

Build an App with TypeScript and the Pexels API



Build an App with TypeScript and the Pexels API

Copyright © 2021 SitePoint Pty. Ltd.

- **Product Manager:** Simon Mackie
- **Technical Editor:** James Hibbard
- **English Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

10-12 Gwynne St,

Richmond, VIC, 3121

Australia

Web: www.sitepoint.com

Email: books@sitepoint.com

About Jack Franklin

Jack is a JavaScript and Ruby Developer working in London, focusing on tooling, ES2015 and ReactJS.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <https://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Chapter 1: A Step-by-Step TypeScript Tutorial for

| | |
|--|-------------|
| Beginners | viii |
| Some Erroneous JavaScript Code | ix |
| Running TypeScript from the Editor | x |
| Installing and Running TypeScript Locally | xi |
| Fixing the Errors in Our JavaScript Code | xii |
| Property <code>querySelector</code> does not exist on type <code>Document</code> | xii |
| Property <code>src</code> does not exist on type <code>HTMLElement</code> | xiii |
| How to Configure TypeScript | xiv |
| Working in strict mode..... | xv |
| Union Types | xvi |
| Implicit any | xvii |
| Describe the Function Signature with JSDoc..... | xvii |
| Declaring Data Types Using an Interface..... | xviii |
| Test if everything is working..... | xix |
| Conclusion | xx |

Chapter 2: Build an Application with TypeScript

| | |
|---------------------------|-----------|
| from Scratch | 21 |
| Pexels API Key | 22 |

| | |
|--|----|
| Setting up Vite as the Build Tool | 23 |
| Installing Dependencies..... | 24 |
| Configuring TypeScript with Vite..... | 25 |
| Building the Project with Vite | 25 |
| Making API Requests with TypeScript..... | 26 |
| Inferring types..... | 27 |
| Interfaces in TypeScript..... | 28 |
| Declaring Function Return Types..... | 32 |
| Generic Types..... | 33 |
| Using Third-party Libraries with TypeScript..... | 35 |
| Using lit-html to Render Image Results | 36 |
| Linting with ESLint-TypeScript..... | 40 |
| Conclusion | 42 |

Chapter 3: Adding More Functionality43

| | |
|--|----|
| Refactoring the App to Create a render Method..... | 44 |
| Rendering a Search Form | 45 |
| Searching for Photos..... | 46 |
| Using FormData to Read Form Values | 48 |
| Using the formData Object | 49 |
| Writing an API for Local Storage | 52 |
| Favoriting Photos..... | 54 |
| Creating pexe1s.ts to Contain Our API Code..... | 55 |

| | |
|--|----|
| Liking a Photo | 57 |
| Defining the Types of Callback Functions | 58 |
| Improving Our Code with readonly..... | 62 |
| Adding Video Results | 66 |
| Rendering Videos | 68 |
| Type Predicates | 69 |
| Rendering Videos | 68 |
| Liking Videos..... | 72 |
| Defining a Resource Type | 75 |
| Defining renderResource..... | 77 |
| Liked Data and Enums..... | 78 |
| Features to Add Next | 83 |
| Conclusion | 83 |

Who Should Read This Book?

This book is for developers who wish to learn TypeScript. We assume a basic knowledge of JavaScript and its tooling, but zero prior knowledge of TypeScript is required to follow along.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>  
<p>It was a lovely day for a walk in the park.  
The birds were singing and the kids were all back at school.</p>
```

Tips, Notes, and Warnings



Hey, You!

Tips provide helpful little pointers.



Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.



Make Sure You Always ...

... pay attention to these important points.



Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- A repository containing all of the code for the final app can be [found on GitHub](#).
- <https://www.sitepoint.com/community/> are SitePoint's forums, for help on any tricky problems.
- books@sitepoint.com is our email address, should you need to contact us to report a problem, or for any other reason.

A Step-by-Step TypeScript Tutorial for Beginners

Jack Franklin

Chapter

1

You've probably heard of TypeScript — the language created and maintained by Microsoft that's had a huge impact on the Web, with many prominent projects embracing and migrating their code to TypeScript. TypeScript is a typed superset of JavaScript. In other words, it adds *types* to JavaScript — and hence the name. But why would you want these types? What benefits do they bring? And do you need to rewrite your entire codebase to take advantage of them? Those questions, and more, will be answered in this TypeScript tutorial for beginners. You'll learn TypeScript basics, including installation, configuration, and declaring data types as an interface, before moving on to build a fully-blown app in TypeScript using the Pexels API.

We assume a basic knowledge of JavaScript and its tooling, but zero prior knowledge of TypeScript is required to follow along.

Some Erroneous JavaScript Code

To start with, let's look at some fairly standard plain JavaScript code that you might come across in any given codebase. It fetches some images from the [Pexels API](#) and inserts them to the DOM.

However, this code has a few typos in it that are going to cause problems. See if you can spot them:

```
const PEXELS_API_KEY = '...';
async function fetchImages(searchTerm, perPage) {
  const result = await fetch(`https://api.pexels.com/v1/search?query=${searchTerm}&per_page=${perPage}`, {
    headers: {
      Authorization: PEXELS_API_KEY,
    }
  });
  const data = await result.json();
  const imagesContainer = document.querySelector('#images-container');
  for (const photo of data.photos) {
    const img = document.createElement('image');
    img.src = photo.src.medium;
    imagesContainer.append(img);
  }
}
fetchImages('dogs', 5);
fetchImages(5, 'cats');
fetchImages('puppies');
```

Can you spot the issues in the above example? Of course, if you ran this code in a browser you'd immediately get errors, but by taking advantage of TypeScript we can get the errors quicker by having TypeScript spot those issues in our editor.

Shortening this feedback loop is valuable — and it gets more valuable as the size of your project

grows. It's easy to spot errors in these 30 lines of code, but what if you're working in a codebase with thousands of lines? Would you spot any potential issues easily then?

Note: there's no need to obtain an API key from Pexels to follow along with this TypeScript tutorial. However, if you'd like to run the code, an API key is entirely free: you just need to [sign up for an account](#) and then generate one.

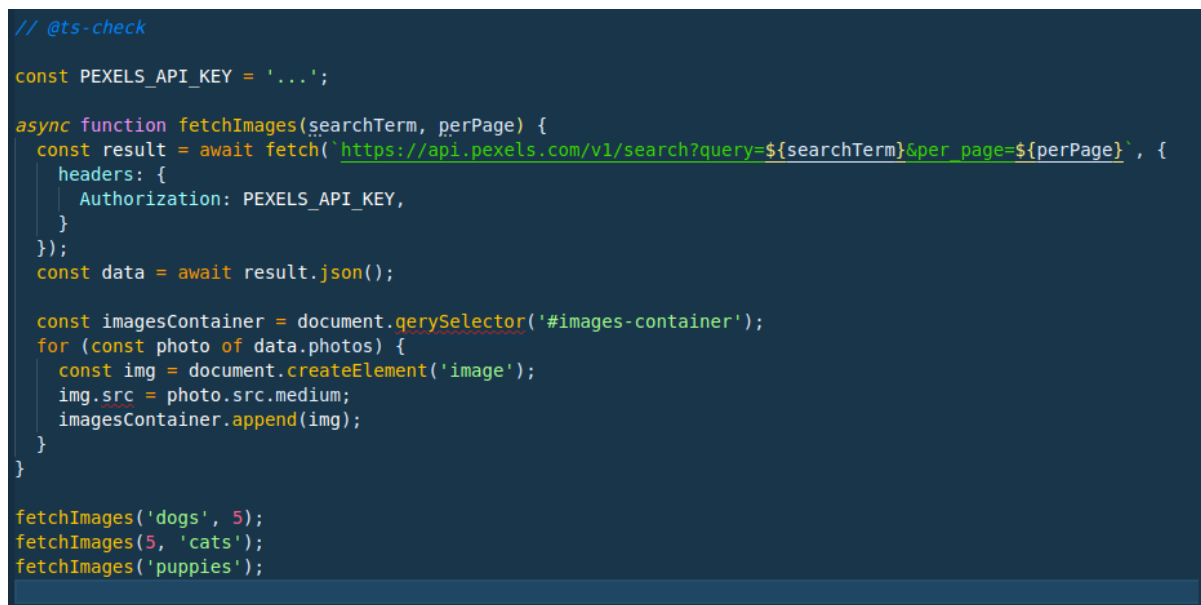
Running TypeScript from the Editor

Once upon a time, TypeScript required that all files be written as `.ts` files. But these days, the onboarding ramp is smoother. You don't need a TypeScript file to write TypeScript code: instead, we can run TypeScript on any JavaScript file we fancy!

If you're a VS Code user (don't panic if you aren't — we'll get to you!), this will work out the box with no extra requirements. We can enable TypeScript's checking by adding this to the very top of our JavaScript file (it's important that it's the first line):

```
// @ts-check
```

You should then get some squiggly red errors in your editor that highlight our mistakes, as pictured below.



```
// @ts-check

const PEXELS_API_KEY = '...';

async function fetchImages(searchTerm, perPage) {
  const result = await fetch(`https://api.pexels.com/v1/search?query=${searchTerm}&per_page=${perPage}`, {
    headers: {
      Authorization: PEXELS_API_KEY,
    }
  });
  const data = await result.json();

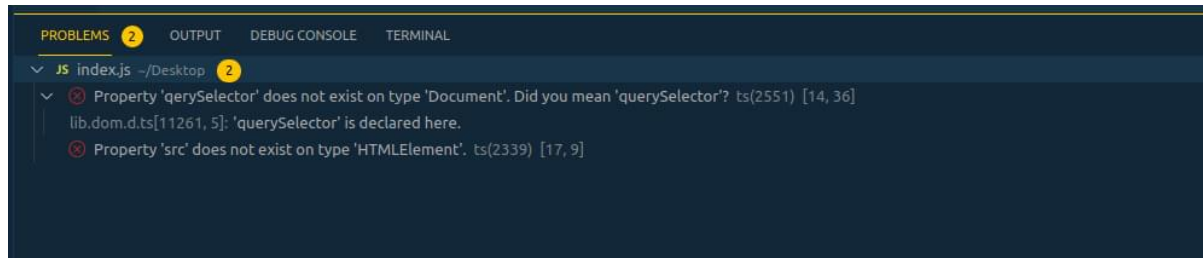
  const imagesContainer = document.gerySelector('#images-container');
  for (const photo of data.photos) {
    const img = document.createElement('image');
    img.src = photo.src.medium;
    imagesContainer.append(img);
  }
}

fetchImages('dogs', 5);
fetchImages(5, 'cats');
fetchImages('puppies');
```

1-1. TypeScript showing errors in VS Code

You should also see a cross in the bottom left-hand corner with a two by it. Clicking on this will

reveal the problems that have been spotted.



1-2. Errors displayed in the VS Code console

And just because you're not on VS Code doesn't mean you can't get the same experience with TypeScript highlighting errors. Most editors these days support the Language Server Protocol (commonly referred to as LSP), which is what VS Code uses to power its TypeScript integration.

It's well worth searching online to find your editor and the recommended plugins to have it set up.

Installing and Running TypeScript Locally

If you're not on VS Code, or you'd like a general solution, you can also run TypeScript on the command line. In this section, I'll show you how.

First, let's generate a new project. This step assumes you have [Node and npm installed upon your machine](#):

```
mkdir typescript-demo
cd typescript demo
npm init -y
```

Next, add TypeScript to your project:

```
npm install --save-dev typescript
```

Note: you could install TypeScript globally on your machine, but I like to install it per-project. That way, I ensure I have control over exactly which version of TypeScript each project uses. This is useful if you have a project you've not touched for a while; you can keep using an older TS version on that project, whilst having a newer project using a newer version.

Once it's installed, you can run the TypeScript compiler (`tsc`) to get the same errors (don't worry about these extra flags, as we'll talk more about them shortly):

```

npx tsc index.js --allowJs --noEmit --target es2015
index.js:13:36 - error TS2551: Property 'qerySelector' does not exist on type 'Document'. Did you mean 'querySelector'?
13   const imagesContainer = document.qerySelector('#images-container');
                                     ~~~~~
node_modules/typescript/lib/lib.dom.d.ts:11261:5
11261     querySelector<K extends keyof HTMLElementTagNameMap>(selectors: K): HTMLElementTagNameMap[K] | null;
       ~~~~~
'querySelector' is declared here.
index.js:16:9 - error TS2339: Property 'src' does not exist on type 'HTMLElement'.
16     img.src = photo.src.medium;
     ~~~
Found 2 errors.

```

You can see that TypeScript on the command line highlights the same JavaScript code errors that VS Code highlighted in the screenshot above.

Fixing the Errors in Our JavaScript Code

Now that we have TypeScript up and running, let's look at how we can understand and then rectify the errors that TypeScript is flagging.

Let's take a look at our first error.

Property `querySelector` does not exist on type `Document`

```

index.js:13:36 - error TS2551: Property 'qerySelector' does not exist on type 'Document'. Did you mean 'querySelector'?
13   const imagesContainer = document.qerySelector('#images-container');
node_modules/typescript/lib/lib.dom.d.ts:11261:5
11261     querySelector<K extends keyof HTMLElementTagNameMap>(selectors: K): HTMLElementTagNameMap[K] | null;
       ~~~~~
'querySelector' is declared here.

```

This can look quite overwhelming if you're not used to reading TypeScript errors, so don't panic if it looks a bit odd! TypeScript has spotted that, on line `13`, we've called a method `document.qerySelector`. We meant `document.querySelector` but made a mistake when typing. We would have found this out when we tried to run our code in the browser, but TypeScript is able to make us aware of it sooner.

The next part where it highlights `lib.dom.d.ts` and the `querySelector<K...>` function is diving into more advanced TypeScript code, so don't worry about that yet, but at a high level it's TypeScript showing us that it understands that there's a method called `querySelector`, and it suspects we might have wanted that.

Let's now zoom in on the last part of the error message above:

```
index.js:13:36 - error TS2551: Property 'querySelector' does not exist on type 'Document'. Did you mean 'querySele
```

Specifically, I want to look at the text `did not exist on type 'Document'`. In TypeScript (and broadly in every typed language), items have what's called a `type`.

In TypeScript, numbers like `1` or `2.5` have the type `number`, strings like `"hello world"` have the type `string`, and an instance of an HTML Element has the type `HTMLElement`. This is what enables TypeScript's compiler to check that our code is sound. Once it knows the type of something, it knows what functions you can call that take that something, or what methods exist on it.

Note: if you'd like to learn more about data types, please consult [‘Introduction to Data Types: Static, Dynamic, Strong & Weak’](#).

In our code, TypeScript has seen that we've referred to `document`. This is a global variable in the browser, and TypeScript knows that and knows that it has the type of `Document`. This type documents (if you pardon the pun!) all of the methods we can call. This is why TypeScript knows that `querySelector` is a method, and that the misspelled `qerySelector` is not.

We'll see more of these types as we go through the later chapters, but this is where all of TypeScript's power comes from. Soon we'll define our own types, meaning really we can extend the type system to have knowledge about all of our code and what we can and can't do with any particular object in our codebase.

Now let's turn our attention to our next error, which is slightly less clear.

Property `src` does not exist on type `HTMLElement`

```
index.js:16:9 - error TS2339: Property 'src' does not exist on type 'HTMLElement'.  
16     img.src = photo.src.medium;
```

This is one of those errors where sometimes you have to look slightly above the error to find the problem. We know that an HTML image element does have a `src` attribute, so why doesn't TypeScript?

