

ALL IN **PROGRAMMING**



BORISLAV HADZHIEV

Intro

Hi folks, I'm [Borislav Hadzhiev](#).

You might have read some of my articles on [bobbyhadz.com](#).

I wrote this book to share everything I know about how to become a better, more efficient programmer.

Don't take what's written in this book as "advice". I am not qualified to give advice to anyone about anything. What works for me might not work for you.

Take what's written in this book as food for thought and a source of ideas.

Do your own research, draw your own conclusions.

[bobbyhadz.com](#)

Copyright © 2022 Borislav Hadzhiev

All rights reserved.

Table of Contents

- [Embrace the Stuck](#)
 - [Always take the time to define the problem accurately](#)
 - [Read the docs](#)

- Abstractions
- Take the time to understand error messages
- Be honest with yourself
- Question decisions made in your and third-party code
 - The change email functionality has been bugged for ~ 3 years
- What to actually learn
 - What to learn
 - What NOT to learn
 - If you don't use it, you'll lose it
 - Learn fundamentals before you learn magic
- Testing
- Focus on adding value
- A poor programmer is a poor programmer
 - A job is a paid internship
- Stand out when applying for jobs
 - When applying for a job with a company, review their website (or service)
 - Start a blog
 - Show that you are invested

- Optimize every action you repeat hundreds of times a day
 - Snippets
 - Linters, Code formatters, Typed Languages
 - Keep track of things
 - Keyboard shortcuts and navigation
 - Some of my most commonly used keyboard shortcuts
- Search for code examples on GitHub
 - Search for Files on GitHub
 - Useful Github keyboard shortcuts
 - Site-wide shortcuts
 - Repositories
 - Source code editing
 - Source code browsing
- Read open source code
- Contribute to open source code

Embrace the Stuck

Continuously getting stuck and unstuck over long periods of time is where I've made the most improvements as a programmer.

The emphasis here being long, but not unnecessarily long. Getting better at programming takes time, but passively trying to get better at programming takes a lot more time.

The difference between active and passive learning, is that with active learning you continuously get stuck and unstuck.

The best way to get better at programming is by:

1. Programming
2. Getting stuck
3. Looking for solutions
4. Implementing your own or using a scalable third-party solution
5. Reusing the solution for similar problems in other projects
6. Improving the solution (if necessary) and applying the improvements to all projects that use it

There are times for both active and passive learning.

I use passive learning to:

- Get comfortable with a technology and get the big picture
- Learn about things I can use the software for

- Learn about the ecosystem

That's not to say you shouldn't watch youtube videos or listen to podcasts while exercising or doing errands. However, in my experience, 2 hours of coding is more valuable than 4 hours of watching other people code.

"Reading is faster than listening, doing is faster than watching" [Naval Ravikant](#)

Most people avoid active learning because it's associated with shortterm pain. To actively learn, you have to do stuff.

Naval Ravikant speaks about how if you have 2 relatively-equal choices to make, you should pick the path that is more difficult and more painful in the short-term.

One of the paths requires short-term pain (active learning), and the other path leads to pain further out in the future (passive learning).

The most natural thing to do is to avoid the short-term pain.

However, the path associated with the short-term pain (active learning) leads to long-term gain.

"You generally want to lean into things with short-term pain and long-term gain" - Naval Ravikant

As a programmer, you have to make friends with the feeling of delayed gratification.

[Wikipedia](#) defines delayed gratification as "The resistance to the temptation of an immediate pleasure in the hope of obtaining a valuable and long-lasting reward in the long-term".

The more problems you encounter when writing code, the more you start noticing patterns.

Every good or bad decision you make is a lesson learned. Take the time to analyze what went well, or what went wrong.

The end goal is to feel confident and satisfied with every aspect of your application. You want to be able to know that if you encounter this problem in another project, you can reuse the same solution and not have to implement a different one.

Compound interest works in a similar way to reusing the same solution between multiple projects.

