    behavior = "single",
    animated = false,
    onSelectedIndexChange$,
    onFocusIndexChange$,
    ...props
  }: AccordionRootProps) => {
    const rootRef = useSignal<HTMLDivElement | undefined>();
    const rootElement = rootRef.value;
    const currFocusedTriggerIndexSig =
      useSignal<number>(-1);
    const currSelectedTriggerIndexSig =
      useSignal<number>(-1);
    const selectedTriggerIdSig = useSignal<string>("");
    const triggerElementsSig = useSignal<
      HTMLButtonElement[]
    >([]);

    useTask$(({ track }) => {
      track(() => currSelectedTriggerIndexSig.value);
```

```
43
44        if (onSelectedIndexChange$) {
45          onSelectedIndexChange$(
46            currSelectedTriggerIndexSig.value
47          );
48        }
49      });
50
51      useTask$(({ track }) => {
52        track(() => currFocusedTriggerIndexSig.value);
53        if (onFocusIndexChange$) {
54          onFocusIndexChange$(
55            currFocusedTriggerIndexSig.value
56          );
57        }
58      });
59
60      const updateTriggers$ = $(() => {
61        if (!rootElement) {
62          return;
63        }
64
65        // needs to grab a new array when adding or removing elements dynamically.
66        const getLatestTriggers = Array.from(
67          rootElement.querySelectorAll("[data-trigger-id]")
68        ) as HTMLButtonElement[];
69
70        triggerElementsSig.value = getLatestTriggers.filter(
71          (element) => {
72            if (
73              element.getAttribute("aria-disabled") === "true"
74            ) {
75              return false;
76            }
77
78            return true;
79          }
80        );
81      });
82
83      const focusPreviousTrigger$ = $(() => {
84        // [...]
```

```
85      });
86
87      const focusNextTrigger$ = $(() => {
88        // [...]
89      });
90
91      const focusFirstTrigger$ = $(() => {
92        // [...]
93      });
94
95      const focusLastTrigger$ = $(() => {
96        // [...]
97      });
98
99      // takes a role call of its children (reactive b/c it's a signal)
100     useVisibleTask$(function reIndexTriggers() {
101       updateTriggers$();
102     });
103
104     const contextService: AccordionRootContext = {
105       updateTriggers$,
106       focusFirstTrigger$,
107       focusPreviousTrigger$,
108       focusNextTrigger$,
109       focusLastTrigger$,
110       currFocusedTriggerIndexSig,
111       currSelectedTriggerIndexSig,
112       selectedTriggerIdSig,
113       triggerElementsSig,
114       collapsible,
115       behavior,
116       animated,
117     };
118
119     useContextProvider(
120       accordionRootContextId,
121       contextService
122     );
123
124     return (
125       <div {...props} ref={rootRef}>
126         <Slot />
```

```
      </div>
    );
  }
);
```

Meanwhile, we can notice that in this long component at the end, we have inserted a `<Slot />` which allows you to render the other components that are enclosed within `<AccordionRoot />`. Furthermore, if we go to the beginning of the file we have different properties that are used to cover different cases and satisfy all end users.

Here are the properties that this component accepts:

- **behavior**: Determines whether the Accordion will open one or multiple items at a time.
- **collapsible**: Will allow the accordion to collapse items if set to true.
- **animated**: Allows the trigger to close using the onAnimationEnd$ and onTransitionEnd$ event handlers.
- **onSelectedIndexChange$**: An event hook that gets notified whenever the selected index changes.
- **onFocusIndexChange$**: An event hook that gets notified whenever the focus index changes.

In addition to the methods that will be used to make the component dynamic, there is `useContextProvider(accordionRootContextId, contextService);` which is used to create a context to be used in the other components.

AccordionItem

```
import { accordionItemContextId } from "./accordion-context-id";
import { type AccordionItemContext } from "./accordion-context.type";

export type AccordionItemProps = {
  defaultValue?: boolean,
} & QwikIntrinsicElements["div"];

export const AccordionItem = component$(
  ({
    defaultValue = false,
    id,
    ...props
  }: AccordionItemProps) => {
    const localId = useId();
    const itemId = id || localId;

    const isTriggerExpandedSig =
      useSignal < boolean > defaultValue;
```

```
19
20      const itemContext: AccordionItemContext = {
21        itemId,
22        isTriggerExpandedSig,
23        defaultValue,
24      };
25
26      useContextProvider(accordionItemContextId, itemContext);
27
28      return (
29        <div
30          id={itemId}
31          data-type="item"
32          data-item-id={itemId}
33          {...props}
34        >
35          <Slot />
36        </div>
37      );
38    }
39  );
```

Here are the properties that this component accepts:

- **id**: Allows the consumer to supply their id attribute for the item and its descendants.
- **defaultValue**: Determines whether the Accordion Item will open by default.

This component is used to render the rest of the components and therefore is short. Here too, a context `useContextProvider(accordionItemContextId, itemContext);` is created, again to share an internal state with the components that will be inserted in place of the `<Slot />`.

AccordionHeader

```
import { accordionItemContextId } from "./accordion-context-id";

type HeadingUnion = "h1" | "h2" | "h3" | "h4" | "h5" | "h6";

export type AccordionHeaderProps =
  QwikIntrinsicElements[HeadingUnion] & {
    as?: HeadingUnion,
  };

export const AccordionHeader = component$(
  ({ as = "h3", ...props }: AccordionHeaderProps) => {
    const itemContext = useContext(accordionItemContextId);
    const itemId = itemContext.itemId;
    const headerId = `${itemId}-header`;

    const PolymorphicHeading = as;

    return (
      <PolymorphicHeading id={headerId} {...props}>
        <Slot />
      </PolymorphicHeading>
    );
  }
);
```

Here are the properties that this component accepts:

- **as**: Sets the heading tag of the Accordion Header

Thanks to `QwikIntrinsicAttributes` you can inherit the properties of native HTML elements.

- **class**: CSS classes you can apply
- **style**: CSS styles you can apply
- **disabled**: Disables the element
- **onClick$**: A custom click handler that executes when the element is clicked.
- **onKeyDown$**: A custom click handler that executes when the key is pressed down.
- **onFocus$**: A custom click handler that executes when an element is focused.

Here the `itemContext` is extracted from the `accordionItemContextId` context and the `headerId` to be assigned to the component is used.

AccordionTrigger

```
import {
  accordionItemContextId,
  accordionRootContextId,
} from "./accordion-context-id";

import { KeyCode } from "../../utils/key-code.type";

const accordionPreventedKeys = [
  KeyCode.Home,
  KeyCode.End,
  KeyCode.PageDown,
  KeyCode.PageUp,
  KeyCode.ArrowDown,
  KeyCode.ArrowUp,
];

export type AccordionTriggerProps = {
  disabled?: boolean;
} & QwikIntrinsicElements["button"];

export const AccordionTrigger = component$(
  ({ disabled, ...props }: AccordionTriggerProps) => {
    const contextService = useContext(
      accordionRootContextId
    );
    const itemContext = useContext(accordionItemContextId);

    const ref = useSignal<HTMLButtonElement>();
    const triggerElement = ref.value;

    const behavior = contextService.behavior;
    const collapsible = contextService.collapsible;
    const defaultValue = itemContext.defaultValue;

    const triggerElementsSig =
      contextService.triggerElementsSig;
    const triggerId = `${itemContext.itemId}-trigger`;

    const updateTriggers$ = contextService.updateTriggers$;