```
const img = document.createElement('image');  
img.src = photo.src.medium;
```

The mistake here is on the first line: when you create a new image element, you have to call `document.createElement('img')` (because the HTML tag is ``, not `<image>`). Once we do that, the error goes away, because TypeScript knows that, when you call `document.createElement('img')`, you get back an element that has a `src` property. And this is all down to the types.

When you call `document.createElement('div')`, the object returned is of the type `HTMLDivElement`. When you call `document.createElement('img')`, the object returned is of type `HTMLImageElement`. `HTMLImageElement` has a `src` property declared on it, so TypeScript knows you can call `img.src`. But `HTMLDivElement` doesn't, so TypeScript will error.

In the case of `document.createElement('image')`, because TypeScript doesn't know about any HTML element with the tag `image`, it will return an object of type `HTMLElement` (a generic HTML element, not specific to one tag), which also lacks the `src` property.

Once we fix those two mistakes and re-run TypeScript, you'll see we get back nothing, which shows that there were no errors. If you've configured your editor to show errors, hopefully there are now none showing.

How to Configure TypeScript

It's a bit of a pain to have to add `// @ts-check` to each file, and when we run the command in the terminal having to add those extra flags. TypeScript lets you instead enable it on a JavaScript project by creating a `tsconfig.json` file.

Create `tsconfig.json` in the root directory of our project and place this inside it:

```
{
  "compilerOptions": {
    "checkJs": true,
    "noEmit": true,
    "target": "es2015"
  },
  "include": ["*.js"]
}
```

This configures the TypeScript compiler (and your editor's TS integration) to:

- 1 Check JavaScript files (the `checkJs` option).
- 2 Assume we're building in an ES2015 environment (the `target` option). Defaulting to

ES2015 means we can use things like promises without TypeScript giving us errors.

- 3 Not output any compiled files (the `noEmit` option). When you're writing TypeScript code in TypeScript source files, you need the compiler to generate JavaScript code for you to run in the browser. As we're writing JavaScript code that's running in the browser, we don't need the compiler to generate any files for us.
- 4 Finally, `include: ["*.js"]` instructs TypeScript to look at any JavaScript file in the root directory.

Now that we have this file, you can update your command-line instruction to this:

```
npx tsc -p jsconfig.json
```

This will run the compiler with our configuration file (the `-p` here is short for "project"), so you no longer need to pass all those flags through when running TypeScript.

Working in strict mode

Now we're here, let's see how we can make TypeScript even more thorough when checking our code. TypeScript supports something called "strict mode", which instructs TypeScript to check our code more thoroughly and ensure that we deal with any potential times where, for example, an object might be `undefined`. To make this clearer, let's turn it on and see what errors we get. Add `"strict": true` to the `"compilerOptions"` part of `jsconfig.json`, and then re-run TypeScript on the command line.

When you make a change to the `jsconfig.json` file, you may find you need to restart your editor for it to pick up those changes. So if you're not seeing the same errors as me, give that a go.

```
npx tsc -p jsconfig.json
index.js:3:28 - error TS7006: Parameter 'searchTerm' implicitly has an 'any' type.
3 async function fetchImages(searchTerm, perPage) {
      ~~~~~

index.js:3:40 - error TS7006: Parameter 'perPage' implicitly has an 'any' type.
3 async function fetchImages(searchTerm, perPage) {
      ~~~~~

index.js:15:5 - error TS2531: Object is possibly 'null'.
15   imagesContainer.append(img);
     ~~~~~

Found 3 errors.
```


Let's start with the last error first and come back to the others:

```
index.js:15:5 - error TS2531: Object is possibly 'null'.
15     imagesContainer.append(img);
    ~~~~~
```

And let's look at how `imagesContainer` is defined:

```
const imagesContainer = document.querySelector('#images-container');
```

Turning on `strict` mode has made TypeScript stricter at ensuring that values we expect to exist do exist. In this case, it's not guaranteed that `document.querySelector('#images-container')` will actually return an element; what if it's not found? `document.querySelector` will return `null` if an element is not found, and now we've enabled strict mode, TypeScript is telling us that `imagesContainer` might actually be `null`.

Union Types

Prior to turning on strict mode, the type of `imagesContainer` was `Element`, but now we've turned on strict mode the type of `imagesContainer` is `Element | null`. The `|` (pipe) operator creates union types — which you can read as "or" — so here `imagesContainer` is of type `Element` or `null`. When TypeScript says to us `Object is possibly 'null'`, that's exactly what it's telling us, and it wants us to ensure that the object does exist before we use it. Let's fix this by throwing an error should we not find the images container element:

```
const imagesContainer = document.querySelector('#images-container');
if (imagesContainer === null) {
  throw new Error('Could not find images-container element.')
}
for (const photo of data.photos) {
  const img = document.createElement('img');
  img.src = photo.src.medium;
  imagesContainer.append(img);
}
```

TypeScript is now happy; we've dealt with the `null` case by throwing an error. TypeScript is smart enough to understand now that, should our code not throw an error on the third line in the above snippet, `imagesContainer` is not `null`, and therefore must exist and must be of type `Element`. Its type was `Element | null`, but if it was `null` we would have thrown an error, so now it must be `Element`. This functionality is known as type narrowing and is a very useful concept to be aware of.

Implicit any

Now let's turn our attention to the remaining two errors we have:

```
index.js:3:28 - error TS7006: Parameter 'searchTerm' implicitly has an 'any' type.
3 async function fetchImages(searchTerm, perPage) {
    ~~~~~
index.js:3:40 - error TS7006: Parameter 'perPage' implicitly has an 'any' type.
3 async function fetchImages(searchTerm, perPage) {
```

One of the implications of turning on strict mode is that it turns on a rule called `noImplicitAny`. By default, when TypeScript doesn't know the type of something, it will default to giving it a special TypeScript type called `any`. `any` is not a great type to have in your code, because there are no rules associated with it in terms of what the compiler will check. It will allow anything to happen. I like to picture it as the compiler throwing its hands up in the air and saying "I can't help you here!" Using `any` disables any useful type checking for that particular variable, so I highly recommend avoiding it.

Describe the Function Signature with JSDoc

The two errors above are TypeScript telling us that we've not told it what types the two variables our function takes are, and that it's defaulting them back to `any`. The good news is that giving TypeScript this information used to mean rewriting your file into TypeScript code, but TypeScript now supports a hefty subset of [JSDoc syntax](#), which lets you provide type information to TypeScript via JavaScript comments. For example, here's how we can provide type information to our `fetchImages` function:

```
/**
 * @param {string} searchTerm
 * @param {number} perPage
 *
 * @return void
 */
async function fetchImages(searchTerm, perPage) {
  // function body here
}
```

All JSDoc comments must start with `/**` (note the extra `*` at the beginning) and within them we use special tags, starting with `@`, to denote type properties. Here we declare two parameters (`@param`), and then we put their type in curly braces (just like regular JavaScript objects). Here we make it clear that `searchTerm` is a `string` and `perPage` is a number. While we're at it, we also use `@return` to declare what this function returns. In our case it returns nothing, and the type we

use in TypeScript to declare that is `void`. Let's now re-run the compiler and see what it says:

```
npx tsc -p jsconfig.json
index.js:30:13 - error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.
30 fetchImages(5, 'cats')
    ~
index.js:31:1 - error TS2554: Expected 2 arguments, but got 1.
31 fetchImages('puppies')
    ~~~~~~
index.js:9:40
   9 async function fetchImages(searchTerm, perPage) {
     ~~~~~~
       An argument for 'perPage' was not provided.
Found 2 errors.
```

This is the beauty of TypeScript. Giving the compiler extra information, it can now spot errors in how we're calling the code that it couldn't before. In this case, it's found two calls to `fetchImages` where we've got the arguments in the wrong order, and the second where we've forgotten the `perPage` argument (neither `searchTerm`, `perPage` are optional parameters). Let's just delete these calls, but I hope it helps demonstrate the power of the compiler and the benefits of giving the compiler extra type information.

Declaring Data Types Using an Interface

Although not flagged by the compiler, one issue our code still has is in this line:

```
const data = await result.json();
```

The problem here is that the return type of `await result.json()` is `any`. This is because, when you take an API response and convert it into JSON, TypeScript has no idea what data is in there, so it defaults to `any`. But because we know what the Pexels API returns, we can give it some type information by using [TypeScript interfaces](#). These let us tell TypeScript about the *shape* of an object: what properties it has, and what values those properties have. Let's declare an interface — again, using JSDoc syntax, that represents the data returned from the Pexels API. I used [the Pexels API reference](#) to figure out what data is returned. In this case, we'll actually define two interfaces: one will declare the shape of a single `photo` that the Pexels API returns, and the other will declare the overall shape of the response from the API. To define these interfaces using JSDoc, we use `@typedef`, which lets us declare more complex types. We then use `@property` to declare single properties on that interface. For example, here's the type I create for an individual `Photo`. Types should always start with a capital letter. *If you'd like to see a full reference to all supported JSDoc functionality, the [TypeScript site has a thorough list complete with examples](#).*

```
/**
 * @typedef {Object} Photo
 * @property {{medium: string, large: string, thumbnail: string}} src
 */
```

This type says that any object typed as a `Photo` will have one property, `src`, which itself is an object with three string properties: `medium`, `large` and `thumbnail`. You'll notice that the Pexels API returns more; you don't have to declare every property an object has if you don't want to, but just the subset you need. Here, our app currently only uses the `medium` image, but I've declared a couple of extra sizes we might want in the future. Now that we have that type, we can declare the type `PexelsSearchResponse`, which will represent what we get back from the API:

```
/**
 * @typedef {Object} PexelsSearchResponse
 * @property {Array<Photo>} photos
 */
```

This is where you can see the value of declaring your own types; we declare that this object has one property, `photos`, and then declare that its value is an array, where each item is of type `Photo`. That's what the `Array<X>` syntax denotes: it's an array where each item in the array is of type `X`. `[1, 2, 3]` would be an `Array<number>`, for example. Once we've done that, we can then use the `@type` JSDoc comment to tell TypeScript that the data we get back from `result.json()` is of the type `PexelsSearchResponse`:

```
/** @type {PexelsSearchResponse} */
const data = await result.json();
```

`@type` isn't something you should reach for all the time. Normally, you want the compiler to intelligently figure out the type of things, rather than have to bluntly tell it. But because `result.json()` returns `any`, we're good here to override that with our type.

Test if everything is working

To prove that this is working, I've deliberately misspelled `medium` when referencing the photo's URL:

```
for (const photo of data.photos) {
  const img = document.createElement('img');
  img.src = photo.src.mediun; // typo!
  imagesContainer.append(img);
}
```

If we run TypeScript again, we'll see the issue that TypeScript wouldn't have spotted if we hadn't done the work we just did to declare the interface:

```
index.js:35:25 - error TS2551: Property 'mediun' does not exist on type '{ medium: string; large: string; thumbnail: string; }'.
35     img.src = photo.src.mediun;
                               ~~~~~

index.js:18:18
18     * @property {{medium: string, large: string, thumbnail: string}} src
                   ~~~~~
'medium' is declared here.
Found 1 error.
```

Conclusion

TypeScript has a lot to offer developers working on complicated codebases. Its ability to shorten the feedback loop and show you errors *before* you have to recompile and load up the browser is really valuable. We've seen how it can be used on any existing JavaScript project (avoiding the need to rewrite your code into `.ts` files) and how easy it is to get started. I hope you've enjoyed this TypeScript tutorial for beginners. In the next chapters, we'll start putting this knowledge into action and build out a fully blown app using TypeScript.

Build an Application with TypeScript from Scratch

Jack Franklin

Chapter

2

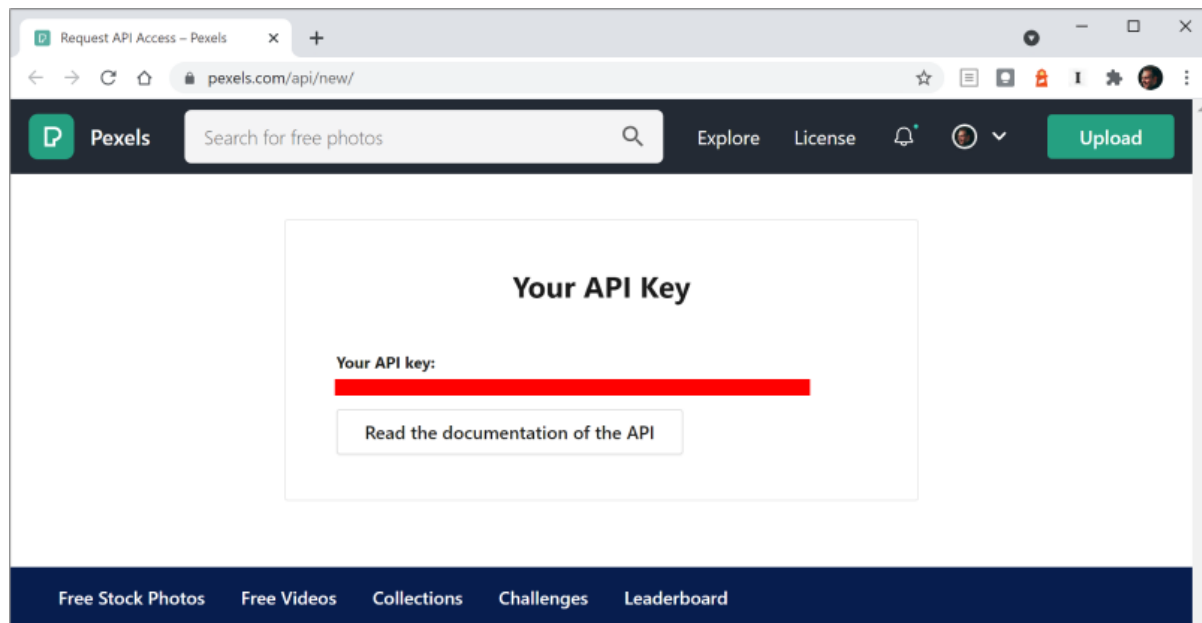
In the previous chapter, we explored and demonstrated the power of TypeScript by adding it to some existing JavaScript code, and we looked at the types of issues it could spot and prevent. In this tutorial, we'll take the next step and look at building our application from scratch using TypeScript files and taking full advantage of the TypeScript ecosystem.

Pexels API Key

As in the previous tutorial, we'll be making use of the [Pexels API](#) to fetch photos to build our little photo viewing tool. To do this, you'll need to get yourself an API key (which is entirely free, with no payment details required) from Pexels. To get an API key you should:

- 1 [Sign up to Pexels](#) to create your account.
- 2 Use the [new API key page](#) to request a new API key. This should happen instantly; you won't have to wait any time before getting the key back.

Once that's done, you should be taken to a page that looks like the image below.



2-1. The Pexels API key website

(I've removed my API key from the above screenshot, but where the red bar is you'll see your API key.)

Once you've got that, you're all set! We'll look at how to use the API key when making requests

later on in the tutorial.

Setting up Vite as the Build Tool

For this tutorial, we're going to use [Vite](#) as our basic build tool. (You can find a thorough introduction to Vite here in "[What is Vitejs? An Overview of the New Front-end Build Tool](#)".) Vite is performant and easy to set up. It does a great job of getting out of the way and letting you focus on building your application. It also has a TypeScript configuration option so there's even less work for us to do; it will generate a project configured with TypeScript right out of the box.

To generate a new application, run the following:

```
npm init vite@latest
```

If you use Yarn, pnpm, or other npm alternatives, check the [Vite guide](#) for more information on using Vite with those tools.

Vite will then prompt you with questions to answer. The first will be what framework to use. Make sure you pick "vanilla" here, as we're not going to use a particular framework in this tutorial. That isn't to say you can't use a framework like React with TypeScript. You definitely can, and I'd encourage you to explore TypeScript along with your framework of choice in a project once you're done with this series of tutorials.

After you pick "vanilla", you'll be asked to pick a variant. This is where you want to pick the "vanilla-ts" option to configure a TypeScript project. If you don't pick this, Vite will assume you want JavaScript.