If you invest \$1,000 and you earn 10% per year:

- You'd have \$1,100 after a year
- You'd have \$1,210 after 2 years
- You'd have \$1,331 after 3 years
- You'd have \$1,464.10 after 4 years

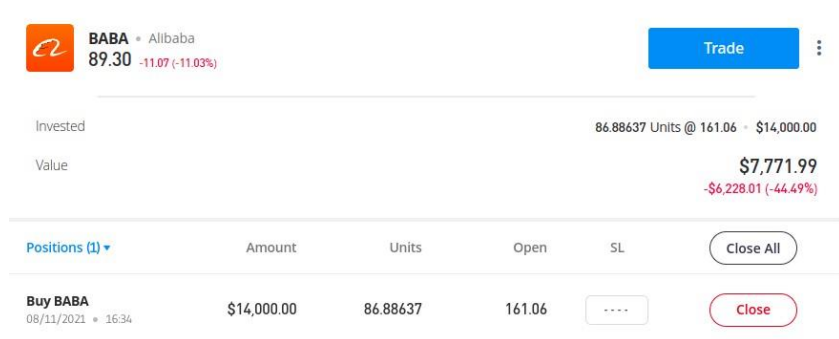
We aren't just making \$100 each year, because the interest gets added to the principal balance, which then earns more interest.

You should look at coming up with a good solution to a common problem in the same way. Once you start reusing your solution in different projects, the growth is exponential.

If you improve your solution, all your projects benefit. You can apply the improvements you've made to any other project that uses this solution.

You'll notice that I write a lot about time and money investments throughout this book.

It's only fair that I show you how my investments are going.



I mean, we're definitely not at 0 (at the time of writing).

As a web developer, I like to think about every single aspect of writing a website.

1. Which parts are time consuming and difficult to understand
2. Is there a way for me to use an abstraction or to come up with a convention that I can follow

The goal is for tasks to be repetitive and boring, not complicated and time consuming.

One way to make complex tasks repetitive and boring is to leverage third-party software where it makes sense.

The best solution is the best solution regardless of who wrote it.

It's best to not dive too deep into niche, one-off problems that you're likely never going to encounter again.

A task can be anything. For example:

- validating emails

- setting up a test environment for a project implementing
- pagination implementing file uploads to a remote data storage
- service full text search authentication and authorization
- You should look at getting stuck on a task as an opportunity to grow.

Prioritize more generic and scalable solutions over more specific and optimized solutions.

You could spend time to look to optimize for everything (e.g. performance). However, you probably shouldn't, unless you work for Amazon or Google.

A good solution is one that solves a specific problem, but is open for extension.

Avoid getting stuck due to premature optimizations and distinctions that will likely add no value, but will force you to implement different solutions for the same (or very similar) problem.

Using different solutions to the same problem in different projects is confusing and not scalable.

"The more you know, the less you diversify" - Naval Ravikant

The more you diversify, the more things you have to think about and keep track of.

The goal is for everything to be as obvious and straightforward as possible.

I have a much better output when I don't have to think very much and can just follow conventions.

Being able to solve a problem is great, but having the solution to a problem is much better.

If you've already solved the same or a similar problem and you don't feel good about reusing the solution in different projects, then you have to look at the solution.

If you haven't done a good job, analyze what went wrong, or use a thirdparty library that does a better job.

Much of how I determine if I should do a better job at something is based on how I feel about it.

I feel good about a solution that:

- works effortlessly abstracts (hides) away a ton of complexity allows
 - me to focus on the business logic is flexible enough for me to make
 - application-specific changes is well-tested
 -
- I feel bad about a solution that:
- - I have to think about every time I use it. If I have to know how the solution is implemented to be able to use it, then it's not good enough (it introduces too much complexity) abstracts (hides) little to no
 - complexity has to be fundamentally changed to be reused between
 - projects • isn't well-tested. I don't want to introduce the same bugs in all of my projects

The goal is to be able to complete commonly encountered tasks with very little resources.

This can be done by establishing easy to follow conventions and coming up with reusable solutions to commonly encountered problems.

Conventions are predefined rules that get you to a desired outcome.

You can come up with conventions for almost all the tasks you have to repeat over and over.