    /* content panel id for aria-controls */
    const contentId = `${itemContext.itemId}-content`;
```

```
43
44      const selectedTriggerIdSig =
45        contextService.selectedTriggerIdSig;
46      const isTriggerExpandedSig =
47        itemContext.isTriggerExpandedSig;
48
49      /* The consumer can use these two signals. */
50      const currFocusedTriggerIndexSig =
51        contextService.currFocusedTriggerIndexSig;
52      const currSelectedTriggerIndexSig =
53        contextService.currSelectedTriggerIndexSig;
54
55      const setSelectedTriggerIndexSig$ = $(() => {
56        if (behavior === "single" && triggerElement) {
57          currSelectedTriggerIndexSig.value =
58            triggerElementsSig.value.indexOf(triggerElement);
59        }
60      });
61
62      const setCurrFocusedIndexSig$ = $(() => {
63        if (!triggerElement) {
64          return;
65        }
66
67        currFocusedTriggerIndexSig.value =
68          triggerElementsSig.value.indexOf(triggerElement);
69      });
70
71      useTask$(function resetTriggersTask({ track }) {
72        track(() => selectedTriggerIdSig.value);
73
74        if (
75          behavior === "single" &&
76          triggerId !== selectedTriggerIdSig.value
77        ) {
78          isTriggerExpandedSig.value = false;
79        }
80      });
81
82      useTask$(function openDefaultValueTask() {
83        if (defaultValue) {
84          isTriggerExpandedSig.value = true;
```

```
85        }
86      });
87
88      useVisibleTask$(function navigateTriggerVisibleTask({
89        cleanup,
90      }) {
91        if (!triggerElement) {
92          return;
93        }
94
95        /* runs each time a new trigger is added. We need to tell the root it's time t\
96  o take a role call. */
97        if (!disabled) {
98          updateTriggers$();
99        }
100
101       function keyHandler(e: KeyboardEvent) {
102         if (
103           accordionPreventedKeys.includes(e.key as KeyCode)
104         ) {
105           e.preventDefault();
106         }
107       }
108
109       triggerElement.addEventListener(
110         "keydown",
111         keyHandler
112       );
113       cleanup(() => {
114         triggerElement?.removeEventListener(
115           "keydown",
116           keyHandler
117         );
118       });
119     });
120
121     useVisibleTask$(
122       function cleanupTriggersTask({ cleanup }) {
123         cleanup(() => {
124           updateTriggers$();
125         });
126       },
```

```
127       { strategy: "document-ready" }
128     );
129     return (
130       <button
131         ref={ref}
132         id={triggerId}
133         disabled={disabled}
134         aria-disabled={disabled}
135         data-trigger-id={triggerId}
136         data-state={
137           isTriggerExpandedSig.value ? "open" : "closed"
138         }
139         onClick$={
140           disabled
141             ? []
142             : [
143                 $(() => {
144                   selectedTriggerIdSig.value = triggerId;
145
146                   setSelectedTriggerIndexSig$();
147
148                   collapsible
149                     ? (isTriggerExpandedSig.value =
150                         !isTriggerExpandedSig.value)
151                     : (isTriggerExpandedSig.value = true);
152                 }),
153                 props.onClick$,
154               ]
155         }
156         aria-expanded={isTriggerExpandedSig.value}
157         aria-controls={contentId}
158         onKeyDown$={[
159           $(async (e: QwikKeyboardEvent) => {
160             if (e.key === "ArrowUp") {
161               await contextService.focusPreviousTrigger$();
162             }
163
164             if (e.key === "ArrowDown") {
165               await contextService.focusNextTrigger$();
166             }
167
168             if (e.key === "Home") {
```

```
169               await contextService.focusFirstTrigger$();
170             }
171
172             if (e.key === "End") {
173               await contextService.focusLastTrigger$();
174             }
175           }),
176           props.onKeyDown$,
177         ]}
178         onFocus$={[setCurrFocusedIndexSig$, props.onFocus$]}
179         {...props}
180       >
181         <Slot />
182       </button>
183     );
184   }
185 );
```

Here too we have a series of properties that come directly from `QwikIntrinsicElements['button']` we inherit all the properties plus we have the possibility of passing the property to deactivate the button. In terms of accessibility, we are supporting everything necessary.

- **Tab**: Moves focus to the next focusable trigger.
- **Shift + Tab**: Moves focus to the previous focusable trigger.
- **Space / Enter**: Expand or collapse the Accordion Trigger.
- **Up Arrow**: When focus is on an accordion trigger, it will move to the previous one, or the last if at the top.
- **Down Arrow**: When focus is on an accordion trigger, it will move to the next one, or the first if at the bottom.
- **Home**: When on an Accordion Trigger, it will focus to the first Accordion Trigger.
- **End**: When on an Accordion Trigger, it will focus to the last Accordion Trigger.

AccordionContent

```
import {
  accordionItemContextId,
  accordionRootContextId,
} from "./accordion-context-id";

export type ContentProps = QwikIntrinsicElements["div"];

export const AccordionContent = component$(
  ({ ...props }: ContentProps) => {
    const contextService = useContext(
      accordionRootContextId
    );
    const itemContext = useContext(accordionItemContextId);

    const ref = useSignal<HTMLElement>();
    const contentElement = ref.value;
    const contentId = `${itemContext.itemId}-content`;

    const animated = contextService.animated;
    const defaultValue = itemContext.defaultValue;
    const totalHeightSig = useSignal<number>(0);

    const isTriggerExpandedSig =
      itemContext.isTriggerExpandedSig;
    const isContentHiddenSig = useSignal<boolean>(
      !defaultValue
    );

    const hideContent$ = $(() => {
      if (!isTriggerExpandedSig.value) {
        isContentHiddenSig.value = true;
      }
    });

    useStylesScoped$(`
    /* check global.css utilites layer for animation */
    @keyframes accordion-open {
      0% {
        height: 0;
      }
      100% {
        height: var(--qwikui-accordion-content-height);
```

```
      }
    }

    @keyframes accordion-close {
        0% {
          height: var(--qwikui-accordion-content-height);
        }
        100% {
          height: 0;
        }
      }
  `);

    /* allows animate / transition from display none */
    useTask$(function animateContentTask({ track }) {
      if (!animated) {
        return;
      }

      track(() => isTriggerExpandedSig.value);

      if (isTriggerExpandedSig.value) {
        isContentHiddenSig.value = false;
      }
    });