```
? Select a framework:
> vanilla
vue
react
preact
lit-element
svelte
```

2-2. Vite prompting us to pick a framework

```
✓ Select a framework: vanilla
? Select a variant:
  vanilla
> vanilla-ts
```

2-3. Vite prompting us to pick vanilla or vanilla-ts to use TypeScript.

Installing Dependencies

Now Vite has generated the new project, we need to do an `npm install` (or the equivalent in your tool of choice if you're not using npm). To do this, navigate to the directory Vite created and run `npm install`. This will configure the project and install TypeScript and all the related dependencies we need.

Configuring TypeScript with Vite

Vite generates a project that has TypeScript configured. It configures TypeScript via a `tsconfig.json` file in the root directory of the project. This is the standard location and filename for a TypeScript project; it's very rare for any TypeScript codebase not to have this file defined here.

Vite's default TypeScript configuration will serve us pretty well, so we'll leave it alone for now. The [TypeScript tsconfig reference](#) is a great place to look should you want to understand what a particular setting does. The main thing that Vite does turn on is `strict` mode, which, as we discovered in the previous tutorial, is great for ensuring TypeScript is as thorough and strict as possible when it comes to checking our code for potential issues.

Vite also turns on other settings that make TypeScript more strict about the type of code it allows:

- `noUnusedLocals`, which will report an error if we create a variable that is then not used.
- `noUnusedParameters`, which will error if we create a function that takes a parameter that is not used.
- `noImplicitReturns`, which ensures that, if we declare that a function must return an item of a specific type, all code paths through that function return the right type. This is useful if you've said your function returns a `string`, but inside that function you have an `if {} else {}` statement. TypeScript will check that both the `if` and the `else` branch return a `string`.

The final configuration of node is `"include": ["/src"]`. The `include` key is used to tell TypeScript which folder(s) it should check for TypeScript files. It's common (and recommended) to put all your TypeScript code into one folder (which can have nested folders within) to keep your project tidy and to ensure the TypeScript compiler isn't accidentally checking files and folders that don't contain any actual TypeScript. For our application, we'll stick with Vite's out-of-the-box defaults; all our code will go into the `src` directory.

Building the Project with Vite

`npm run dev` is your main command when working on a site backed by Vite. It will build your application (including running TypeScript) and run it on a local server. Vite also rebuilds automatically when files change and will refresh the browser. It's a great developer workflow.

Although we've generated our application with Vite, you can still run the TypeScript compiler just like we did in the last tutorial:

```
npx tsc -p tsconfig.json
```

One extra tip: you can omit the `-p tsconfig.json` if your configuration file is called `tsconfig.json` and is in the root directory. `npx tsc` will run the compiler and will find your configuration file.

Making API Requests with TypeScript

In the previous tutorial, we wrote the `fetchImages` function that could make a request to the Pexels API and then insert those images into the DOM. We're now going to rewrite that code, starting with just the `fetch` part that will talk to the Pexels API and return a promise with some images.

In `src`, open `main.ts` and delete everything that's in there. Vite will have created some starter code for you that includes `main.ts` and `style.css`. For now, we'll be focusing purely on the TypeScript as we get this app up and running and won't worry about styling. Feel free to remove that file or leave it and do some of your own styling if you'd like. We'll write all our code in `src/main.ts`.

In `main.ts`, define `fetchImagesFromAPI`:

```
async function fetchImagesFromAPI(searchTerm: string, perPage: number) {  
  
}
```

We'll make the function `async` so we can use `await` within the function. ([This article on flow control in JavaScript](#) is a great reference for all the ways we can write asynchronous code in JavaScript.) Using `async` and `await` makes it easier to write our async code to deal with fetching images from the API.

In the previous tutorial, we were writing JavaScript and therefore had to use the JSDoc comments syntax to add type information to our code. Now we're writing in a TypeScript file, we can use TypeScript syntax to add type information. When defining function parameters, we can add type information to those parameters by adding a colon and then the type. We'll meet more TypeScript syntax as we go through the rest of this tutorial.

Now, within this function we can write the code to make the API request. To authenticate, we pass an `Authorization` header that contains the API key that we got earlier, as described in [the Pexels API docs](#). I've defined a constant `PEXELS_API_KEY` which is set to my API key. Make sure you do the same with your API key:

```

const PEXELS_API_KEY = '...api key here...';

async function fetchImagesFromAPI(searchTerm: string, perPage: number) {
  const result = await fetch(
    `https://api.pexels.com/v1/search?query=${searchTerm}&per_page=${perPage}`,
    {
      headers: {
        Authorization: PEXELS_API_KEY,
      },
    }
  );

  const json = await result.json();
  return json;
}

```

This function is enough to get started. We can call this and it will fetch us data from the Pexels API, but there's much more we can do from a TypeScript point of view to improve its type safety. You can see this in action [in this CodeSandbox](#) — but you'll first need to update the code with your API key to make the Pexels API authorize the requests. If you don't update the code with your API key, you'll see CodeSandbox show a "Type Error - failed to fetch".

Inferring types

When we define a variable in TypeScript, we can explicitly tell the compiler the type of the variable we're creating:

```

const x: number = 5;
const y: string = "hello world";

```

Much like how we declared the type of function parameters, when creating variables we use the `: type` syntax after the variable name to tell TypeScript what type the value is. However, often we don't need to do that, because TypeScript has *type inference*. This means that the TypeScript compiler will try to infer the type of a variable given the rest of the code it has available to it. It will also take into account the type of the variable: whether it's a constant, and therefore cannot change, or a `Let`, that could change.

Take a look at this [CodeSandbox](#) and, using your mouse, hover over the variable declarations for `x` and `y`.

When you hover over `x`, you'll see that TypeScript shows you `const x: 5`. This means that the type of `x` that TypeScript has inferred is the literal number value `5`. It knows that because `x` is a `const`, its value cannot change, so rather than give it a generic type like `number`, it can be even

more specific and pin the value down to the number `5`. If you hover over `y`, you'll see `Let y: number`. Here, TypeScript has recognized that initially `y` was set to `2`, but because it's declared using `let` its value can change over time, so it's gone with the type `number`. This is why often you'll see TypeScript developers omit the explicit type annotation:

```
// Often people will omit the type
let y: number = 2;
// Because TS can be trusted to infer the type accurately.
let y = 2;
```

I typically will let TypeScript infer types for primitive values — for example, strings, numbers or Booleans — but I like to explicitly declare types for values that are objects or arrays.

You can declare something to be an array in one of two ways:

```
const names: Array<string> = ["alice", "bob"];
const names: string[] = ["alice", "bob"];
```

These are both equivalent, and the choice comes down to personal preference. Again, if the array is of a primitive type, I'll tend to prefer the `string[]` approach, but I don't personally have hard rules for this. You might always want to use `string[]`, and that's perfectly OK. You'll get a feel for your preferred approach as you write and work with TypeScript more.

Interfaces in TypeScript

Let's start to work on improving our function. TypeScript will also infer the return type of functions, so let's see what it's inferred for `fetchImagesFromAPI`. In most editors, you can hover over the function definition in your editor to see what TypeScript thinks the function will return. If you can't hover, most editors that integrate with TypeScript will provide this functionally in their UI somehow; it's worth checking the documentation for your editor. This is definitely something you're going to want to do often, so spending a few minutes now learning how to do it in your editor will be beneficial.

In VS Code, I'm able to hover and this is what's pictured below.

```
main.ts
const PEXELS_API_KEY = 'not-my-real-api-key';
function fetchImagesFromAPI(searchTerm: string, perPage: number): Promise<any>
async function fetchImagesFromAPI(searchTerm: string, perPage: number) {
```

2-4. TypeScript inferring the function return type as Promise

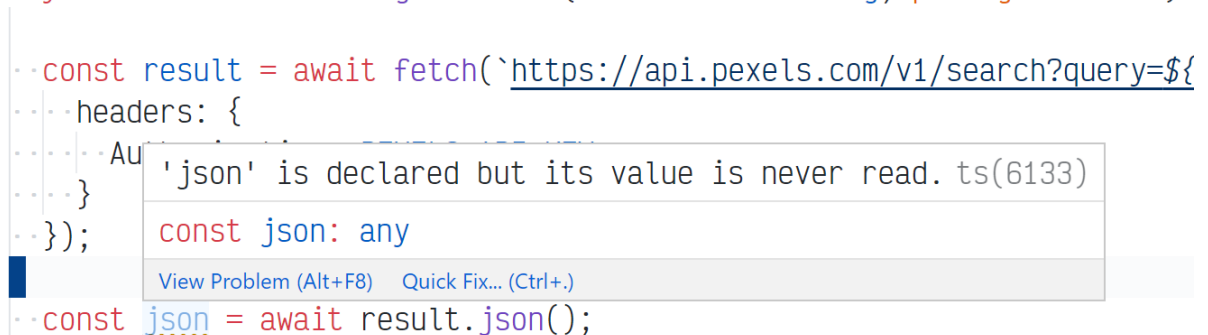
TypeScript has decided that our function returns `Promise<any>`. You can read this syntax as “a

promise that will resolve with data of type `any`” — much like how `Array<string>` can be read as “an array that contains `string` values”. We’ll look more at this `Foo<bar>` syntax shortly.

We touched upon it in the previous tutorial, but `any` is not a good type to have floating around your type system. `any` denotes a complete lack of type safety, and the compiler here is telling us that it effectively has no information on the type of data that our promise will resolve to. But why is this?

Let’s take a look at the return type of `await result.json()`.

```
async function fetchImagesFromAPI(searchTerm: string, perPage: number) {
  const result = await fetch(`https://api.pexels.com/v1/search?query=${searchTerm}&per_page=${perPage}`, {
    headers: {
      'Authorization': `Bearer ${API_KEY}`
    }
  });
  const json = await result.json();
}
```



2-5. TypeScript showing that `result.json()` returns `any`

The issue here stems from TypeScript (understandably) not being able to have any confidence in what `result.json()` will return. This makes sense, as it doesn’t know our API, or what data is coming back. Remember that TypeScript runs at *compile time*, so it has to figure out everything without running any code. That makes it impossible for it to know what data is coming back from the API request. But we know, and so we could tell it what types we expect. But first, we need to define those types.

The [Pexels API](#) documents what data a `Photo` has associated with it, so what we’ll do is create an interface that represents that photo. Place this code towards the top of `src/main.ts`. The exact location doesn’t matter, but I tend to prefer to put interfaces towards the top of the file. Feel free to place it where you’d like, as long as it’s in `src/main.ts`:

```
interface Photo {
  id: number;
  width: number;
  height: number;
  url: string;
  photographer: string;
}
```

```
photographer_url: string;
photographer_id: string;
avg_color: string;
src: {
  original: string;
  large2x: string;
  large: string;
  medium: string;
  small: string;
  portrait: string;
  landscape: string;
  tiny: string;
};
}
```

In TypeScript, an interface is the primary way to describe a JavaScript object and what properties and/or methods it has available to it. When you define an object and declare the interface that it follows, TypeScript will ensure that the object is valid and implements that interface. For example, this code would error:

```
interface Person {
  name: string;
}

const jack: Person = {
  myName: 'Jack',
}
```

This is because the object has `myName`, and doesn't have the `name` property. This code is valid:

```
interface Person {
  name: string;
}

const jack: Person = {
  name: 'Jack',
}
```

Interfaces in TypeScript have to start with a capital letter, and you should name them based on what they represent. In our case this interface is fairly straightforward; most of the keys are either `string` or `number` types. Note that the value of the `src` key is an object. We could have made this object into its own interface if we wanted to. It's up to you when you use objects in interfaces like we did here, or if you want to create separate interfaces and have one interface reference another. In this case because the object under the `src` key doesn't really make sense outside of the `Photo` interface, I've left it as part of the `Photo` interface.

Now that we have this type, we can tell TypeScript that `result.json()` is going to give us back an array of these interfaces. However, the Pexels API actually returns some other information along with the request, [as we can see by the documentation for the photo search API](#), so let's define an interface for that. Define this one just below where you placed the definition for the `Photo` interface in `src/main.ts`:

```
interface PhotoSearchAPIResult {
  total_results: number;
  page: number;
  per_page: number;
  photos: Photo[];
  next_page: string;
}
```

Here, you can see how one interface can reference another. The `photos` key is an array of objects where each object is of type `Photo`. There's two ways in TypeScript to denote this:

- `Photo[]`
- `Array<Photo>`

Both are functionally equivalent, so it's up to you which one you'd prefer. Most teams will have a preference and stick with it. Later on in this tutorial, we'll see how we can use ESLint to enforce a particular style when we set up `typescript-eslint`.

Now we need to tell TypeScript that `await result.json()` returns our `PhotoSearchAPIResult` type. We can do this using [type assertions](#), which exist as a way of telling the compiler that we know what the given type of something is.

We do this like so:

```
const json = (await result.json()) as PhotoSearchAPIResult;
```

Note: the extra brackets around `await result.json()` are important here. Without them, it's not clear if we're asserting the result of `await result.json()` or the result of `result.json()` (which would be a `Promise`) — so the extra brackets make it clear that we're asserting the result of `await result.json()`.

You should be very careful with using `as`. Ultimately `as` is telling the compiler that we know better than it. There are no checks when you run this code in the browser that the type you nominated is actually correct. To try to prevent against conversions that are impossible, TypeScript won't let you do conversions that don't make any sense:


```
const x = "hello" as number; // Causes an error.
```

In our case, `await result.json()` returns the type `any`, which is really the compiler saying it has no idea, so it's happy for us to assert a more specific type. But try to limit how often you use this. You'll see others use it liberally in codebases, but if you use it and assert the wrong type, the compiler will be happy and you'll end up with errors when your code actually runs in the browser. You should view `as X` as an escape hatch that you can use if you really have to, but try to not routinely rely on it.

Declaring Function Return Types

Now that we've got the data back from the API, and we have it typed as `PhotoSearchAPIResult`, we can now declare the return type of our function. To declare a return type, you add a colon after the function arguments, and then the type:

```
// Returns an object of type Bar
function foo1(): Bar {
};

// Returns a number
function foo2(): number {
};

// Returns an object with a key `x` of type `number`
function foo3(): {x: number} {
};
```

Given that, you might be tempted to update the `fetchImagesFromAPI` function like so (I've split the function definition over multiple lines because it's a lot to fit onto one line once you have the types in here):

```
async function fetchImagesFromAPI(
  searchTerm: string,
  perPage: number
): PhotoSearchAPIResult {
  // ...
}
```

However, TypeScript isn't happy, as VS Code will show you.

```
The return type of an async function or method must be the global Promise<T> type. Did you mean to write 'Promise<PhotoSearchAPIResult>'? ts(1064)
interface PhotoSearchAPIResult
PhotoSearchAPIResult {
const result = await fetch(
```

2-6. VS Code erroring with the custom return type

You can also see this error by running `npx tsc` :

```
src/main.ts:36:4 - error TS1064:
The return type of an async function or method must be the global Promise<T> type.
Did you mean to write 'Promise<PhotoSearchAPIResult>'?