Once I come up with a convention, I will follow it blindly until it no longer feels right. If my convention stops serving me well, I will go back and revisit it.

Some questions to ask are:

- What is wrong with my solution?
- What has changed since I came up with the solution?

If you look at some of the freelancers who make the most money, they are specialized. They do the same thing for different companies, over and over again.

If it's boring and makes money, you're probably doing something right.

If you've been programming for a while, you probably know when something feels off.

The complexity introduced in one part of your project cascades into your whole application. You constantly have to work around something.

Addressing these things over long periods of time is what separates good programmers from not so good programmers.

The solution is often as simple as integrating with a service that solves the problem for you.

When I first started using AWS (Amazon web services), I had to find a way to automate and scale the process of provisioning my infrastructure.

All the servers, databases, serverless functions, auth services, APIs, queues, etc had to be provisioned in an automated way.

My first solution was to use the UI (user interface) to manually provision everything, which took about 3 hours.

Provisioning some of the services took around 30 minutes. I had to think about the order the services should be provisioned in and how they depend on one another.

Doing things manually was extremely inefficient.

This is when I found a service called CloudFormation. This service allows you to write YAML or JSON code to provision your AWS infrastructure.

It was much better than the solution I had. Once I had written the YAML or JSON template, I was able to deploy all my infrastructure using a single command.

However, there were some issues with the solution.

- I now had to manage YAML or JSON templates that were thousands of lines long. Once I had the template, I could provision hundreds of services with a single command, but writing 5,000

lines of YAML wasn't fun, and updating something in 5,000 lines of YAML was definitely not fun.

- I had to understand all 5,000 lines in order to make a change. Every time I'd go back to a project, I had to spend 2 hours to make sure I understood all the moving parts. I was maintaining 5,000 lines of relationships between all the resources that made up my infrastructure.
- CloudFormation was extremely flexible. It allowed me to provision anything. However, it also didn't provide any defaults. I had to explicitly specify every little detail about a service's configuration.

I would much rather override the things I need to override than specify everything, and then have to look at, and manage everything.

- Try testing YAML or JSON code. It's not very fun.

Trying to reuse, maintain, update or extend my solution was a nightmare.

Ideally, you want a solution to a problem to offer different levels of abstraction.

By different levels of abstraction, I mean, you only want to be able to specify the things you care about. These are the things that diverge from the default (90%) use case.

You want to be able to use sane defaults that would solve the problem for 90% of the cases without having to write too much code.

If your use case requires you to move away from the defaults, you want to be able to do that using arguments or different methods or classes.

I'd much rather update the configuration properties of resources when my use case doesn't match the default behavior than have to explicitly set every property.

Try to welcome a new developer to the team with a 5,000 lines long YAML template and let me know how much time it takes them to make the first change.

After a while, AWS came up with a service called CDK (Cloud development kit). The service allows you to provision infrastructure using a programming language (TypeScript, JavaScript, Python, etc) rather than a configuration language (YAML, JSON).

The service was a much better solution to my problem than managing 5,000 lines of YAML because:

- It provides different levels of abstraction with sane defaults that can be updated when necessary. Now I had to manage 500 lines of code instead of 5,000. Most, if not all, the complexity is hidden from me until I have to know about it.

Hiding the complexity is easy. However, hiding the complexity and leaving the solution open for extension is difficult.

- I can now test my code in a more straightforward way because it's written in TypeScript.
- It allowed me to write my application and infrastructure code using the same language (TypeScript). IDE support for YAML or JSON doesn't come close to IDE support for TypeScript.

I am a big fan of the no-code movement. The less code I have to manage to complete a task, the better.

Being able to scale my solution to a problem is what motivates me to always look for good solutions.

Like [uncle Bob Martin](#) says: "The only way to go fast is to go well".

The only way that enables me to go fast is to hide the complexity associated to solving a problem, but leave a door open to be able to customize the solution if requirements change.

Managing the complexity associated to solving a problem is usually done by removing distractions and thinking about things in isolation.

When you get stuck on something, try writing pseudocode. Pseudocode allows us to think about things in isolation and often shows us where functions or classes should be extracted.