    /* calculates height of the content container based on children */
    useVisibleTask$(function calculateHeightVisibleTask({
      track,
    }) {
      if (animated === false) {
        return;
      }

      track(() => isContentHiddenSig.value);

      if (totalHeightSig.value === 0) {
        getCalculatedHeight();
      }

      function getCalculatedHeight() {
        if (!contentElement) {
```

```
85            return;
86          }
87
88          const contentChildren = Array.from(
89            contentElement.children
90          ) as HTMLElement[];
91
92          contentChildren.forEach((element, index) => {
93            totalHeightSig.value += element.offsetHeight;
94
95            if (index === contentChildren.length - 1) {
96              contentElement.style.setProperty(
97                "--qwikui-accordion-content-height",
98                `${totalHeightSig.value}px`
99              );
100           }
101         });
102       }
103     });
104
105     return (
106       <div
107         ref={ref}
108         role="region"
109         id={contentId}
110         data-content-id={contentId}
111         data-state={
112           isTriggerExpandedSig.value ? "open" : "closed"
113         }
114         hidden={
115           animated
116             ? isContentHiddenSig.value
117             : !isTriggerExpandedSig.value
118         }
119         onAnimationEnd$={[
120           hideContent$,
121           props.onAnimationEnd$,
122         ]}
123         onTransitionEnd$={[
124           hideContent$,
125           props.onTransitionEnd$,
126         ]}
```

```
        style={{
          ["--qwikui-collapsible-content-height" as string]:
            "var(--qwikui-accordion-content-height)",
          ["--qwikui-collapsible-content-width" as string]:
            "var(--qwikui-accordion-content-width)",
        }}
        {...props}
      >
        <Slot />
      </div>
    );
  }
);
```

This is the last component, and we are going to exploit a series of properties that come directly from `QwikIntrinsicElements['div']`. We see that the animation of the icon to open and close the accordion was then implemented using CSS.

For a simple component as the accordion may seem, we see that there is a lot of code behind it and therefore it makes us reflect on the fact that implementing an accessible component that satisfies everyone's needs is not simple.

End-to-end Component Testing

The components of the library are all tested and this is no exception we can go and see the Cypress tests that have been implemented. Here is a link to the previous section where we went to see how to test our component in isolation.

```
import {
  AccordionRoot,
  AccordionItem,
  AccordionContent,
  AccordionTrigger,
  AccordionHeader,
} from "./index";

interface AccordionProps {
  behavior?: "single" | "multi";
  collapsible?: boolean;
}

const ThreeItemAccordion = component$(
```

```
15    ({ behavior, collapsible, ...props }: AccordionProps) => {
16      return (
17        <AccordionRoot
18          behavior={behavior}
19          collapsible={collapsible}
20          {...props}
21        >
22          <AccordionItem class="border-b">
23            <AccordionTrigger>Trigger 1</AccordionTrigger>
24            <AccordionContent>
25              <p>Content 1</p>
26            </AccordionContent>
27          </AccordionItem>
28          <AccordionItem class="border-b">
29            <AccordionTrigger>Trigger 2</AccordionTrigger>
30            <AccordionContent>
31              <p>Content 2</p>
32            </AccordionContent>
33          </AccordionItem>
34          <AccordionItem class="border-b">
35            <AccordionTrigger>Trigger 3</AccordionTrigger>
36            <AccordionContent>
37              <p>Content 3</p>
38            </AccordionContent>
39          </AccordionItem>
40        </AccordionRoot>
41      );
42    }
43  );
44
45  describe("Critical Functionality", () => {
46    it("INIT", () => {
47      cy.mount(<ThreeItemAccordion />);
48
49      cy.checkA11yForComponent();
50    });
51
52    it(`GIVEN 3 accordion items
53        WHEN clicking the 2nd item's trigger
54        THEN the 2nd trigger should have aria expanded set to true
55      `, () => {
56      cy.mount(<ThreeItemAccordion />);
```

```
57
58      cy.findByRole("button", { name: /Trigger 2/i })
59        .click()
60        .should("have.attr", "aria-expanded", "true");
61    });
62
63    it(`GIVEN 3 accordion items
64        WHEN clicking the 2nd item's trigger twice
65        THEN the 2nd trigger should have aria expanded set to false
66    `, () => {
67      cy.mount(<ThreeItemAccordion />);
68
69      cy.findByRole("button", { name: /Trigger 2/i })
70        .click()
71        .click()
72        .should("have.attr", "aria-expanded", "false");
73    });
74
75    it(`GIVEN 3 accordion items
76        WHEN clicking the 2nd item's trigger
77        THEN render the 2nd item's content
78    `, () => {
79      cy.mount(<ThreeItemAccordion />);
80
81      cy.findByRole("button", { name: /Trigger 2/i }).click();
82
83      cy.findByRole("region").should("contain", "Content 2");
84    });
85
86    it(`GIVEN 3 accordion items
87        WHEN clicking the 2nd item's trigger twice
88        THEN the 2nd item's content should be hidden.
89    `, () => {
90      cy.mount(<ThreeItemAccordion />);
91
92      cy.findByRole("button", { name: /Trigger 2/i })
93        .click()
94        .click();
95
96      cy.findByRole("region").should("not.exist");
97    });
98  });
```

```
99
100 describe("Keyboard Navigation", () => {
101   it(`GIVEN 3 accordion items and the 2nd item's trigger is focused
102   WHEN the enter key is pressed
103   THEN the content of the 2nd item should render
104 `, () => {
105     cy.mount(<ThreeItemAccordion />);
106
107     cy.findByRole("button", { name: /Trigger 2/i })
108       .focus()
109       .type(`{enter}`);
110
111     cy.findAllByRole("region").should(
112       "contain",
113       "Content 2"
114     );
115   });
116
117   it(`GIVEN 3 accordion items and the 3rd item's trigger is focused
118     WHEN the space key is pressed
119     THEN the content of the 3rd item should render
120   `, () => {
121     cy.mount(<ThreeItemAccordion />);
122
123     cy.findByRole("button", { name: /Trigger 3/i })
124       .focus()
125       .type(" ");
126
127     cy.findAllByRole("region").should(
128       "contain",
129       "Content 3"
130     );
131   });
132
133   it(`GIVEN 3 accordion items and the focus is on the 1st trigger
134     WHEN down arrow key is pressed
135     THEN the next trigger should get focus
136   `, () => {
137     cy.mount(<ThreeItemAccordion behavior="single" />);
138
139     cy.findByRole("button", { name: /Trigger 1/i })
140       .focus()
```

```
141         .type("{downArrow}");
142
143       cy.findByRole("button", { name: /Trigger 2/i }).should(
144         "have.focus"
145       );
146     });
147
148     it(`GIVEN 3 accordion items and the focus is on the 2nd trigger
149     WHEN the up arrow key is pressed
150     THEN the previous trigger should get focus
151     `, () => {
152       cy.mount(<ThreeItemAccordion behavior="single" />);
153
154       cy.findByRole("button", { name: /Trigger 1/i })
155         .focus()
156         .type("{downArrow}");
157
158       cy.findByRole("button", { name: /Trigger 2/i })
159         .focus()
160         .type("{upArrow}");
161
162       cy.findByRole("button", { name: /Trigger 1/i }).should(
163         "have.focus"
164       );
165     });
166 });
167
168 describe("Prop Behavior", () => {
169     it(`GIVEN 3 accordion items with type single
170         WHEN clicking the 1st trigger, and soon after the 2nd trigger
171         THEN the content of the 1st item should close, and the 2nd should open.
172       `, () => {
173       cy.mount(<ThreeItemAccordion behavior="single" />);
174
175       cy.findByRole("button", { name: /Trigger 1/i })
176         .click()
177         .should("have.attr", "aria-expanded", "true");
178
179       cy.findByRole("button", { name: /Trigger 2/i })
180         .click()
181         .should("have.attr", "aria-expanded", "true");
182
```