36 ): PhotoSearchAPIResult {
    ~~~~~
```

And if you'd like to play with the code itself, you can see the same error in [this CodeSandbox](#). *Don't forget to update your API key if you want the requests to work.*

To fully understand this, we need to talk about what TypeScript means and what is represented by the `Promise<T>` syntax. At that means talking about generics.

Generic Types

Some types need to be reusable and apply to multiple situations. Therefore, we need to be allowed to define types that have some flexibility in what they represent. Take JavaScript arrays as an example. We might have arrays of numbers, which we'd represent with `Array<number>` . (Or alternatively, `number[]` , but we'll use `Array<>` syntax for this example to be consistent. The `Array<T>` syntax also matches the `Promise<T>` syntax.) If we had an array of strings, we would represent that with the type `Array<string>` .

For every possible type we have in the world, we can have an array of that type. So TypeScript can't possibly define a separate array type for every type. If TypeScript provided `NumberArray` and `StringArray` , that would work for primitives, but what if we'd like an array for our `Photo` type? We need a general type that can be reused to represent arrays of some type, and that's why generics exist.

So when we create `Array<string>` , or `Array<number>` , or `Array<Photo>` , what we are doing more generally is creating arrays of type `Array<T>` , where `T` is a placeholder for any type we

want, to create an array where each item in that array is of type `T`. So when we create `Array<string>`, we're replacing the type `T` with `string`.

We can see this in action in this [TypeScript Playground example](#). Each time I create a new array, TypeScript is looking at the contents of that array and using it to replace the generic type argument `T` with the type that I've passed it. We can also see in this example how `Array<string>` and `string[]` are equivalent.

Go back now and look at the error TypeScript gave us:

```
The return type of an async function or method must be the global Promise<T> type.
Did you mean to write 'Promise<PhotoSearchAPIResult>'?
```

We can see the `Promise<T>` syntax, which is identical to the `Array<T>` syntax. `Promise<T>` is how TypeScript represents promises. Here, the type `T` is used to represent the type that a promise resolves to:

```
somePromise().then(data => {
  // the type of data is represented by the T in Promise<T>
})
```

This TypeScript error boils down to the fact that we've created our function as an `async` function:

```
async function fetchImagesFromAPI(...) {}
```

When you define a function with the `async` keyword, it will always return a promise. Even if you don't explicitly return a promise, JavaScript will wrap what you return in a promise. Again, [this article on flow control in JavaScript](#) is a good reference if you need a refresher on `async` and `await`.

Once we fix our return type to wrap our `PhotoSearchAPIResult` in a `Promise`, TypeScript no longer errors, and if we call the code and log the result, you can see that TypeScript has correctly figured out the type of our `data` argument.

```
fetchImagesFromAPI('dogs', 5).then((data) => {
  console.log(data);
});
```

2-7. TypeScript understanding the promise type correctly

And our code now looks like this:

```
async function fetchImagesFromAPI(  
  searchTerm: string,  
  perPage: number  
): Promise<PhotoSearchAPIResult> {  
  // ...  
}
```

You don't have to always explicitly declare the return type of a function. As mentioned previously, TypeScript can often *infer* types from your code. So in our case, TypeScript would be able to read through our code and determine the return type of `fetchImagesFromAPI` correctly. I find that, nearly all the time, it's worth explicitly declaring the return types of your functions, for two reasons:

- It helps keep your code readable and explicit.
- It prevents you returning something other than what you expect. If you don't declare a return type and what TypeScript infers isn't what you intended to use, you might have a bug. If you explicitly declare a return type and then return the wrong type from the function, TypeScript will error.

In the last part of this tutorial, we'll see how we can ensure that functions always have return types declared explicitly by configuring TypeScript-ESLint.

Using Third-party Libraries with TypeScript


We've spent a lot of time fetching data from our API, but as of yet we've not got anything rendered on the screen! Let's fix that. To help us render to the page, we're going to use a small library called [lit-html](#).

TypeScript can and does work with any frameworks: Angular, React, Vue, Svelte, and so on all, have good integrations with TypeScript. I don't want us to get bogged down with specific frameworks in this tutorial, so we're going to use lit-html, as it's a very small library that helps with rendering HTML easily onto the page. Please feel free to swap this out for any framework of your choice.

If we [look up the lit-html package on npm](#), you'll notice the blue TypeScript icon by the package name on the npm website.

lit-html

2.0.0-rc.5 • Public • Published 4 days ago

 [Readme](#)

 [Explore](#) BETA

lit-html 2.0 Release Candidate

2-8. npm package site showing a TypeScript icon for the lit-html package

Note: at the time of writing, lit-html v2 is in release candidate status and therefore not considered fully released. Despite that, we're going to use the v2 beta, as it contains a number of changes from v1, and doing so will ensure this tutorial isn't outdated almost as soon as it's released.

The blue TS icon means that the package ships with built-in type definitions out of the box. This means that either the library is written in TypeScript, or the authors provide a types file containing all the type information for lit-html. This is important, because it means that, when you use this library, TypeScript is able to understand it and will know the types of functions as you use them.

If you come across a library that you want to use that doesn't have TypeScript support, that doesn't mean you can't use it, but does mean you might need to do a bit of extra work to integrate it into your codebase. In the next tutorial, we'll explore just how to do that.

Let's install Lit. I'm installing `lit-html@next` to ensure we get the latest beta version (`2.0.0-rc.5` at the time of writing):

```
npm install --save lit-html@next
```

Note: if you're reading this at a time when lit-html 2 has been fully released, you can run `npm install lit-html` to get the latest version.

Using lit-html to Render Image Results

To use lit-html, we'll need to import two functions it provides: `render` and `html`. Place this

import statement at the top of `src/main.ts` :

```
import {render, html} from 'lit-html';
```

This is where it becomes clear why it matters that lit-html is TypeScript compatible: TypeScript will know the types of these functions that we've imported, so it will be able to check as you're using the library that you're doing so correctly.

To create HTML for lit-html to render, we call the HTML function and pass it a template string:

```
html`<p>this is my HTML</p>`
```

This will look familiar if you've used any front-end framework before.

We then pass that into `render`, which takes a second argument of the element to render into:

```
const div = document.getElementById('some-id');
render(
  html`<p>hello world</p>`,
  div
);
```

In our case, we'll loop over each image that we got back from the API, generate some HTML, and then insert it into a containing element. The `index.html` file that Vite generated when we created our project includes a `<div>` with an ID of `app` that we'll use.

Let's start by looping over each photo from the API and generating some HTML for it:

```
fetchImagesFromAPI('dogs', 5).then((data) => {
  const htmlToRender = html`
    <h1>Results for "dogs"</h1>
    <ul>
      ${data.photos.map((photo) => {
        return html`<li><img src=${photo.src.small} /></li>`;
      })}
    </ul>
  `;
});
```

Notice that, within the first `html` call, we can then use JavaScript interpolation to map over the photos and generate the `` and `` for each one. This is a nice feature of lit-html: it's easy to programmatically generate big chunks of HTML.

Now we need to query the DOM for the `<div>` that we want to render into, and then pass it, along with `htmlToRender`, into the `render` function:

```
const div = document.getElementById('app');
render(htmlToRender, div);
```

However, if we run `npx tsc` now, we'll see a compiler error:

```
src/main.ts:62:24 - error TS2345: Argument of type 'HTMLElement | null' is not assignable to parameter of type 'HTMLElement | DocumentFragment'.
  Type 'null' is not assignable to type 'HTMLElement | DocumentFragment'.
```

```
62  render(htmlToRender, div);
```

As we recall from the last tutorial, TypeScript in strict mode needs us to ensure that an element we queried the DOM for definitely exists. So let's ensure it does by throwing an error if it doesn't:

```
fetchImagesFromAPI('dogs', 5).then((data) => {
  const htmlToRender = html`
    <h1>Results for "dogs"</h1>
    <ul>
      ${data.photos.map((photo) => {
        return html`<li><img src=${photo.src.small} /></li>`;
      })}
    </ul>
  `;
  const div = document.getElementById('app');
  if (!div) {
    throw new Error('could not find app div');
  }

  render(htmlToRender, div);
});
```

And now, once you rebuild the project (`npm run dev` if you're not running it already), you'll see some dogs on the page!

Results for "dogs"



2-9. Our project running and showing five pictures of dogs

And [here's the code running on CodeSandbox](#). Again, you'll need to update the code with your Pexels API key to get it up and running.

Linting with ESLint-TypeScript.

To round off this tutorial, let's get linting set up with ESLint. In the past, you had to use the separate tool TSLint in order to lint your code. But thankfully, ESLint now has TypeScript support via [typescript-eslint](#). This is what we'll set up to have linting enabled on our TypeScript code. The good news is that, because it's ESLint, if you've set that up before on a JavaScript project, a lot of these steps will be very familiar to you!

We're going to need to install a few different packages:

- `eslint` : although `typescript-eslint` provides the rules, we'll still run our linting via the main ESLint tool. This is great if you're working in a codebase with JS and TS files, as you can lint them both with one tool.
- `@typescript-eslint/parser` : this provides the ESLint parser, which is what ESLint will use to parse and understand when it hits a file with a `.ts` extension.
- `@typescript-eslint/eslint-plugin` : this is the ESLint plugin that provides the rules that we'll enable to lint our code with.

You should install all of these, and remember to use `--save-dev` to have them added to our `package.json` :

```
npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin
```

Once those are installed, create a `.eslintrc.js` configuration file in the root of your project with the following in it:

```
module.exports = {
  parser: '@typescript-eslint/parser',
  plugins: ['@typescript-eslint'],
  extends: ['eslint:recommended', 'plugin:@typescript-eslint/recommended'],
};
```

This configuration tells ESLint to use the `@typescript-eslint/parser` library for parsing code. Without this, ESLint would error when trying to parse a TypeScript file. It then enables the `typescript-eslint` plugin and turns out the recommended set of rules that `eslint` ships with, and the ones that the `typescript-eslint` plugin ships with.

The main catch is that, by default, ESLint will **only look for** `.js` files. So when we run ESLint, we need to tell it to look for `.ts` files:

```
npx eslint src/ --ext .ts
```

Make sure you run this command from the root directory of the project (the directory with the `tsconfig.json` and `package.json` files in it).

Since we'll be using this a lot, I like to add it as a `Lint` script, so we can run `npm run lint` and have it run for us. Let's update our `scripts` entry in `package.json`:

```
"scripts": {
  "dev": "vite",
  "build": "tsc && vite build",
  "serve": "vite preview",
  "lint": "eslint src/ --ext .ts"
}
```

Running `npm run lint` will return nothing, because we're not breaking any rules. But head into `main.ts` and, anywhere outside of a function, and add this:

```
interface Foo {}
```

Now you'll see an error when running `npm run lint`:

```
> 2@0.0.0 lint
> eslint src/ --ext .ts

/home/jack/git/typescript-course-article-2-code/src/main.ts
  35:11 error    An empty interface is equivalent to `{}` @typescript-eslint/no-empty-interface
  35:11 warning  'Foo' is defined but never used         @typescript-eslint/no-unused-vars

✖ 2 problems (1 error, 1 warning)
```

There are two issues that typescript-eslint has found:

- An empty interface isn't allowed; it's telling us that we could just use the TypeScript type `{}` (which denotes an empty object).
- The interface `Foo` isn't used, so it's telling us that we could probably delete it.

There are many more rules that the typescript-eslint plugin provides, so I'd encourage you to give the full list a read and configure any extra rules you might like to enable.

Conclusion

That draws this tutorial to an end. We've done a lot here, and we now have a solid TypeScript foundation on which to build in the next chapter, where we'll add more to our application and explore how to use libraries from npm that don't provide types out of the box, along with some more advanced features that TypeScript provides.

Adding More Functionality

Jack Franklin

Chapter

3

In the previous chapter, we started to put together our application by leveraging TypeScript to write type-safe code to fetch data from our API and render it using lit-html. In this final installment, we'll finish this series of tutorials by adding more functionality to our site. Users will be able to search for photos or videos and star them as a favorite. We'll store these favorites in local storage and see how we can interact with browser APIs whilst benefiting from TypeScript's type safety.

We'll pick up exactly where we left off on tutorial two and start by letting users have the ability to enter a search query. We'll then fire that off to the API and render the resulting photos in the app.

Refactoring the App to Create a render Method

In the previous tutorial, we had hard-coded the app to render after we queried the API for `dogs`, but now that we're going to start building the app properly, we want to let the user determine what to search for.

The way that lit-html works is that once we've rendered our application for the first time into some element, we can then call the Lit `render` method again, passing in different HTML and the same element. It will then efficiently re-render the HTML, doing the minimum amount of work required to update the DOM.

To take advantage of this, we'll create a function called `renderApp` that we can call many times to re-render our app as required. Any data that we need to render will be passed into this function, so to firstly we'll make it take one argument, `results`, which is the result of calling the Pexels API to fetch data. We can reuse the `PhotoSearchAPIResult` type that we defined in the previous tutorial. However, sometimes when we render we won't have any photos available, so we'll declare that `results` can be `PhotoSearchAPIResult | null`:

```
function renderApp(results: PhotoSearchAPIResult | null): void {
  const div = document.getElementById('app');
  if (!div) {
    throw new Error('could not find app div');
  }

  const htmlToRender = html`we will fill this out in a minute...`;
  render(htmlToRender, div);
}
```

If you're working from the code in the second tutorial, you need to replace the block of code that starts with `fetchImagesFromAPI('dogs', 5).then((data) => {` with this code above.

This means if we want to call `renderApp` and we don't have any results, we have to explicitly pass

in `null` :

```
renderApp(null);
```

Often people won't use this and instead will let us call `renderApp()` with no arguments, but I like the explicitness of passing in `null`. It makes it harder to forget to pass in the argument.

Notice that we provide a return type of `void` to signify that our function doesn't return anything at all.

Rendering a Search Form

The HTML that we render will look the same as before when it comes to rendering the results, but we'll also render a small `<form>` that can be submitted to query our API. We'll use Lit's `@submit` syntax to bind a function to the `submit` event. We'll define that function shortly, so don't worry about what that function does right now:

```
const htmlToRender = html`
  <h1>Amazing Photo App</h1>

  <form id="search" @submit=${onFormSubmit}>
    <input type="text" name="search-query" placeholder="dogs" />
    <input type="submit" value="Search" />
  </form>
  <ul>
    ${results
      ? results.photos.map((photo) => {
          return html`<li><img src=${photo.src.small} /></li>`;
        })
      : nothing}
  </ul>
`;
```

Notice that now, within our ``, we dynamically render different content based on whether `results` is `null` or not. If it isn't, we render all our results (this hasn't changed from the previous tutorial), and if it's `null`, we return `nothing`. This is a special value from `lit-html` which needs to be imported:

```
import { render, html, nothing } from 'lit-html';
```

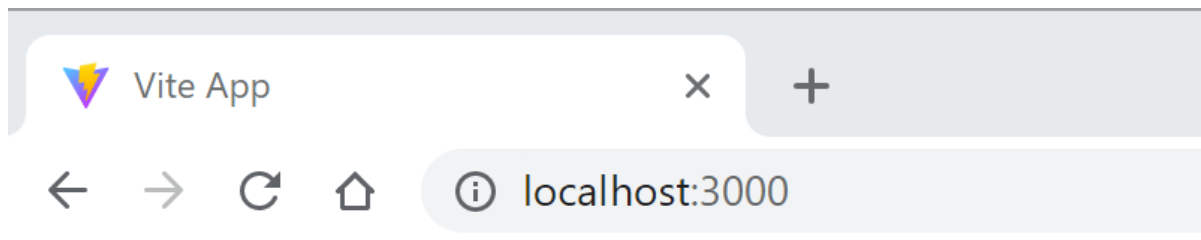
`nothing` signifies to Lit that it shouldn't render anything into the page, and is useful when you only want to render some HTML should a particular value exist, or be not `null`.

We're nearly ready to render the application, but we need to define the `onFormSubmit` function that we referenced in the HTML above. Above the definition of `renderApp`, add this:

```
async function onFormSubmit() {};
```

That's all we need right now to get the app running. We'll fill this function out shortly.

Now we can render the app by calling `renderApp(null)`, and you should see a form appear on the page (we'll add some CSS later on). If you'd like to explore the code a little more, you can [see these changes running on CodeSandbox](#).



Amazing Photo App

3-1. Our application running showing our form input

Searching for Photos

To search for photos, we need to implement the `onFormSubmit` method. The first part of this method will call `event.preventDefault()` to prevent the default form action happening. We'll also check for `event.target` being present. `event.target` represents the element that the event occurred on, and we want to make sure it isn't null. If it is `null`, we can just return from the function early and do nothing.

We'll also define the function as an `async` function, as we'll want to use `await` in here when calling out to the API:

```
async function onFormSubmit(event: SubmitEvent) {
  event.preventDefault();
  if (!event.target) {
    return;
  }
}
```

Notice that I gave the `event` parameter a type of `SubmitEvent`. This is the type of event that gets emitted when an HTML form is submitted (read more on the [MDN docs here](#)). For any global event types like this, TypeScript will have a type that you can use to represent it.

These types don't magically exist, but are defined within TypeScript in a `.d.ts` file. These files are special TypeScript files that define types, but not the actual implementation code. When TypeScript ships, it includes a `Lib.dom.d.ts` file that contains a number of types that represent all the DOM APIs and events that are implemented in the browser.

Once you've typed `SubmitEvent`, you can use your editor's "Go to Definition" functionality (in VS Code it's `F12`) to see the definition of `SubmitEvent`. In the image below, VS Code shows the `SubmitEvent` definition.