```
    cy.findByRole("button", { name: /Trigger 1/i }).should(
      "have.attr",
      "aria-expanded",
      "false"
    );
  });

  it(`GIVEN 3 accordion items of type multi
      WHEN clicking multiple different triggers
      THEN none of the content should close.
  `, () => {
    cy.mount(<ThreeItemAccordion behavior="multi" />);

    cy.findByRole("button", { name: /Trigger 1/i }).click();
    cy.findByRole("button", { name: /Trigger 2/i }).click();

    cy.findAllByRole("region")
      .eq(0)
      .should("contain", "Content 1");
    cy.findAllByRole("region")
      .eq(1)
      .should("contain", "Content 2");
  });

  const DefaultValueAccordion = component$(() => {
    return (
      <AccordionRoot>
        <AccordionItem class="border-b">
          <AccordionTrigger>Trigger 1</AccordionTrigger>
          <AccordionContent>Content 1</AccordionContent>
        </AccordionItem>
        <AccordionItem defaultValue>
          <AccordionTrigger>Trigger 2</AccordionTrigger>
          <AccordionContent>Content 2</AccordionContent>
        </AccordionItem>
      </AccordionRoot>
    );
  });

  it(`GIVEN 2 accordion items
      WHEN the 2nd item has the defaultValue prop
      THEN the 2nd item's content should be open.
```

```
225   `, () => {
226     cy.mount(<DefaultValueAccordion />);
227
228     cy.findByRole("region").should("contain", "Content 2");
229   });
230
231   it(`GIVEN 2 accordion items of type single
232       WHEN the 2nd item has the defaultValue prop, and the 1st trigger is clicked
233       THEN the 2nd item's content should be closed, and the 1st opened.
234   `, () => {
235     cy.mount(<DefaultValueAccordion />);
236
237     cy.findByRole("button", { name: /Trigger 1/i }).click();
238
239     cy.findByRole("region").should("contain", "Content 1");
240
241     cy.findByRole("region").eq(1).should("not.exist");
242   });
243
244   const HeaderAccordion = component$(() => {
245     return (
246       <AccordionRoot>
247         <AccordionItem class="border-b">
248           <AccordionHeader as="h4">
249             <AccordionTrigger>Trigger 1</AccordionTrigger>
250             <AccordionContent>Content 1</AccordionContent>
251           </AccordionHeader>
252         </AccordionItem>
253       </AccordionRoot>
254     );
255   });
256
257   it(`GIVEN 1 accordion item
258       WHEN there is an AccordionHeader wrapped around the item
259       THEN it should be a heading
260   `, () => {
261     cy.mount(<HeaderAccordion />);
262
263     cy.findByRole("heading");
264   });
265
266   it(`GIVEN 1 accordion item with the as prop on an accordion header
```

```
      WHEN as is set to h4
      THEN AccordionHeader should be rendered as an h4
  `, () => {
    cy.mount(<HeaderAccordion />);

    cy.get("h4");
  });
});
```

Here we can see that although we have several tests for this component, I omitted many because otherwise the file would have become too long. The description of the various tests is explanatory, but it should be noted that initially an accessibility check is immediately performed with the Cypress `checkA11yForComponent` method. In the various tests, a component created on the fly is used, which allows us to carry out our tests in isolation. Among the various tests, those relating to `Keyboard Navigation` are also performed, which is fundamental for accessibility. I invite you to take a look at it to understand what the code does because by looking at other people's code we can learn a lot.

Qwik UI team

Together with other developers, I started this project which was born with the desire to produce a library of totally accessible components, in fact we have never compromised on the topic of accessibility and use of Qwik with the best practices. From the beginning, these two have been the key points, with the addition of the fact that this project was born and wants to remain a community-driven project. Finding time for this project was not easy, during working hours there is no possibility of dedicating oneself to extra activities, and therefore as often happens, it is precisely by sacrificing free time that one can make one's contribution. Today, I am not as actively engaged in the project, but I keep track of its developments from a distance. If there is a need and time permits, I am always happy to lend a hand. I want to express my gratitude to all the individuals who have collaborated on the project; even though it is still in beta, their collective effort has resulted in truly sensational work.

Summary

We opened this chapter by providing an overview of why writing tests is a good practice. Detecting bugs in advance is certainly the trigger that motivates us to create a well-made test suite. This will allow us to improve the quality of our code and save money in the medium/long term. Refactoring is made easier thanks to this practice and the result is clean, well-structured code that will be easier to understand. By writing small decoupled code functions our application will become modular and easy to test. It's important to think small and not want to solve the whole problem right away. We then saw the various types of tests together, we started by analyzing the unit tests, which are very important because they have high execution speed and return immediate feedback. We have seen the Visual regression tests which are useful to avoid producing graphic errors that we did not notice with our own eyes. These tests allow us to automatically perform tests on multiple devices or resolutions, which is of great help when we need to cover a very large range of devices that may not necessarily be available for manual testing. With the integration tests, we started

to see how to simulate the user and the various flows that he can perform in our application. Having these tests gives us peace of mind that our code hasn't introduced any regressions and that the integrations we've tested have remained as desired. Multiple integration tests are combined to build the end-to-end tests. While these tests can be expensive in terms of maintenance, they provide us with a significant level of security. As mentioned, the ideal would at least be to test the main flow to ensure that users can use the core functionality of our platform. To offer everyone the opportunity to use the application, it is also correct to also check that accessibility is highly respected. With modern HTML you just need to be careful in using the right tags and attributes to make our application truly accessible. We discussed together how it is important to understand when it is right to invest time in testing and I introduced you to TDD, a truly exceptional methodology that offers us many advantages because it allows us to write better code. We integrated Cypress as a tool to run our end-to-end tests, we started from a first component and performed an initial refactoring which allowed us to improve the code, the tests then certified that our new implementation did not change the expected result. Finally, we analyzed the QwikUI library and the Accordion component in detail, we saw together the tests that have been implemented to guarantee consistency and full accessibility. In the next chapter, we will make some final considerations and we will see together some advice on how to migrate your application to Qwik.

Final thoughts

A bit of history of Qwik

During these chapters, we went into detail about Qwik, an innovative framework that, compared to all the others available, approaches flows in a different way to allow you to create modern applications. Clearly, like all ideas, Qwik also started with a prototype which evolved over the months. If we execute the following command within the Qwik codebase we can see how and when the repository was born.

```
git log --reverse
```

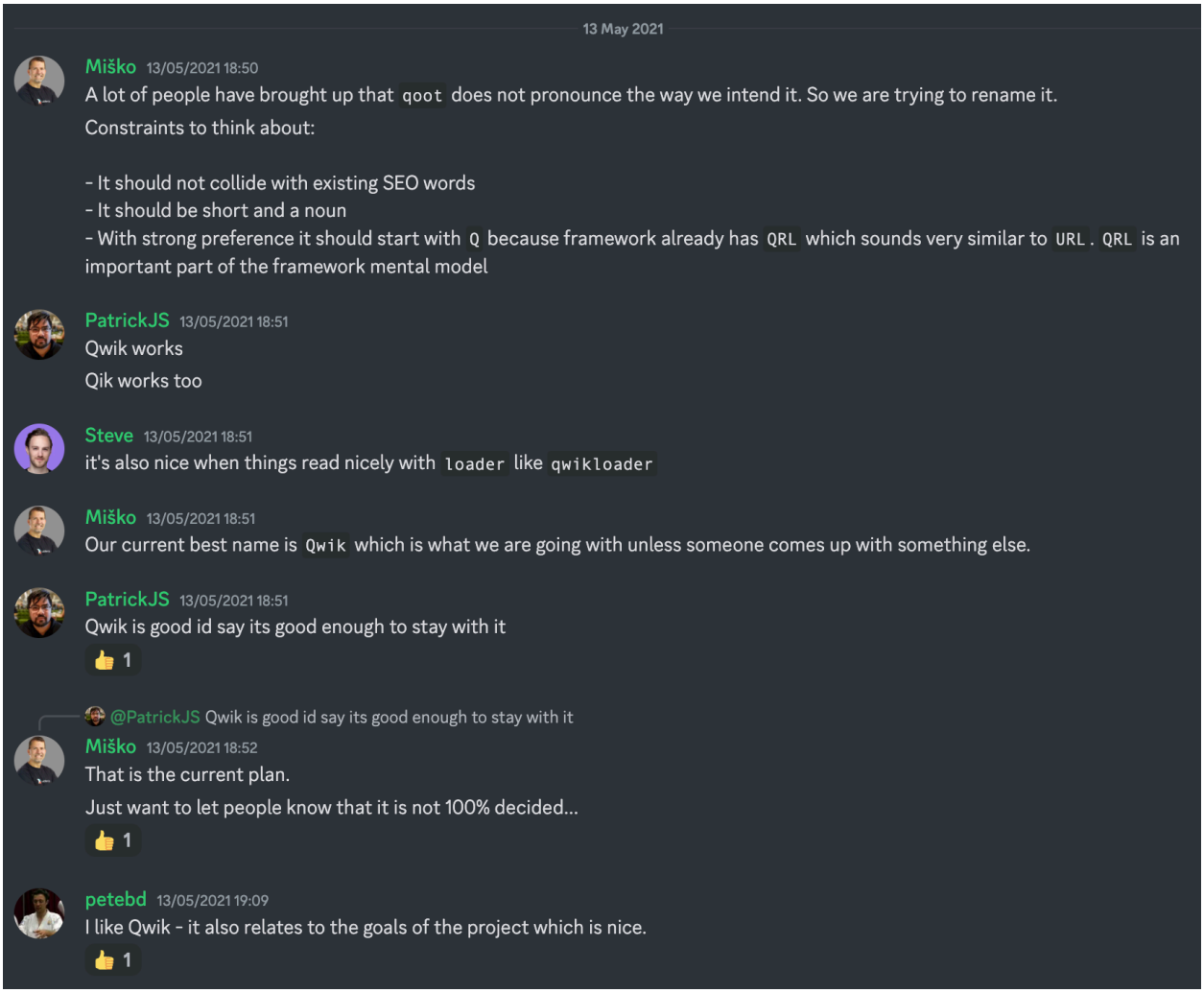
Output:

```
commit bfaf99c568b0eea147a40d1503370cd6de1bf4ea
Author: Miško Hevery <misko@hevery.com>
Date:   Sat Feb 13 13:23:14 2021 -0800

    Initial commit

commit ea318b072b7c43078e32d3500b6fdf5f88ac2a8a
Author: Misko Hevery <misko@hevery.com>
Date:   Sat Feb 13 15:30:01 2021 -0800

    feat: initial implementation of qootloader vs sample apps

commit 4f90f1d7f986177f767a01225c8e10c8177d4ae0
Author: Misko Hevery <misko@hevery.com>
Date:   Mon Feb 15 13:24:39 2021 -0800

    chore: Setup unit and e2e testing infrastructure.

commit 86dc3e5c1977dd4fe31f41c97b95d73006372749
Author: Misko Hevery <misko@hevery.com>
Date:   Mon Feb 15 22:30:59 2021 -0800

    feat: JSX scaffolding for rendering

commit 7b01a60d7b48fd6f72abbe174ce0c0178e7a3bbc
```

```
Author: Misko Hevery <misko@hevery.com>
Date:   Wed Feb 17 19:22:11 2021 -0800

    chore: setup unit tests infrastructure

commit 600918b704a2fc1167ecca9f838b199b5440149e
Author: Misko Hevery <misko@hevery.com>
Date:   Thu Feb 18 22:43:12 2021 -0800

    feat: basic JSX implementation for rendering
```

It's February 2021, the creator of the framework, `Miško Hevery`, was starting to lay the foundations for what today we can now consider, without presumption, the framework that is changing the world of frontend. This new mental model had already been in his head for many years, initially as an embryonic idea and then more and more concrete. Already in May 2019, in the Keynote of Day3 of the ng-conf[49], Miško Hevery presented to the world his idea who only after a few years would have had the time and strength to make it effective. To present the 2019 talk, the idea had certainly been in his mind for some time, who knows, perhaps as early as 2018. So it is a newly implemented framework, but it was digested for a long time before taking shape. We see that in 2021 there is the first commit, an `initial commit` and then a `feat: initial implementation of qootloader vs sample apps` and then continuing in the following days up to today. We can see that instead of the current `qwikloader`, there was a `qootloader` because the framework was initially called `qoot` and then was renamed to `Qwik`. Also for this decision, we can go back in time and see some old messages in the Qwik Discord server. Here we have gone back to May 2021, when the decision to change the name of the framework was made.

[49] https://youtu.be/-kYtw3CSe6s?t=797



1_name

It's nice to read that obviously, they checked if the domain was available to buy it and then a discussion started on which logo to choose for the framework. It's nice to note that all the decisions that were made were always shared with the community and were open to any changes. Now the way of proceeding with decisions has not changed because the community comes first. Moments are organized to share the news of the framework and external contributions are encouraged. This way of operating makes it pleasant to be part of a project and know that, if we want, we can have our say and influence future changes. The first integrations were born precisely to help the community, and among the first integrations, the one with React was developed. In fact, it is possible to integrate React components within our Qwik applications.

qwikify$

The integration with React components was created under the name `qwikify$` and the use case is soon explained. Let's imagine we have a series of components written in React that we would like to reuse in our application. If we want to use Qwik and use these components it is clear that without this integration it

would not be possible. Furthermore, it is also possible to use React libraries such as MaterialUI, and Threejs, in short, we can draw on the vast React ecosystem. This approach can also be used to get the benefits of Qwik without having to rewrite your React application. This solution is integrated into the Qwik codebase and therefore it is clear that any problems will be followed directly by the Qwik team or by the large community that supports it, in fact, the import of `qwikify$` comes directly from a core Qwik package.

> Thanks to the community it is possible to also integrate Angular, Preact, or other framework components within our Qwik application. These integrations have APIs completely similar to the React one, which we will analyze, and were developed as mentioned by the community. Thanks to the commitment of all the developers involved the Qwik ecosystem has grown enormously.

As we have seen many times throughout this book, with `qwik add` we have understood that we can do many things, the only thing missing is being able to make coffee. So to add the integration with React we can type:

```
pnpm qwik add react
```

This integration configures everything automatically:

```
┌  □  Add Integration  react
│
□  □  Ready?  Add react to your app?
│
│  □ Modify
│     - package.json
│     - vite.config.ts
│
│  □ Create
│     - src/integrations/react/mui.tsx
│     - src/routes/react/index.tsx
│
│  □ Install pnpm dependencies:
│     - @emotion/react ^11.11.1
│     - @emotion/styled ^11.11.0
│     - @mui/material ^5.13.0
│     - @mui/x-data-grid ^6.4.0
│     - @types/react ^18.2.28
│     - @types/react-dom ^18.2.13
│     - react 18.2.0
│     - react-dom 18.2.0
│
◆  Ready to apply the react updates to your app?
│  □ Yes looks good, finish update!
│  □ Nope, cancel update
└
```

We can see that some dependencies will be added to our application and then to create the React example some libraries are installed. A new `qwikReact` plugin will also be added to the Vite configuration. This serves to optimize the dependencies related to React during the compilation phase. The Qwik CLI will also create a `src/integrations/react/mui.tsx` file where our React components are located and where they are wrapped with `qwikify$`.

FILE: `src/integrations/react/mui.tsx`

```
// This pragma is required so that React JSX is used instead of Qwik JSX
/** @jsxImportSource react */

import { Button, Slider } from "@mui/material";
import {
  DataGrid,
  GridColDef,
  GridValueGetterParams,
} from "@mui/x-data-grid";

export const MUIButton = qwikify$(Button);
export const MUISlider = qwikify$(Slider, {
  eagerness: "hover",
});

export const TableApp = qwikify$(() => {
  const columns: GridColDef[] = [
    { field: "id", headerName: "ID", width: 70 },
    {
      field: "firstName",
      headerName: "First name",
      width: 130,
    },
    {
      field: "lastName",
      headerName: "Last name",
      width: 130,
    },
    {
      field: "age",
      headerName: "Age",
      type: "number",
      width: 90,
    },
    {
```

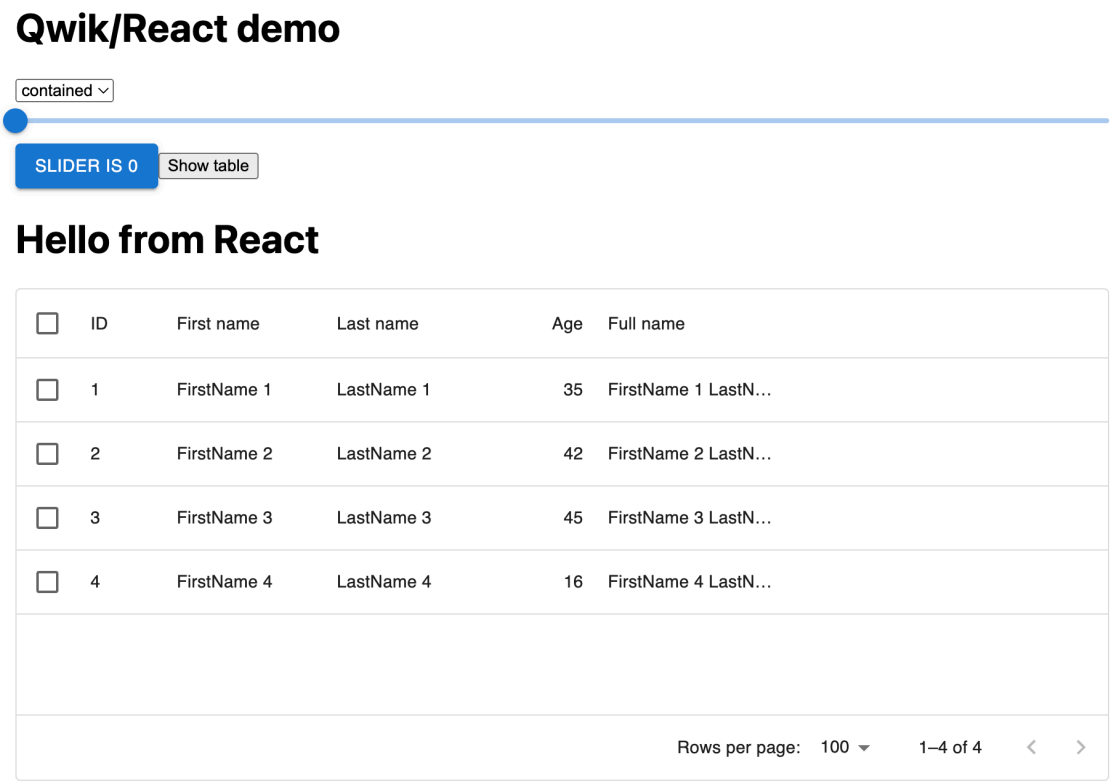
```
      field: "fullName",
      headerName: "Full name",
      description:
        "This column has a value getter and is not sortable.",
      sortable: false,
      width: 160,
      valueGetter: (params: GridValueGetterParams) =>
        `${params.row.firstName || ""} ${
          params.row.lastName || ""
        }`,
    },
  ];

  const rows = [
    {
      id: 1,
      lastName: "LastName 1",
      firstName: "FirstName 1",
      age: 35,
    },
    {
      id: 2,
      lastName: "LastName 2",
      firstName: "FirstName 2",
      age: 42,
    },
    {
      id: 3,
      lastName: "LastName 3",
      firstName: "FirstName 3",
      age: 45,
    },
    {
      id: 4,
      lastName: "LastName 4",
      firstName: "FirstName 4",
      age: 16,
    },
  ];

  return (
    <>
```

```
78        <h1>Hello from React</h1>
79
80        <div style={{ height: 400, width: "100%" }}>
81          <DataGrid
82            rows={rows}
83            columns={columns}
84            checkboxSelection
85          />
86        </div>
87      </>
88    );
89  });
```

And finally in the `https://my-website.com/react` route an example page will be created where we can already see `qwikify$` in action.

> It should be noted that this is not an emulation of React because we are using the real library.

Here is the result in the `/react` route



qwikify

This is a complete example that is provided out of the box but it makes more sense to analyze a basic example in detail to see all the various cases. It is not possible to use React components in Qwik without converting them using `qwikify$`. React components are very similar but behind the scenes, they are very different. React and Qwik cannot be mixed in the same file. Best practices indicate inserting the React components in the `src/integrations/react/` folder.

> It should be noted that with this integration we are not going to exploit the resumability of Qwik on our React components, but we can improve the hydration process of React by postponing it and therefore making it lazy and more. Unfortunately, due to how React works, if we did not execute the hydration, our Counter button, even if pressed, would not perform any action because it is precisely the hydration process that attaches the event to the button.

Let's analyze this React component:

```tsx
/** @jsxImportSource react */
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button
      className="react"
      onClick={() => setCount(count + 1)}
    >
      Count: {count}
    </button>
  );
}

export const QCounter = qwikify$(Counter, {
  eagerness: "hover",
});
```

In our example, we see that we have a React component that defines a Counter. When we export the component and wrap it with `qwikify$` we define the condition for which the React code is executed to perform the lazy hydration. I deliberately said "executed" because behind the scenes thanks to the Qwik service worker we have already downloaded all the files and placed them in the cache to be executed when necessary. So we are rendering the component on the server (with SSR), and we are improving React's hydration by not only making it lazy but also caching all the necessary files in the service worker. All this makes our application better and more performant. We have different configurations to define how to delay the hydration process, let's see them together:

- **eagerness**: '**hover**': The component eagerly hydrates `when the mouse is over the component` and you can use this for lowest-priority UI elements which interactivity is not crucial, and only needs to run on desktop. (hover event is not present on mobile devices)
- **eagerness**: '**idle**': The component eagerly hydrates when the `browser first becomes idle`, i.e., when everything important has already run before and you can use this for lower-priority UI elements that don't need to be immediately interactive.
- **eagerness**: '**load**': The component eagerly hydrates `when the document loads` and you can use this for your immediately visible UI elements that need to be interactive as soon as possible.
- **eagerness**: '**visible**': The component eagerly hydrates `when it becomes visible in the viewport` and you can use this for low-priority UI elements that are either far down the page ("below the fold") or so resource-intensive to load that you would prefer not to load them at all if the user never saw the element.

We therefore have many possibilities to make the expensive process of hydration better, but it is possible to provide the same configuration via a sugar syntax that allows us to give the same instructions. Let's see how:

```
<QCounter client:hover></QCounter>
```

Here we see that we have used `client:hover` as an attribute in our component wrapped with `qwikify$` and this instructs the system to behave like the previous example i.e., like this `export const QCounter = qwikify$(Counter, { eagerness: 'hover' });`. The other attributes we can pass are `client:idle`, `client:load`, and `client:visible`, but thanks to the use of this attribute we can have further possibilities.

- **client:signal**: This is an advanced API that allows to hydrating of the component whenever the passed signal becomes true. This effectively allows you to implement custom strategies for hydration.

```
export default component$(() => {
  const hydrateReact = useSignal(false);
  return (
    <>
      <button onClick$={() => (hydrateReact.value = true)}>
        Hydrate Slider when click
      </button>

      <MUISlider client:signal={hydrateReact}></MUISlider>
    </>
  );
});
```

- **client:event**: The component eagerly hydrates when specified DOM events are dispatched.

```
<MUISlider client:event="click"></MUISlider>
```

- **client:only**: When true, the component will not run in SSR, only in the browser.

```
<MUISlider client:only></MUISlider>
```

Sorry for the long overview of the configurations, but for completeness, it was right to show all the possibilities we have available to use and improve React within our Qwik application.

We can achieve the best benefits by defining large islands rather than many leaves. It is enough that the outermost component is wrapped with `qwikify$`, and the rest of the internal components can be classic React components. It is therefore possible to manage a state internal to our React islands, but thanks to Qwik, we can act as a bridge, and in a very simple way, it is also possible to share the state between our islands. Let's see an example that is often worth a thousand words.

FILE: `src/integrations/react/islands.tsx`

```
function ReactIsland1({
  onClick,
}: {
  onClick: () => void,
}) {
  return (
    <>
      // More reactive things here
      <button onClick={onClick}>+1</button>
    </>
  );
}

function ReactIsland2({ count }: { count: number }) {
  return (
    <>
      // More reactive things here
      <p className="react">Count: {count}</p>
    </>
  );
}

export const QIsland1 = qwikify$(ReactIsland1, {
  eagerness: "idle",
});
export const QIsland2 = qwikify$(ReactIsland2, {
  eagerness: "idle",
});
```

FILE: src/routes/react/index.tsx

```
import {
  QIsland1,
  QIsland2,
} from "../../integrations/react/islands";

export default component$(() => {
  const count = useSignal(0);
  return (
    <main>
      <QIsland1
        onClick$={() => {
```

```
          console.log("click", count.value);
          count.value++;
        }}
      />
      <QIsland2 count={count.value}></QIsland2>
    </main>
  );
});
```

Here we can see that thanks to Qwik's state management, in this case, we are using `useSignal` we can go and share the state between our React islands. This is a behavior that is very simple to manage. However, during these chapters, we have seen that with Astro, it is not possible to perform this sharing except with an external library. Therefore, in my opinion, Qwik's approach is more consistent and robust because we do not have to add other layers but only use the APIs that the framework makes available to us. So I would say that we have everything needed to integrate React into Qwik for a future full migration.

Qwik inside Astro application

Just talking about Astro, if you are already using it, there is a library available that allows you to insert Qwik components into Astro, a framework created more than anything else to render static pages. Adding dynamism with Astro's API is very complex because you have to use vanilla JavaScript. Therefore, Qwik comes in handy because it is more fun and satisfying to write applications with Qwik. There are two ways to add Qwik to Astro, the simplest is to run the following command:

```
pnpm astro add @qwikdev/astro
```

Once this is done you can use Qwik within Astro.

One of the main features of Astro is zero JavaScript at startup, because it handles static content, I would say it's too easy that way. As mentioned before, to add interactivity easily, it interfaces with many frameworks as well as Qwik. But only with Qwik can we exploit the mental model of resumability, and therefore maintain the advantage of having zero JavaScript. With the other frameworks the JavaScript is downloaded to perform the hydration, so with the others, we lose the greatest benefit that Astro can give us.

Migrating your application to Qwik

We have given a nice overview of how it is possible to migrate and merge multiple frameworks. The best approach to refactoring your application is to proceed step by step, starting to create a Qwik application and replacing one page after another, let's start with the simplest one, and make sure to guarantee a graphic appearance equal to the application that we are going to replace. Let's make sure we can replicate all the integrations with external services to avoid having surprises (e.g., Identity provider) to avoid finding ourselves, after weeks of work with a complex integration that requires several days of work. It is better to solve the most important tasks first. So let's proceed step by step and, page after page, remove features from

the old application to inject them into the new one. These are general recommendations valid for all types of refactorings because Qwik is a modern framework that can replicate all the APIs available in other frontend frameworks. It also offers unique features (e.g., micro-frontends without the need for external libraries and much more).

> Throughout this book, we have seen that Qwik uses JSX as template syntax. This can be useful in a possible refactoring from a React or Next.js application because it is simpler and faster; most of the time, it's a copy/paste activity.

But I realize that it may not be easy for you to go to your boss and say: "Qwik is fantastic, let's migrate all our applications to Qwik". Throughout the previous chapters, I have shared some tips on how to do it collaboratively together with the team and how to go and speak to your boss and try to strike the right chords. Sharing choices with the team will make everyone involved in the decision and will avoid unpleasant attacks from someone. You will surely discover that the other team members will also find Qwik truly sensational. On the business side, however, we must translate our findings to an economic or time aspect. Costs and times, in fact, are two very delicate issues for those who do business. I won't deny that if I had my own company, I would probably do the same. But there can be a thousand different variations or scenarios that perhaps do not allow you to use Qwik in everyday work, in fact, the decision often does not depend on us. To be a better developer and increase your knowledge, it is good to look around and understand that there is more than what you use every day at work. We cannot stop learning just because of these decisions; they do not depend on us, and we have no control over them. So go, try, experiment, and make your reasoning.

Micro-frontends with Qwik

The approach just described offers us a beautiful hook for the micro-frontends architecture. The first page can only be the beginning of the migration, and then complete the entire transition of the codebase to Qwik. However, the micro-frontends architecture was created for a different purpose, the need to scale development and coordinate various teams within medium/large applications. Imagine having an application developed by different teams and having to ship features and fix bugs all together within the same code base. It is very complex to coordinate the development of many people. Taking inspiration from micro-services, the micro-frontends architecture involves the division of the final application into micro applications (aka micro-frontends). These micro apps are as independent as possible to avoid the various teams having to coordinate frequently, all this to be able to scale better and avoid wasting effort. However, the final application must be uniform, so libraries are usually used (e.g., module federation[50]) to load the various micro-frontends to create a single final application. Here too, Qwik still amazes us because no external library is needed to "merge" multiple applications into a single final one. It will be sufficient to load them into a Qwik host app and by its nature, it will then be possible to use the final application optimally. With micro-frontends architecture, the various applications can reside in independent repositories or a monorepo, a single repository containing all the applications. By taking advantage of the monorepo approach, I created a working example of Qwik micro-frontends in this repository qwik-micro-frontends[51] from which you can take inspiration. Qwik also solves a well known problem with micro-frontends, that of sending too much JavaScript into the browser. We have seen in detail how Resumability works and therefore it does not matter if I am loading one, two, or five

[50] https://webpack.js.org/concepts/module-federation/
[51] https://github.com/gioboa/qwik-microfrontends

applications at the same time, the JavaScript that I am going to execute will only be 1KB of our `qwikloader`. Therefore, however big or small my application is, I will always have an instant application.

Managing Monorepo with Qwik and Nx

We said that `qwik-micro-frontends` is a monorepo all the micro-frontends reside in the same repository. To manage the monorepo we can create the various folders one inside the other without the aid of any tool, but as the project evolves some problems arise which are difficult to master without a specific tool to help us. The management of shared dependencies between micro-frontends, the sharing of a component library, or simply the sharing of the style are just some of the problems that we will be able to solve. Fortunately for us, there is Nx[52] the de facto standard for mono repo and offers many integrations including Qwik[53]. Nx can manage all these problems for us and more, it will allow us to focus on our functionality and avoid investing time in configuration and infrastructure tasks which are often boring and repetitive. My favorite feature is incremental compilation with caching because once your application grows, your build and test times will grow too. This impacts all developers running the application locally on their machine but also impacts the build/test/deploy pipelines for the test or production environment. In large projects, this problem is very common because it is easy to see execution times of these processes that last precious minutes. Here, with the use of this tool, it's possible to optimize all these times and productivity will benefit from it.

Real-world Qwik applications

There are so many applications that are using Qwik in production, closing thousands of orders and basing their business on this framework. Not only this, there are medical applications that take hospital bookings, applications that manage online auctions, e-commerce, and many others. They all feature sensational Core Web Vitals and great developer and user experience. Many web agencies have built their stack around Qwik and many others, with legacy applications, are making the gradual transition, it's certainly an intelligent choice to migrate gradually. An idea could be to put a homepage in Qwik in front of the legacy application to take advantage of the truly amazing performance and Core Web Vitals, then the user will be redirected to the legacy application smoothly. The important thing is to keep the design of our pages the same to make this step transparent, without making it obvious that there are two different technologies under the hood.

Summary

This summary marks the end of this in-depth overview of Qwik, I enjoyed being able to share with you my passion for the frontend ecosystem and I am sure that, thanks to the notions learned, you will immediately be able to write Qwik applications with extreme confidence. In this chapter, we have seen together a little of the history of Qwik, the framework as mentioned is a new concept and can be seen from the fact that it uses all the latest technologies to make working with it fast and pleasant. Continuing we saw how easy it is to integrate React components within our Qwik application, with the `qwikify$` core API the integration is smooth. We can delay the hydration process and thanks to the service worker pre-download the JavaScript

[52] https://nx.dev/
[53] https://nx.dev/showcase/example-repos/add-qwik

in a secondary thread to reduce the hydration problem and improve the experience for our users. But you can't only integrate React; thanks to the community, many other integrations have been created that allow us to insert various frameworks within Qwik. Throughout the chapter, we saw that the community created the integration with Astro. If you are already using Astro, you can include Qwik components to enrich the developer experience. It will be easier to make our Astro pages dynamic. The end user will also thank you because you will get an increase in responsiveness. We discussed possible strategies on how to migrate our applications to Qwik and I gave you some advice on how to "sell" this technology choice to your managers or bosses. Don't forget that if you want to grow as a senior developer, you can't be stopped by the fact that your company is not interested in changing or using other technologies, you have to be the one to roll up your sleeves, look around and experiment. We have seen that Qwik also has a unique feature compared to all other frameworks, it can manage micro-frontends without the use of external libraries because out of the box, its architecture lends itself to this approach. Furthermore, we will also solve the well known problem of micro-frontends, too much JavaScript sent to the browser. In the end, we saw that if we want to manage our projects as a monorepo, I would say that Nx is the best choice to automate a whole series of boring and time-consuming tasks. A tool like Nx allows us to make them transparent.

Final thoughts

I take this opportunity to thank `Miško Hevery` from the bottom of my heart for writing the foreword of this book and for believing in me from the first moment we met. It was June 2022 when we were both speakers at a conference. In the speaker's room, Miško was reviewing his talk because he would soon present Qwik to the public. That occasion was the first or perhaps one of the very first times he showed Qwik to the audience. I had the pleasure of seeing the framework in a private demo with him while he was going over the features to show during his talk. It was exciting to see his passion for development and technology. In addition to him, it is also thanks to the excellent Builder.io[54] team and the Qwik community that we can now enjoy a truly sensational framework. Today Qwik is a completely community-driven project and this makes it independent in terms of development and decisions. The present and future of this framework is bright.

I hope you enjoyed reading `Qwik in Action` and that the knowledge you learned in this book will help you make better future decisions. I wanted to leave you with a thought that came to me while writing this chapter. We have seen that the idea of Qwik started from afar. So, if we have an idea, we must not give up at the first difficulties because maybe it is just the wrong time to implement it. If we have passion and perseverance, we can make it possible.

Best wishes, Giorgio.

[54] https://www.builder.io/