```
/home/jack/.vscode-server/bin/ee8c7def80afc00dd6e593ef12f37756d8f504ea/extensions/node_modules/typescript/lib - Definition
7 }
6
5 declare var StyleSheetList: {
4   ...prototype: StyleSheetList;
3   ...new(): StyleSheetList;
2 };
1
14433 interface SubmitEvent extends Event {
1   .../**
2   ... * Returns the element representing
   ... the submit button that triggered
   ... the form submission, or null if the
   ... submission was not triggered by a
   ... button.
3   ... */
```

3-2. VS Code showing the SubmitEvent definition

Using FormData to Read Form Values

Now, in the submit function we can gather the value of our text input. There's loads of ways we could do this, but we're going to make use of the `FormData` API.

A `FormData` object works by taking an HTML `form` element and constructing an object of key/value pairs that represent all the inputs in the form and the values of those inputs. We already have our form - it's `event.target`, because we're working with the `submit` event, so we can construct a `FormData` object by passing it that:

```
async function onFormSubmit(event: SubmitEvent) {
  event.preventDefault();
  if (!event.target) {
    return;
  }

  const formData = new FormData(event.target);
}
```

However, if you do that and now run the TypeScript compiler (`npm run tsc` on the command line, or look for red squiggles in your editor!) you will get an error:

```
src/main.ts:58:33 - error TS2345: Argument of type 'EventTarget' is not assignable to parameter of type 'HTMLFormElement'.
Type 'EventTarget' is missing the following properties from type 'HTMLFormElement': acceptCharset, action, autoComplete

58   const formData = new FormData(event.target);
                                ~~~~~

Found 1 error.
```

Note: you may also get an error that "`formData` is declared, but its value is never used". That's the compiler letting you know that we've created this variable and never used it. But we'll be using it shortly so this one is safe to ignore.

This error happens because TypeScript knows that the element passed to `new FormData()` has to be of type `HTMLFormElement`: a `FormData` object can only be created by passing in a form element. However, TypeScript doesn't know for sure that our `event.target` is an `HTMLFormElement`. It only knows that it's an `EventTarget` — a general type for any HTML element that can listen to and emit events.

This is a good example of a situation that can occur when TypeScript knows less about the world than you. Here, we know reasonably that `event.target` will be a form, because we're in a form's submit event handler, which we've bound with lit-html via the `@submit` syntax. But TypeScript

doesn't know that. It has no special understanding of Lit's syntax. Therefore, this is a case where I feel confident using the `as` syntax to tell TypeScript that, in this instance, we know best:

```
const formData = new FormData(event.target as HTMLFormElement);
```

I don't like using `as`, because it overrides what the compiler thinks, and is an easy way to cause errors that TypeScript would have otherwise spotted, but in this case it's warranted. Sometimes you have to be pragmatic when working with TypeScript and recognize that, due to the nature of the Web, occasionally TypeScript needs a little bit of help.

Using the formData Object

Once we have the `formData` object, we can use `.get()` to query the object. We can pass in the name of the input we want to query, and get back the value of that input:

```
const query = formData.get('search-query');
if (query) {
  ...
}
```

`.get()` returns `FormDataEntryValue | null`, so we check that it has a value before continuing. At this point, you might be wondering what the type of `FormDataEntryValue` actually is. I was too! There are two ways to find this out. The first is to [check the MDN docs for `.get\(\)`](#), which will link you to [the MDN docs for `FormDataEntryValue`](#), where you'll find this:

A string or File that represents a single value from a set of `FormData` key-value pairs.

Another way to find this out would be to use TypeScript. I use “Go to Definition” on the `formData.get()` method in our code, which takes me into the TypeScript definitions. At this point, you can see the definition of the `get` method:

```
get(name: string): FormDataEntryValue | null;
```

And then I can use “Go to Definition” again on the `FormDataEntryValue` type.

```
interface FormData {
  ...append(name: string, value: string | Blob, fileName?: string)
    : void;
  ...delete(name: string): void;
  ...get(name: string): FormDataEntryValue | null;
  ...getAll(name: string): FormDataEntryList;
  ...has(name: string): boolean;
  ...set(name: string, value: string | Blob, fileName?: string)
    : void;
  ...forEach(callbackfn: (value: FormDataEntryValue, key: string,
    parent: FormData) => void, thisArg?: any): void;
}
```

| | |
|--------------------------|---------------|
| Go to Definition | F12 |
| Go to Type Definition | |
| Go to Implementations | Ctrl+F12 |
| Go to References | Shift+F12 |
| Peek | > |
| Find All References | Shift+Alt+F12 |
| Find All Implementations | |

3-3. Using go to definition on the formData.get() method

This reveals the type:

```
type FormDataEntryValue = File | string;
```

So we know that the type of the value returned from the `get()` method, should it not be `null`, is either a `File` or a `string`. This means that, when we deal with this value, we need to deal with TypeScript thinking that it might be a file — even though we know, because of the form we've built, that it's impossible to be a file. We could cast it using `as`, but in this instance we can take advantage of TypeScript's type narrowing.

We could write this code:

```
if (query) {
  const results = await fetchImagesFromAPI(query, 10);
  renderApp(results);
}
```

If we do, we'll get an error, because `fetchImagesFromAPI` expects to be given a `string`, but at this point our `query` is `File | string`. What we can do here is use the `typeof` operator to check that the value is actually a string at runtime:

```
if (query && typeof query === 'string') {
  const results = await fetchImagesFromAPI(query, 10);
  renderApp(results);
}
```

```
}
```

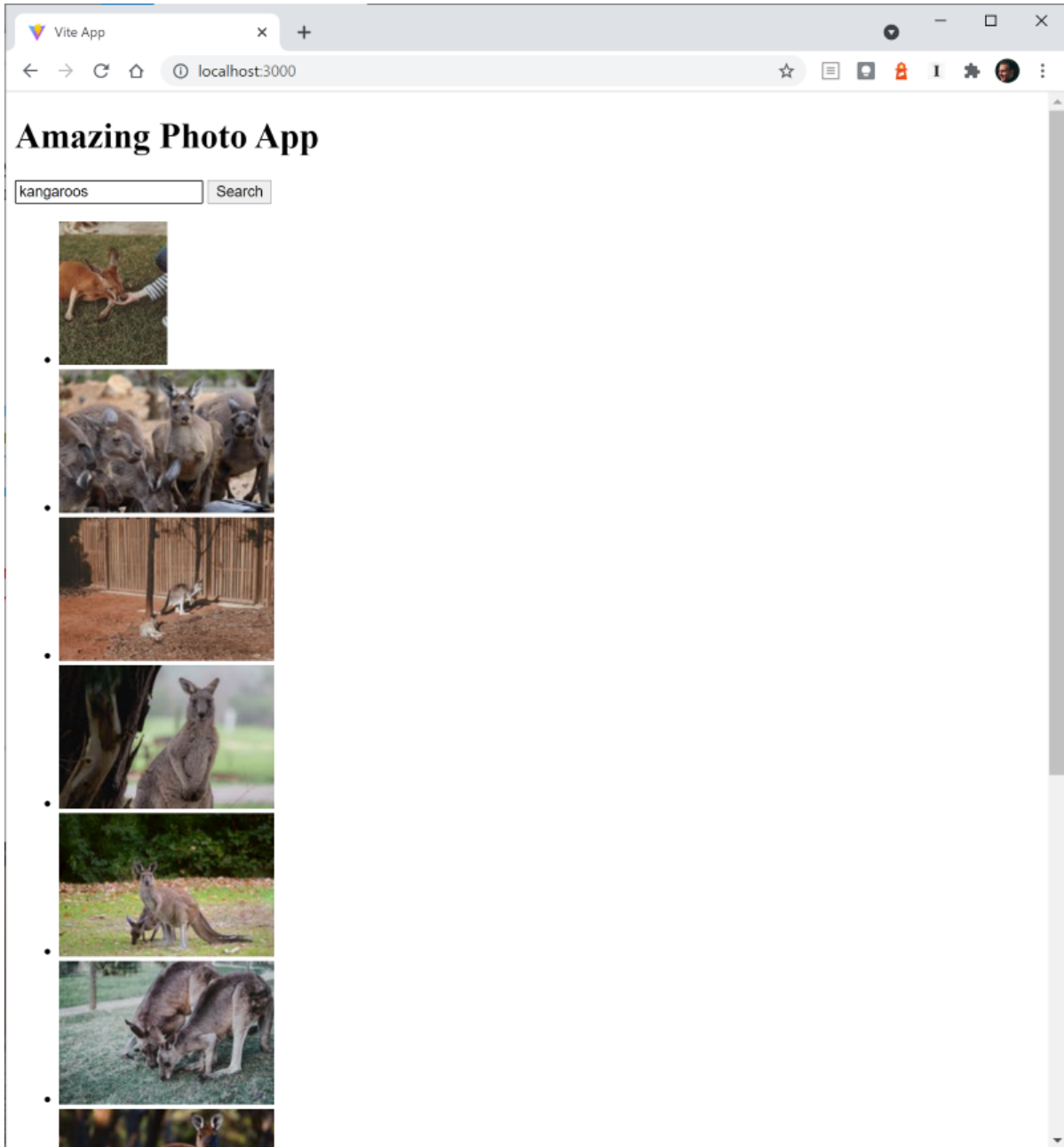
Not only does this ensure — when this code actually runs — that `query` is indeed a string value, but TypeScript can read this code and understand that we've added a conditional check to ensure the value is a string. This is called **type narrowing**. In our case, we started out with a type of `FormDataEntryValue | null` — which actually was `File | string | null` if we expand `FormDataEntryValue` .

Then, in our conditional, we first check for the presence of `query` . Strictly, this check is now not necessary, because the `typeof` check ensures it's not `null` , but I like to leave it because I think it makes it clearer in the code that `query` might not exist. Once we check for the presence of `query` , TypeScript has narrowed the type to `File | string` , because we now know it can't be null. Following the `typeof` check, TypeScript can narrow our type again to just `string` , meaning we can pass it into `fetchImagesFromAPI` safely.

Once we have our query, we call `fetchImagesFromAPI` (this is the same function, as we left it in the previous tutorial, with no changes) and can pass them into `renderApp` to cause Lit to re-render our app with some pictures!

You can [try this out on CodeSandbox](#) — but remember that you'll need to update the code to put your Pexels API key into the app.

Use `npm run dev` to trigger Vite to rebuild and serve the application locally on port `3000` .



3-4. Our app working once I've searched for Kangaroos

Writing an API for Local Storage

Now let's start working towards enabling people to like photos (and, in a bit, videos) by creating a little wrapper around the `LocalStorage` API, which is where we'll save our videos. This is an area where TypeScript shines, because we can take an API built into the platform and wrap it in a layer of type-safety that can also provide information to other developers on what data we're storing.

We'll write this code in a new file, `src/storage.ts`. The first thing we'll do is define the interface for what our stored data will look like:

```
interface StoredUserLikes {
  photos: number[];
  videos: number[];
}
```

When working with TypeScript, it's nearly always a good idea to start with the types — or at least, the main type around which you want to write code. That's not to say you can't change it later on, but I think it really helps get my head clear about what we're working towards.

Knowing that we'll be storing liked videos as well as photos means that I can structure my saved data accordingly, and store two arrays of numeric IDs — one for `photos`, and one for `videos`.

Next let's export the function `saveLikes`:

```
const LOCAL_STORAGE_KEY = '__PEXELS_LIKES__';

export function saveLikes(likes: StoredUserLikes): void {
  window.localStorage.setItem(LOCAL_STORAGE_KEY, JSON.stringify(likes));
}
```

Because the local storage API can only store strings, we first have to take our data and use `JSON.stringify` to convert it into a string.

Note that I define a constant `LOCAL_STORAGE_KEY` for the key that we use when saving the data, and I don't expose it. This is to make sure nothing in our app directly uses the local storage API. I want it to always go through our `src/storage.ts` module.

`LoadLikes` is the opposite of `saveLikes`, but it does potentially return `null` in the case that there is no stored data:

```
export function loadLikes(): StoredUserLikes | null {
  const data = window.localStorage.getItem(LOCAL_STORAGE_KEY);
  if (!data) {
    return null;
  }
  return JSON.parse(data);
}
```

I highly recommend taking advantage of TypeScript and wrapping these built-in APIs in a small

layer that can provide you with some type-safety and an API catered to your particular use case.

Now that we've got these functions defined, let's start to let people like photos.

Favoriting Photos

Before we add functionality to like photos, we should first think about breaking our application down a bit. The current render method renders the entire app, and as we add functionality it's going to become quite unwieldy to work with if we have one huge function.

Let's create `src/photo-renderer.ts`, which will be where we move the logic for rendering an individual photo.

Note: if you were using React/Angular/Vue/Svelte and so on here, you'd probably create a new component. For the sake of focusing on TypeScript, we'll just keep defining functions that return HTML via lit-html.

Let's define the shell of `src/photo-renderer.ts`:

```
import { Photo } from './main';
import { html } from 'lit-html';

export function renderPhoto(photo: Photo) {
  return html`<li><img src=${photo.src.small} /></li>`;
}
```

Immediately, though, we hit a problem. `main.ts` doesn't expose the `Photo` interface. We could fix this, by making it `export interface Photo {...}`, but then we'd hit on a different problem: `main.ts` would need to import `renderPhoto` from `photo-renderer.ts`, but at the same time `photo-renderer.ts` would need to import `Photo` from `main.ts`. This means we've introduced a *circular dependency* between `main.ts` and `photo-renderer.ts`, because both files depend on the other.

We could actually continue down that path; the circular dependency can be handled and the code will work, but it's highlighting a general issue with our application's structure that I think is worth resolving. In any TypeScript application, you're going to have some types that are core to your entire system, and those types are going to need to be imported by lots of code in lots of files across your codebase. Having ours in `main.ts` was fine when we were getting started, but now we need something a little more structured.

Creating `pexels.ts` to Contain Our API Code

To fix this, let's create `src/pexels.ts`, which can be the module that contains all the code for making requests to the API, and defining the types that can be returned from that API:

```
const PEXELS_API_KEY = 'your-pexels-api-key-goes-here';

export interface Photo {
  id: number;
  width: number;
  height: number;
  url: string;
  photographer: string;
  photographer_url: string;
  photographer_id: string;
  avg_color: string;
  src: {
    original: string;
    large2x: string;
    large: string;
    medium: string;
    small: string;
    portrait: string;
    landscape: string;
    tiny: string;
  };
}

export interface PhotoSearchAPIResult {
  total_results: number;
  page: number;
  per_page: number;
  photos: Photo[];
  next_page: string;
}

export async function fetchImagesFromAPI(
  searchTerm: string,
  perPage: number
): Promise<PhotoSearchAPIResult> {
  const result = await fetch(
    `https://api.pexels.com/v1/search?query=${searchTerm}&per_page=${perPage}`,
    {
      headers: {
        Authorization: PEXELS_API_KEY,
      },
    }
  );
}
```



```
const json = (await result.json()) as PhotoSearchAPIResult;
return json;
}
```

Don't forget to ensure that all the types and functions have an `export` at the start, so they can be used elsewhere. Then in `src/main.ts`, we can delete all the code that's been moved into `src/pexels.ts`, and update our imports:

```
import { render, html, nothing } from 'lit-html';
import { renderPhoto } from './photo-renderer';
import { fetchImagesFromAPI, PhotoSearchAPIResult } from './pexels';
```

And now in `src/photo-renderer.ts` we can update our imports too:

```
import { Photo } from './pexels';
import { html } from 'lit-html';

export function renderPhoto(photo: Photo) {
  return html`<li><img src=${photo.src.small} /></li>`;
}
```

Finally, in `src/main.ts`, we can update our main render code to use the `renderPhoto` function:

```
<ul>
  ${results
    ? results.photos.map((photo) => {
      return renderPhoto(photo);
    })
    : nothing}
</ul>
```

After that little tidy up, we're now ready to crack on with allowing the user to favorite photos. I've found during my time with TypeScript that I'm much more willing to make these small refactors as I work, because the compiler will error loudly if something goes wrong. If at any point during the refactoring I've made a mistake — such as forgetting to import a function, or forgetting to add an `export` keyword before a type that I wanted to expose — the compiler will error and I'll be made aware of the problem early on. That's one of the best things about TypeScript — the shorter feedback loop when you've made a mistake. [You can explore the full changes on CodeSandbox](#) if you'd like to compare your app's code to ensure you've not missed anything.

Make sure at this point you check the app is working: `npm run dev` will get it running again if you don't have it. You can also run `npm run tsc` to ensure that TypeScript is giving you no errors.

Liking a Photo

Let's now update `photo-renderer.ts` with a button that the user can click to like the photo:

```
export function renderPhoto(photo: Photo) {
  return html`<li class="photo">
    <img src=${photo.src.small} />
    <button class="like">Like</button>
  </li>`;
}
```

At this point, we'll also throw some CSS in to make the app look a tiny bit less ugly. So, in `src/main.ts`, add back in the import for `style.css`:

```
import './style.css';
```

And then update `style.css` like so:

```
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}

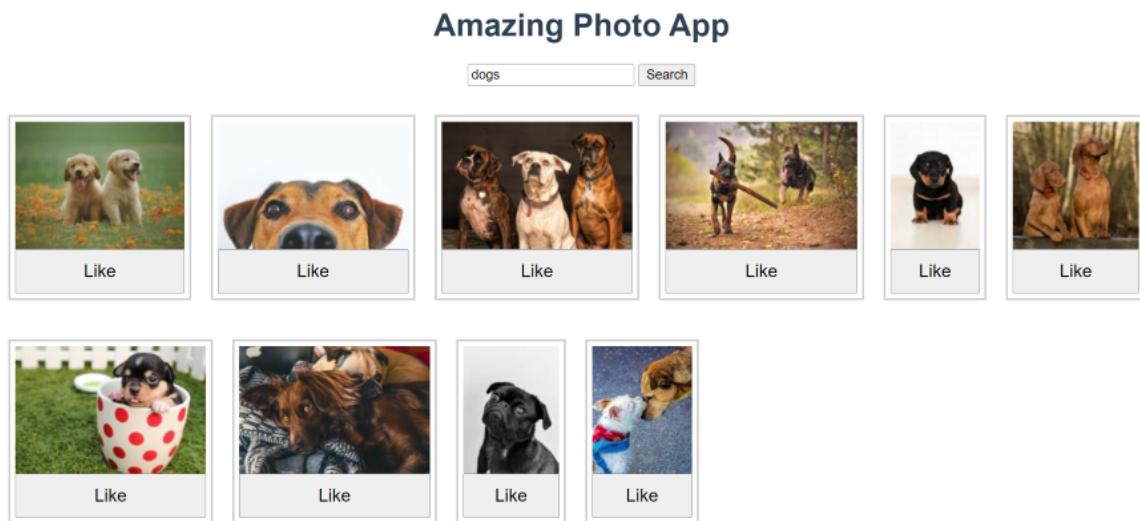
ul {
  margin: 10px;
  padding: 0;
  display: flex;
  flex-wrap: wrap;
}

.photo {
  display: flex;
  margin: 20px 10px;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  border: 2px solid #ccc;
  padding: 5px;
}

.photo img {
  display: block;
```

```
}  
  
.photo button {  
  width: 100%;  
  font-size: 18px;  
  padding: 10px;  
  cursor: pointer;  
}
```

Feel free to change this as much as you like if you prefer a different look, but after rebuilding the app with those styles applied, it should appear like the picture below.



3-5. Our photo application with styles applied

Defining the Types of Callback Functions

To enable the photos to be liked (or then disliked), we're going to have to do some work to organize our application and how the `renderPhoto` function will let the application know that the user has clicked the button. This will mean that we'll pass `renderPhoto` a callback function that it can call when its like button is pressed. Then our main application can listen for that callback and deal with liking the photo and updating the data.

First, update `photo-renderer.ts` like so:

```
import { Photo } from './pexels';  
import { html } from 'lit-html';
```

```

export function renderPhoto(
  photo: Photo,
  onLikeClick: (photoId: number) => void
) {
  return html`<li class="photo">
    <img src=${photo.src.small} />
    <button class="like" @click=${() => onLikeClick(photo.id)}>Like</button>
  </li>`;
}

```

Notice how we pass in the type for the `onLikeClick` function:

```

export function renderPhoto(
  photo: Photo,
  onLikeClick: (photoId: number) => void
) {...}

```

Here we see how to define callback functions in TypeScript. You can think of it like defining an arrow function, except you don't fill out the body. All you put after the `=>` is the return type — in our case, `void`, as we don't expect anything to be returned.

We can then use the lit-html syntax of `@click` to bind to the button's click event and call the callback with the ID of our photo. This is so our main application knows whether to update the photo to be liked, or disliked if the user had previously liked it:

```

<button class="like" @click=${() => onLikeClick(photo.id)}>Like</button>

```

Now we can return to `src/main.ts`, where, if you have it set up, your editor should be highlighting the call to `renderPhoto` as an error, because you're not passing in the second argument. This is without a doubt the best benefit of TypeScript: when you're in the middle of some work, it'll often remind you exactly what the next steps are.

Let's fix the error by defining a placeholder method:

```

function renderApp(results: PhotoSearchAPIResult | null): void {
  const div = document.getElementById('app');
  if (!div) {
    throw new Error('could not find app div');
  }

  function onUserLikeClick(photoId: number): void {}

  const htmlToRender = html`

```

```

<h1>Amazing Photo App</h1>

<form id="search" @submit=${onFormSubmit}>
  <input type="text" name="search-query" placeholder="dogs" />
  <input type="submit" value="Search" />
</form>
<ul>
  ${results
    ? results.photos.map((photo) => {
      return renderPhoto(photo, onUserLikeClick);
    })
    : nothing}
</ul>
`;
render(htmlToRender, div);
}

```

And now we can fill in the function:

```

function onUserLikeClick(photoId: number): void {
  const likedData = loadLikes() || {
    photos: [],
    videos: [],
  };
  const photoIsLiked = likedData.photos.includes(photoId);
  if (photoIsLiked) {
    likedData.photos = likedData.photos.filter((id) => id !== photoId);
  } else {
    likedData.photos.push(photoId);
  }
  saveLikes(likedData);
}

```

Don't forget to also import the functions we need:

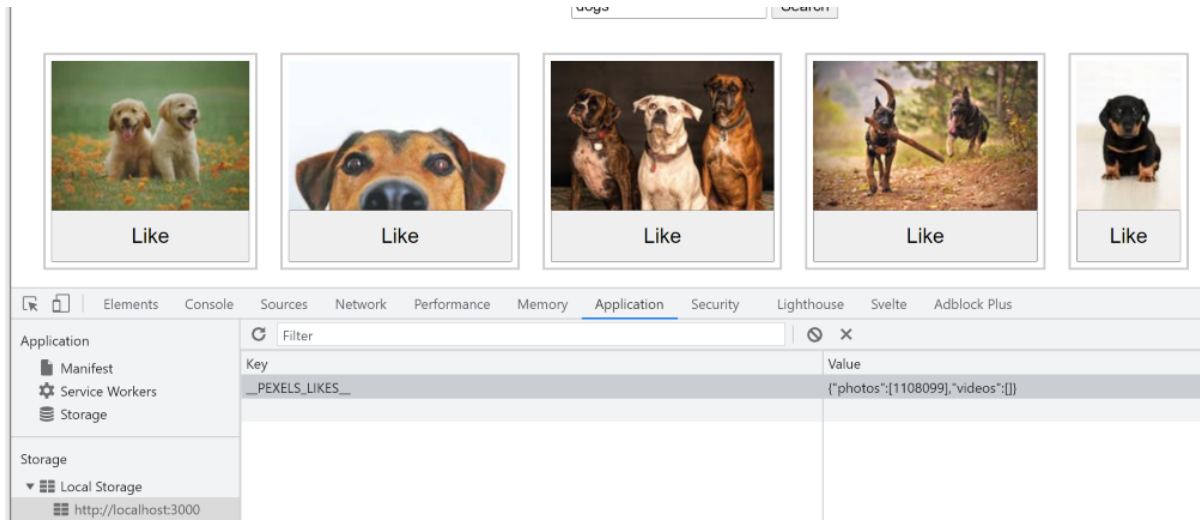
```

import { loadLikes, saveLikes } from './storage';

```

We first use `LoadLikes()` to fetch the data from local storage. We know that this can return `null`, so if it does, we fall back to a new blank object that we can use as our stored data. We then check if the photo is liked or not. If it is, we filter the list to remove it, and if it isn't, we push the new ID onto the list to make it liked. Finally, we write that data to local storage.

We can use the developer tools to confirm the data has been written: under the **Application** tab (this is the Chrome Developer Tools, so it may be located differently in your browser), you can see local storage and confirm that our data has been stored.



3-6. Local storage updated with our liked photo

Once we have our photo liked (or disliked), we need to re-render the application so we can update the UI. We'll also pass in a `true` or `false` value to `renderPhoto`, telling it if the photo is liked or not.

First, let's move the `const LikedData = ...` line out of our `onUserLikeClick` callback and up to the top level of the `renderApp()` function. This means that, every time we render, we'll get the updated set of liked photos and videos:

```
function renderApp(results: PhotoSearchAPIResult | null): void {
  const div = document.getElementById('app');
  const likedData = loadLikes() || {
    photos: [],
    videos: [],
  };
  ...
}
```

Then in `onUserLikeClick` we need to call our `renderApp` method again:

```
function onUserLikeClick(photoId: number): void {
  const photoIsLiked = likedData.photos.includes(photoId);
  if (photoIsLiked) {
    likedData.photos = likedData.photos.filter((id) => id !== photoId);
  } else {
    likedData.photos.push(photoId);
  }
  saveLikes(likedData);
}
```

```
renderApp(results);
}
```

And then, when we call `renderPhoto`, let's pass in a Boolean to determine if the photo is liked or not:

```
results.photos.map((photo) => {
  const photoIsLiked = likedData.photos.includes(photo.id);
  return renderPhoto(photo, onUserLikeClick, photoIsLiked);
})
```

And finally, update `photo-renderer.ts` to show **Dislike** or **Like** depending on whether the photo is liked or not:

```
export function renderPhoto(
  photo: Photo,
  onLikeClick: (photoId: number) => void,
  photoIsLiked: boolean
) {
  return html`<li class="photo">
    <img src=${photo.src.small} />
    <button class="like" @click=${() => onLikeClick(photo.id)}>
      ${photoIsLiked ? 'Dislike' : 'Like'}
    </button>
  </li>`;
}
```

And with that, we have our basic like functionality working! Make sure yours is working by running the application. You should see a button with the text “Like” appear below each photo. You can click it to change the text to “Dislike”, which should also update local storage. You can test this by liking a photo, refreshing the page, and searching for the same query again. The photo you liked previously should appear, and the text underneath will say “Dislike”.

[This CodeSandbox](#) has all the changes we've just made — but don't forget to update the API key in `pexels.ts` before it will start working.

Let's now move on to some small TypeScript improvements we can make, before diving into adding support for showing videos from search results too.

Improving Our Code with `readonly`

Often, when working in client-side applications, you'll want to create objects or arrays that represent data, and ideally you don't want your code to mutate them. For example, our results

from the API include an array of photos. At no point in the codebase does it make sense to update this array to push data onto it. Because that data represents data from the API, we don't want our code to modify it. It effectively is a readonly object.

TypeScript lets us do this for both objects and arrays. `Readonly` lets us make objects whose properties are readonly, and `ReadonlyArray` lets us define arrays that can't be mutated.

Let's update some of our API results code to make use of this. In `pexels.ts`, let's update the `src` object to be readonly:

```
export interface Photo {
  id: number;
  width: number;
  height: number;
  url: string;
  photographer: string;
  photographer_url: string;
  photographer_id: string;
  avg_color: string;
  src: Readonly<{
    original: string;
    large2x: string;
    large: string;
    medium: string;
    small: string;
    portrait: string;
    landscape: string;
    tiny: string;
  }>;
}
```

We can also update `PhotoSearchAPIResult` :

```
export interface PhotoSearchAPIResult {
  total_results: number;
  page: number;
  per_page: number;
  // Note: these are both equivalent; use whichever you prefer
  photos: readonly Photo[];
  // photos: ReadonlyArray<Photo>;
  next_page: string;
}
```

When defining arrays, you can use `readonly Photo[]` or `ReadonlyArray<Photo>` . Both mean the same thing. I tend to prefer the `readonly` keyword as I find it slightly easier to read, but it's

entirely up to you which one you prefer.

We can go one step further by making each property on the interfaces we have `readonly` too. This means that TypeScript will not allow any code to modify them. To do this, you can preface each key in the object with the `readonly` keyword:

```
export interface Photo {
  readonly id: number;
  readonly width: number;
  readonly height: number;
  readonly url: string;
  readonly photographer: string;
  readonly photographer_url: string;
  readonly photographer_id: string;
  readonly avg_color: string;
  readonly src: Readonly<{
    original: string;
    large2x: string;
    large: string;
    medium: string;
    small: string;
    portrait: string;
    landscape: string;
    tiny: string;
  }>;
}

export interface PhotoSearchAPIResult {
  readonly total_results: number;
  readonly page: number;
  readonly per_page: number;
  readonly photos: readonly Photo[];
  readonly next_page: string;
}
```

Whilst this is undoubtedly quite verbose, it does mean that it's now practically impossible to accidentally mutate data because the compiler will error.

You might notice in the code above that for the `src` key on `Photo` there's a bit of inconsistency in where we have the `readonly` keyword:

```
readonly src: Readonly<{
  original: string;
  large2x: string;
  large: string;
  medium: string;
```

```
    small: string;
    portrait: string;
    landscape: string;
    tiny: string;
  }>;
```

You might be wondering why we have it on the `src` property, but then not on any of the properties within there. That's because, when you take an object and pass it as the generic argument to `ReadOnly<T>` (in the above code, the generic `T` is our object with `original`, `large2x`, and so on), each property within the object that you pass in will automatically be made `readonly`. The utility `ReadOnly<T>` type does exactly that: it creates a new version of the type `T` where each item has been made `readonly`. So another way to write that code would be:

```
readonly src: {
  readonly original: string;
  readonly large2x: string;
  readonly large: string;
  readonly medium: string;
  readonly small: string;
  readonly portrait: string;
  readonly landscape: string;
  readonly tiny: string;
};
```

You can feel free to do this, if you'd like to be more explicit, but I like saving the typing of `readonly` by using the utility type.

The reason we need the `readonly` before `src` is to ensure that the `src` property itself is `readonly`, rather than just the object it contains. Imagine if we only had this:

```
src: ReadOnly<{...}>;
```

Then I could still write code that looked like this:

```
photo.src = { ... };
```

This is because we didn't set `src` to be `readonly`, but only the object that it contained. By prepending `src` with the `readonly` keyword, we've made it so the above code would error, and ensured that our API results can't be messed with.

Adding Video Results

So that we can explore some more of what TypeScript has to offer, we're going to mix up our application a little and when we search, also search for videos, which [the Pexels API also provides](#). When a user enters a query, we'll search for both photos and videos and combine them into one list that the user can then scroll through and like/dislike.

The first thing to do in `pexels.ts` is define the interfaces that we'll use. I'll spare you all the details, but the process here is to look at the API documentation and translate that into a set of interfaces that, unsurprisingly, look a lot like the `Photo` interfaces we've previously defined:

```
export interface VideoFile {
  id: number;
  quality: 'hd' | 'sd';
  file_type: string;
  width: number;
  height: number;
  link: string;
}

export interface Video {
  readonly id: number;
  readonly url: string;
  readonly image: string;
  readonly duration: number;
  readonly video_files: readonly VideoFile[];
}

export interface VideoSearchAPIResult {
  readonly page: number;
  readonly per_page: number;
  readonly next_page: number;
  readonly total_results: number;
  readonly videos: readonly Video[];
}
```

Note that I haven't included all the fields Pexels provides here, but just a subset that we care about.

We can then define `fetchVideosFromAPI`, which is nearly identical to `fetchImagesFromAPI`, but uses a different URL:

```
export async function fetchVideosFromAPI(
  searchTerm: string,
```

```

    perPage: number
  ): Promise<VideoSearchAPIResult> {
    const result = await fetch(
      `https://api.pexels.com/v1/videos/search?query=${searchTerm}&per_page=${perPage}`,
      {
        headers: {
          Authorization: PEXELS_API_KEY,
        },
      }
    );

    const json = (await result.json()) as VideoSearchAPIResult;
    return json;
  }

```

Now that we can fetch videos from the API too, let's update our `renderApp` method so it takes in an array of photos and videos and renders them in a random order — so that when you search, you get a mixture of photos and videos for your search topic. We'll take the `PhotoSearchAPIResult` and the `VideoSearchAPIResult` and combine the arrays of photos and videos into one array:

```

async function onFormSubmit(event: SubmitEvent) {
  // code omitted to save space

  if (query && typeof query === 'string') {
    const results = await fetchImagesFromAPI(query, 10);
    const videos = await fetchVideosFromAPI(query, 10);

    const photosAndVideos = [];
    for (let i = 0; i < results.photos.length; i++) {
      photosAndVideos.push(results.photos[i]);
      photosAndVideos.push(videos.videos[i]);
    }
  }

  renderApp(photosAndVideos);
}

```

It's a rudimentary approach, but we can loop over the resulting data and push a photo, then a video, into our array, to ensure we end up with an array of mixed photos and videos.

We have a problem, though. If you hover over `photosAndVideos` in your editor (or do the appropriate action to have your editor show you the type of this array), you'll see `any[]`.

```

const photosAndVideos: any[]
const photosAndVideos = [];
for (let i = 0; i < results.photos.length; i++) {
  ··photosAndVideos.push(results.photos[i]);
  ··photosAndVideos.push(videos.videos[i]);
}

```

3-7. TypeScript showing that the type of photos and videos is any

We can fix this with an explicit type declaration:

```
const photosAndVideos: Array<Photo | Video> = [];
```

We also need to make sure we update the import from `./pexels` to include `Photo` and `Video`:

```
import {
  fetchImagesFromAPI,
  fetchVideosFromAPI,
  Photo,
  PhotoSearchAPIResult,
  Video,
} from './pexels';
```

The line that calls `renderApp` and passes in `photosAndVideos` will cause an error, because `photosAndVideos` is not the expected type to pass into the `renderApp` function. Let's resolve that now.

Rendering Videos

Now that we have our list of resources to render, let's update our `renderApp` method to take this list, rather than the `PhotoSearchAPIResult` it was taking:

```
function renderApp(results: Array<Photo | Video> | null): void {}
```

We'll still allow the `results` to be `null`, but now alternatively they can be our mixed array of photos and videos.

After changing that type, you'll get an error when it comes to rendering the data:

```
results.photos.map((photo) => {
  const photoIsLiked = likedData.photos.includes(photo.id);
  return renderPhoto(photo, onUserLikeClick, photoIsLiked);
})
```

The error will appear on `results.photos.map`. Whereas previously `results` was an object that had an array of `photos` inside, it's now an array itself, so TypeScript is giving us an error that the key `photos` doesn't exist on an array.

Firstly, we can update the code to map directly over `results`, which is now an array, but rather than the object passed to the callback being a `Photo`, it can now also be a `Video`. And how do we determine what object we're working with? That leads us nicely to type predicates.

Type Predicates

A **type predicate** enables us to write a function that determines if a particular object is of a given type. We can then communicate that to the compiler, which can then understand our code and narrow the types accordingly.

In our case, we can write an `isPhoto(object)` function, which will take in an object that's either a `Photo` or a `Video`, and return `true` or `false` depending on whether or not it's a `Photo`. Using a special syntax, we can then communicate that to TypeScript, so that when we call `if(isPhoto(x))` the compiler will know that `x` is a `Photo` inside that conditional.

To determine if something is a `Photo`, we need to find something unique to a `Photo` object that a `Video` object doesn't have. Looking at the interfaces we've declared, we can notice that a `Video` has a `duration` field, which a `Photo` doesn't.

Note: I could just as easily define `isVideo`, not `isPhoto` here — so feel free to do that if you prefer. There's no real reason why I've gone for `isPhoto`.

We'll define this function in `pexels.ts`. Here's how we could define the `isPhoto` function normally:

```
export function isPhoto(object: Photo | Video): boolean {
  const hasDuration = 'duration' in object;
  return !hasDuration;
}
```

This would work, but for this function to become a type predicate, you tell the compiler what this function asserts about the type, rather than the return type of `boolean`. All type predicates have

to return `boolean`, so for the return type, we use the `x is Y` syntax to state what this predicate tells us:

```
export function isPhoto(object: Photo | Video): object is Photo {
  const hasDuration = 'duration' in object;
  return !hasDuration;
}
```

The TypeScript compiler can now understand that if we call this function and it returns `true`, that `object` is indeed a `Photo`. But, because we told it the input is `Photo | Video`, it can also understand that if the function returns `false`, that `object` must be a `Video`.

Type predicates are an extremely useful tool to have up your sleeve. Whenever you're working with mixed data and need to be able to assert that a particular object is a given type, these should be your go-to option. One word of warning: be very careful that your assertion code is accurate, and that at runtime, when you call `isPhoto`, it does correctly return `true / false` for an object. TypeScript completely trusts that your implementation of the predicate function is correct.

Rendering Videos

Now that we can distinguish between a photo and a video, we can get back to rendering videos on the page. In `renderApp` we can now make use of our type predicate:

```
results.map((resource) => {
  if (isPhoto(resource)) {
    const photoIsLiked = likedData.photos.includes(resource.id);
    return renderPhoto(resource, onUserLikeClick, photoIsLiked);
  } else {
    // TODO: render video
    return nothing;
  }
})
```

Don't forget to also import `isPhoto` from `pexels.ts`:

```
import {
  fetchImagesFromAPI,
  fetchVideosFromAPI,
  isPhoto,
  Photo,
  Video,
} from './pexels';
```

Now in `photo-renderer.ts` (which we probably want to consider renaming!), we can define `renderVideo`, which will look suspiciously similar to `renderPhoto`. The only difference is the URL we use for the image, which for the video is simply `video.image`:

```
export function renderVideo(
  video: Video,
  onLikeClick: (videoId: number) => void,
  videoIsLiked: boolean
) {
  return html`<li class="photo">
    <img src=${video.image} />
    <button class="like" @click=${() => onLikeClick(video.id)}>
      ${videoIsLiked ? 'Dislike' : 'Like'}
    </button>
  </li>`;
}
```

Make sure you update the imports to include the `Video` type:

```
import { Photo, Video } from './pexels';
```

While we're talking rendering, we'll also update the CSS. The images that represent videos are much larger, so let's ensure we limit the width of each `` that we render. Add `max-width: 50%` to the `.photo` class, and `max-width: 100%` to the `.photo img` styles:

```
.photo {
  display: flex;
  max-width: 50%; /* ADD THIS */
  margin: 20px 10px;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  border: 2px solid #ccc;
  padding: 5px;
}

.photo img {
  display: block;
  max-width: 100%; /* ADD THIS */
}
```

Now in `renderApp` we can render a video if `isPhoto(resource)` returns `false`:

```
if (isPhoto(resource)) {
```

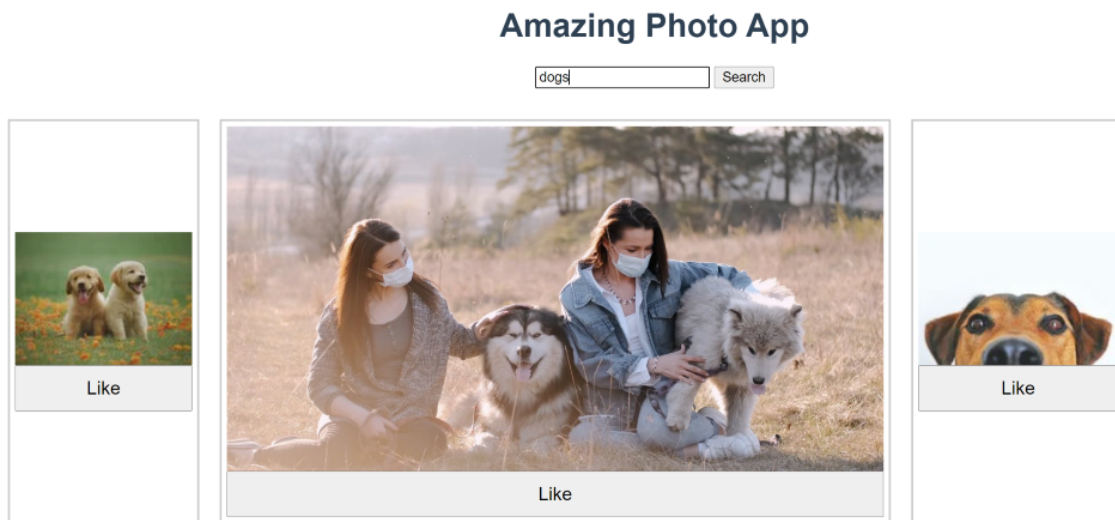


```
const photoIsLiked = likedData.photos.includes(resource.id);
return renderPhoto(resource, onUserLikeClick, photoIsLiked);
} else {
const videoIsLiked = likedData.videos.includes(resource.id);
return renderVideo(resource, onUserLikeClick, videoIsLiked);
}
```

You'll also need to make sure `renderVideo` is imported:

```
import { renderPhoto, renderVideo } from './photo-renderer';
```

And with that change, we now have both photos and videos being rendered.



3-8. Our application rendering both photos and videos

You can also [try it out on CodeSandbox](#)(with the usual caveat of adding your Pexels API key!).

For the sake of this tutorial, we're only going to show an image that represents each video, and not integrate a video player into our application. If we were continuing to build this app, we'd probably want to allow the user to click on the image and get a video player to appear.

Liking Videos

While we can render videos, liking them still assumes that the given ID represents a photo:

```
function onUserLikeClick(photoId: number): void {
```

```

const photoIsLiked = likedData.photos.includes(photoId);
if (photoIsLiked) {
  likedData.photos = likedData.photos.filter((id) => id !== photoId);
} else {
  likedData.photos.push(photoId);
}
saveLikes(likedData);
renderApp(results);
}

```

We could define a separate callback function for when a user clicks to like a video, but instead we can update this function to deal with either a photo or a video. Rather than take an ID, let's take the entire resource as the argument, because then we can use our `isPhoto` type predicate again.

This is a good tip to note generally: when you end up passing small parts of your objects around — in this case, an ID for a photo/video — you lose the ability to then check the exact type the ID takes. Sometimes this is not a problem: if you have a function that only deals with photos, and only needs the ID, then just pass it the ID.

Now that our function gets the entire resource, we need to check what type it is and look at either `LikedData.photos` or `LikedData.videos`. We can then update the relevant array, before updating our saved data again at the end of the function:

```

function onUserLikeClick(resource: Photo | Video): void {
  let arrayOfLikes: number[] = [];
  if (isPhoto(resource)) {
    arrayOfLikes = likedData.photos;
  } else {
    arrayOfLikes = likedData.videos;
  }
  const resourceIsLiked = arrayOfLikes.includes(resource.id);
  if (resourceIsLiked) {
    arrayOfLikes = arrayOfLikes.filter((id) => id !== resource.id);
  } else {
    arrayOfLikes.push(resource.id);
  }
  if (isPhoto(resource)) {
    likedData.photos = arrayOfLikes;
  } else {
    likedData.videos = arrayOfLikes;
  }
  saveLikes(likedData);
  renderApp(results);
}

```

Now we can update `renderPhoto` to make the type definition for the callback correct, and also pass the entire photo into the callback:

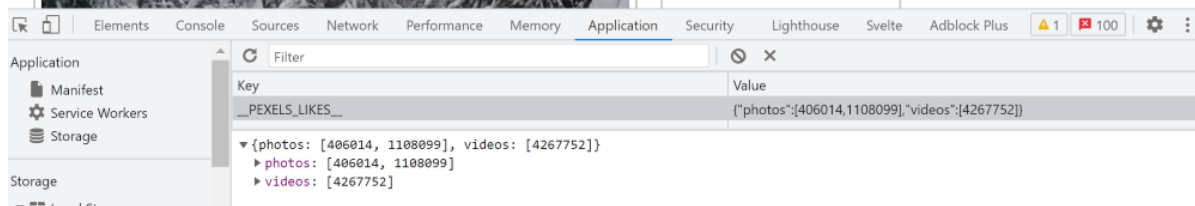
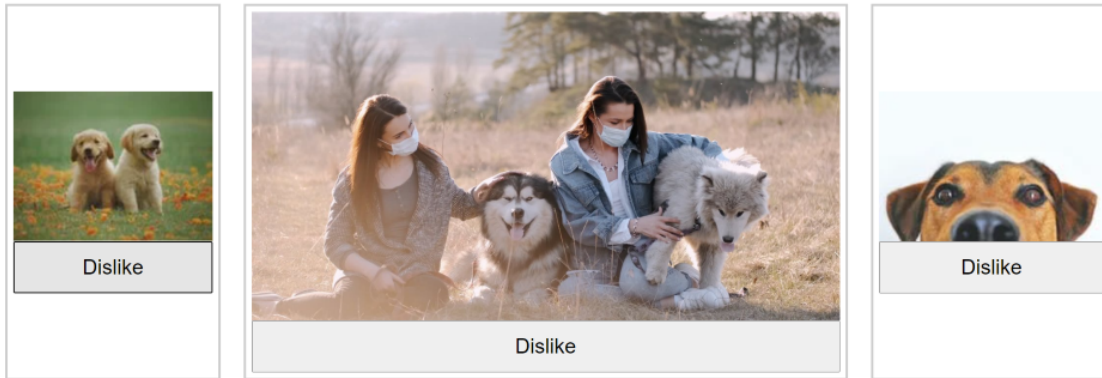
```
export function renderPhoto(
  photo: Photo,
  onLikeClick: (resource: Photo) => void,
  photoIsLiked: boolean
) {
  return html`<li class="photo">
    <img src=${photo.src.small} />
    <button class="like" @click=${() => onLikeClick(photo)}>
      ${photoIsLiked ? 'Dislike' : 'Like'}
    </button>
  </li>`;
}
```

Notice how despite the fact that the callback takes a `Photo | Video`, in the type definition for `renderPhoto` we only say that it takes a `Photo`. This is useful for anyone in the future reading this code to help clarify that this function will only ever pass a `Photo` through to the callback. TypeScript is happy with this, because anything that matches the type of `Photo` will also match the type of `Photo | Video`.

We then make a similar change to `renderVideo`:

```
export function renderVideo(
  video: Video,
  onLikeClick: (resource: Video) => void,
  videoIsLiked: boolean
) {
  return html`<li class="photo">
    <img src=${video.image} />
    <button class="like" @click=${() => onLikeClick(video)}>
      ${videoIsLiked ? 'Dislike' : 'Like'}
    </button>
  </li>`;
}
```

And now we can render our application, search for something, and like either photos or videos!



3-9. Chrome Developer Tools showing localStorage updated with liked photos and videos

Defining a Resource Type

Our application is now working and done, but there's some nice tidying up we can do. You might have noticed in the last few sections that the type `Photo | Video` has appeared a lot. When you get the same union type appearing multiple times, that can be a sign that you should consider defining that as its own type. In this case, we can define a union type that is defined as `Photo | Video`.

Let's define this in `pexels.ts`:

```
export type Resource = Photo | Video;
```

And now anywhere in our application where we've typed `Photo | Video` can be updated to use this `Resource` type, such as `isPhoto`:

```
export function isPhoto(object: Resource): object is Photo {
  const hasDuration = 'duration' in object;
```

```
    return !hasDuration;
  }
```

And `onUserLikeClick` :

```
function onUserLikeClick(resource: Resource): void {...}
```

And `renderApp` :

```
function renderApp(results: Array<Resource> | null): void {
```

To use the `Resource` type in `main.ts` , you'll need to import it:

```
import {
  fetchImagesFromAPI,
  fetchVideosFromAPI,
  isPhoto,
  Resource,
} from './pexels';
```

Remember, you could also write that type as `results: Resource[] | null` if you wanted. One further improvement we can make while we're here is to also make it `readonly` , as discussed earlier in this tutorial:

```
function renderApp(results: readonly Resource[] | null): void {}
```

We can also update the `photosAndVideos` array that's defined in `onSubmit` :

```
const photosAndVideos: Resource[] = [];
```

And with that, we've now got our `Resource` type being used everywhere. One of the main reasons to do this becomes clear if, in the future, we have to consider another type of resource. Let's imagine that our app expands into animated GIFs. We can now update our `Resource` type to:

```
type Resource = Photo | Video | Gif;
```

And now, everywhere in our application where we don't deal with that third type will cause compiler errors. It's a nice way to have the compiler almost generate a to-do list for you!

Defining `renderResource`

It won't have escaped your attention that `renderPhoto` and `renderVideo` are almost identical, so as one small refactoring step, let's define `renderResource`, which can deal with rendering either. The only difference is the URL to use for the image. All the other code is the same, regardless of which resource we're dealing with. We'll define this in `photo-renderer.ts`:

```
export function renderResource(  
  resource: Resource,  
  onLikeClick: (resource: Resource) => void,  
  resourceIsLiked: boolean  
) {  
  const imageURL = isPhoto(resource) ? resource.src.small : resource.image;  
  
  return html`<li class="photo">  
    <img src=${imageURL} />  
    <button class="like" @click=${() => onLikeClick(resource)}>  
      ${resourceIsLiked ? 'Dislike' : 'Like'}  
    </button>  
  </li>`;  
}
```

Yet again we make use of `isPhoto` to dynamically decide how to get the URL for the image, and other than that, this function is identical to `renderPhoto`, except that I've updated any variables that started with `photo` to now start with `resource`.

We can now delete `renderPhoto` and `renderVideo` and update the entire of `photo-renderer.ts` to be as follows:

```
import { isPhoto, Resource } from './pexels';  
import { html } from 'lit-html';  
  
export function renderResource(  
  resource: Resource,  
  onLikeClick: (resource: Resource) => void,  
  resourceIsLiked: boolean  
) {  
  const imageURL = isPhoto(resource) ? resource.src.small : resource.image;  
  
  return html`<li class="photo">  
    <img src=${imageURL} />  
    <button class="like" @click=${() => onLikeClick(resource)}>  
      ${resourceIsLiked ? 'Dislike' : 'Like'}  
    </button>  
  </li>`;  
}
```

```
}
```

In `main.ts`, we can update our rendering code:

```
const htmlToRender = html`
  <h1>Amazing Photo App</h1>

  <form id="search" @submit=${onFormSubmit}>
    <input type="text" name="search-query" placeholder="dogs" />
    <input type="submit" value="Search" />
  </form>
  <ul>
    ${results
      ? results.map((resource) => {
          const resourceIsLiked = isPhoto(resource)
            ? likedData.photos.includes(resource.id)
            : likedData.videos.includes(resource.id);
          return renderResource(resource, onUserLikeClick, resourceIsLiked);
        })
      : nothing}
  </ul>
`;
```

And update our imports:

```
import { renderResource } from './photo-renderer';
```

This code is now much cleaner and less duplicated — a very solid refactoring! [Feel free to explore these changes on CodeSandbox](#). There's one more part I'd like to tidy up in how we store liked data. Having two separate arrays for photos and videos feels a bit messy, when we know that the IDs will be unique across all resources. To tidy this up, though, we need to look at enums.

Liked Data and Enums

TypeScript enums are a way to define a set of named constants which can make your code more documenting. In our case, what we'd like to do now is store our liked IDs as one list of IDs - but with a way to denote if that ID is a photo or a video (just in case it's ever possible for a video and photo to have the same ID — not that it should be):

```
const likedData = [{id: 123, type: 'photo'}, {id: 456, type: 'video'}];
```

But rather than use the strings `'photo'` and `'video'` or any similar combination, we can use an enum (we'll define this in `storage.ts`, as it relates to local storage):

```
enum LikedResource {
  Photo,
  Video
};
```

By defining this enum, we can then refer to `LikedResource.Photo` or `LikedResource.Video` in our code. Under the hood, TypeScript will convert these values to integers for us. This is so that, in our compiled code, referring to `LikedResource.Photo` will actually refer to `0`, and `LikedResource.Video` will refer to `1`. But that's hidden for us, and allows our code to become more documenting by using named enum members rather than any other values.

TypeScript uses numbers starting from `0` for enums by default, but we can also define the values it will use:

```
enum LikedResource {
  Photo = 'PHOTO',
  Video = 'VIDEO'
}
```

Now in the compiled code, any reference to `LikedResource.Photo` will be replaced with the string `'PHOTO'`. This is my preferred solution, because should you ever need to debug the code, it will be clearer.

Diving into `storage.ts`, let's update our types to use this enum:

```
export enum LikedResource {
  Photo = 'PHOTO',
  Video = 'VIDEO',
}

interface StoredLike {
  id: number;
  resourceType: LikedResource;
}

type StoredLikes = StoredLike[];
```

Note: because we're changing the structure of how we store likes, make sure you delete any local storage that already exists from our application, as we'll otherwise get errors in our code as the structure we load won't be what we are expecting. In a real life application, we'd have to deal with this in our code and support both types, but we'll ignore that for this tutorial.

Now we just update the type passed into `saveLikes` and `LoadLikes` to use our `StoredLikes`

type, and now `storage.ts` should look like this:

```
export enum LikedResource {
  Photo = 'PHOTO',
  Video = 'VIDEO',
}

export interface StoredLike {
  id: number;
  resourceType: LikedResource;
}

export type StoredLikes = StoredLike[];

const LOCAL_STORAGE_KEY = '__PEXELS_LIKES__';

export function saveLikes(likes: StoredLikes): void {
  window.localStorage.setItem(LOCAL_STORAGE_KEY, JSON.stringify(likes));
}

export function loadLikes(): StoredLikes | null {
  const data = window.localStorage.getItem(LOCAL_STORAGE_KEY);
  if (!data) {
    return null;
  }
  return JSON.parse(data);
}
```

We now have a lot of TypeScript compiler errors in `main.ts`, as we need to update all the code that works with our local storage data.

Firstly, we need to update the `LikedData` constant to now fall back to an empty array if local storage is empty:

```
function renderApp(results: readonly Resource[] | null): void {
  const div = document.getElementById('app');
  const likedData = loadLikes() || []
  // ...
}
```

And then `onUserLikeClick` needs to change. Let's first remove the entire body of the function so it's completely empty. Firstly, we'll fetch the correct value from the enum based on whether or not the resource is a photo:

```
const enumResourceType = isPhoto(resource)
  ? LikedResource.Photo
```

```
: LikedResource.Video;
```

We can then determine if it's liked by looking through our stored data for an entry with a matching ID and resource type:

```
const likedResourceEntry = likedData.find((entry) => {
  return (
    entry.id === resource.id && entry.resourceType === enumResourceType
  );
});
const resourceIsLiked = likedResourceEntry !== undefined;
```

We can then check if the resource is already liked, and if so, filter our array to find all the entries that are not the `LikedResourceEntry` we queried for. If the resource isn't liked, we can just push it onto the array of liked resources, before then saving our new likes and triggering a re-render:

```
let newLikedResources = likedData;

if (resourceIsLiked) {
  newLikedResources = newLikedResources.filter(
    (entry) => entry !== likedResourceEntry
  );
} else {
  newLikedResources.push({
    id: resource.id,
    resourceType: enumResourceType,
  });
}

saveLikes(newLikedResources);
renderApp(results);
```

With that, the entire function looks like so:

```
function onUserLikeClick(resource: Resource): void {
  const enumResourceType = isPhoto(resource)
    ? LikedResource.Photo
    : LikedResource.Video;

  const likedResourceEntry = likedData.find((entry) => {
    return (
      entry.id === resource.id && entry.resourceType === enumResourceType
    );
  });
  const resourceIsLiked = likedResourceEntry !== undefined;
```

```

let newLikedResources = likedData;

if (resourceIsLiked) {
  newLikedResources = newLikedResources.filter(
    (entry) => entry !== likedResourceEntry
  );
} else {
  newLikedResources.push({
    id: resource.id,
    resourceType: enumResourceType,
  });
}

saveLikes(newLikedResources);
renderApp(results);
}

```

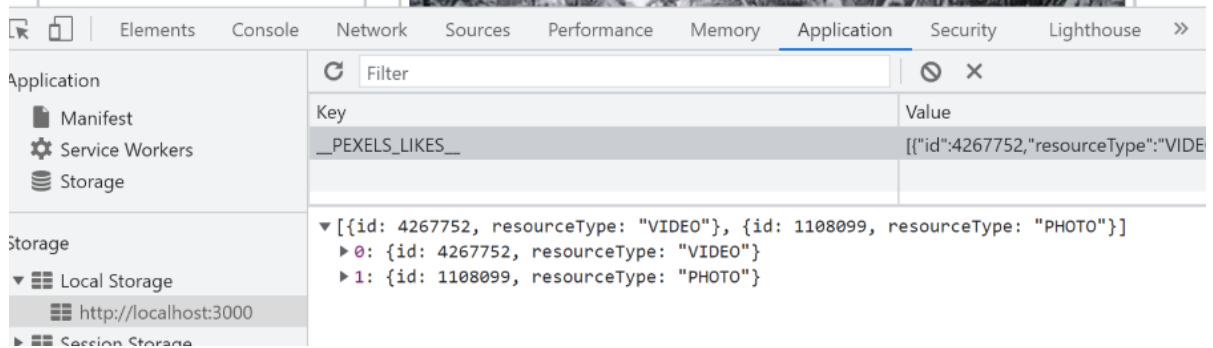
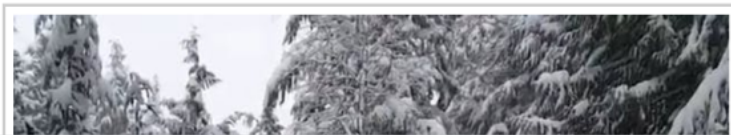
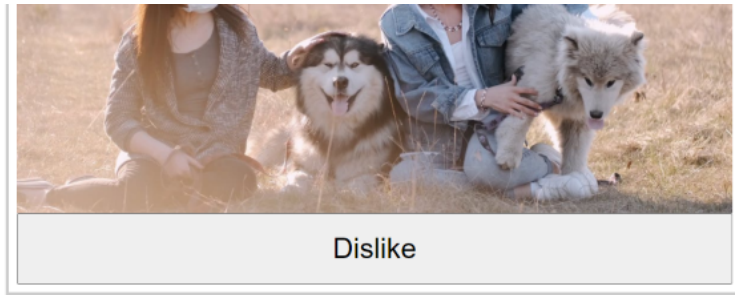
Finally, we have to update our `htmlToRender`, which has to search to find if an item is liked. We can check if the item is liked by looking through the list of liked resources and using the ID and the right type from the `LikedResource` enum:

```

results.map((resource) => {
  const resourceIsLiked = likedData.some((entry) => {
    const enumResourceType = isPhoto(resource)
      ? LikedResource.Photo
      : LikedResource.Video;
    return (
      entry.id === resource.id &&
      entry.resourceType === enumResourceType
    );
  });
  return renderResource(resource, onUserLikeClick, resourceIsLiked);
})

```

And with all those updates, we can now like and dislike photos, and see that local storage is correctly updating.



3-10. localStorage updating with our new storage structure

I highly recommend using enums any time you need to track what the type of an object is, or need to represent a category of items. They're a really great way to keep code tidy, but also self-documenting, and they're one of the TypeScript features I reach for most frequently.

[Feel free to play with the finished application on CodeSandbox.](#)

Features to Add Next

If we were to keep working on this application, the next features that I'd prioritize would be a no results page, in case the user's search comes up blank, and then pagination to enable the user to scroll through more than the first ten results that come back. I'd then look at a way to show the user all their liked content in one page, rather than having to search again for a previous query to see content they had liked before.

Conclusion

I hope that this series of tutorials has helped you understand and see the benefits that writing your code in TypeScript can bring you. From the basic type information that can make refactoring

code much safer and easier, through to features like enums and type predicates that enable your code to be more thorough and self documenting, TypeScript has a lot to offer.