This helps, because when writing pseudocode, you are able to ignore a lot of the distractions.

Here is an example of writing pseudocode to implement a function that allows a user to register with both an email and a google account.

```
def register_user (google_id, email):  
    if user_registered_with_google:  
        user_with_email = list_users_by_email(email)  
  
        if user_with_email:  
            link_user_accounts(email, google_id)  
        else:  
            user_with_email = create_user_with_email(email)  
            link_user_accounts(email, google_id)  
    else:  
        user_with_email = create_user_with_email(email)  
        request_email_confirmation(email)
```

If you don't have real abstractions (functions that hide unnecessary details to allow you to focus on a single problem) yet, write pseudocode with well-named functions.

It's quite obvious what the `create_user_with_email` function does.
I don't have to read the function's code to know what it does.

This enables me to not think about its implementation in the scope of the `register_user` function.

The biggest mistake you can make is to read code line by line. You'd have to be a genius to solve any moderately-complex problem if you're reading code line by line.

Trying to read code line by line is like bailing water out of a sinking boat.

It forces you to keep all of the details of all of the moving parts in your brain at every moment.

If a function or a variable name makes its purpose obvious, it's easy to know if you care about that function or variable at any point in time without having to read its code.

You should extract logic into well-named functions that do one thing only. You want to allow yourself to minimize the noise (the things you don't care about at that point in time).

When writing pseudocode, the intent is not for it to be perfect. The intent is to gather your thoughts better than you would if you had started to implement everything at the same time.

The goal is to simplify things and think about them in isolation.

When writing code, you should only have to be smart on very rare occasions. If you have to be smart most of the time, you're probably doing something wrong. It's just not very sustainable.

The most common things to do wrong are:

- Thinking about too many things at the same time
 - because of trying to produce a perfect solution without iterations because of reading code line by line
 - because of not extracting code into well-named, single responsibility functions
- Solving problems that you don't have (premature optimizations)
- Not using third-party modules or services to solve problems you don't specialize in. These are problems that require domain specific knowledge (e.g. accounting), or simply knowledge you don't have and you aren't interested to acquire.

At any point in time, you want to deal with as few moving parts as possible.

Always take the time to define the problem accurately

One thing I've noticed helps me get unstuck quite often is taking a minute or two to define the problem (or query) accurately.

When you search for solutions to problems on the internet, or you're just trying to come up with a solution yourself, always take the time to define the problem correctly.

Not only are you helping search engines, but you're also helping yourself.

Make sure to associate the definition of the problem with the solution. Don't just blindly copy and paste (unless it's just a one-off query, e.g. a complex regex, or nonsensical module bundler configuration).

"If you define the problem correctly, you almost always have the solution" - Steve Jobs

Read the docs

Another thing that helps me get unstuck is spending a couple of minutes to skim over the documentation of the technology I'm learning.

When starting to learn a programming language or a library, skim over the documentation. I mostly look at structure, headings, subheadings.

I do this because:

- No source is more accurate than the official docs
- A 15 minute investment gives me confidence and saves me hours

It's not about reading everything that's in the docs.

It's about knowing:

- what's in the docs what
- technology X can do how the
- docs are structured

It's quite nice when you can leverage your IDE for:

- auto imports (so you don't have to remember where to import X from)
- looking up a method's signature (parameters, types, return value, short description)
- linting and type checking

If I cannot find a solution to the problem directly in my IDE and I know the information is available in the docs, I'd prioritize the docs over googling.

Chances are the docs are going to be the most accurate, up-to-date source of the information you're looking for.

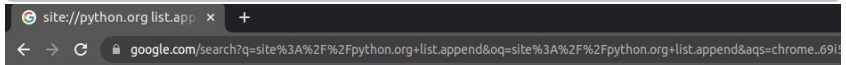
This is why I always spend at least 15 minutes to make note of what the docs cover and where to look for the things I am likely to need.

If the docs are no good, or don't cover the issue, then I google stuff.

Sometimes the official docs don't rank very well in google, and don't have a search input I can use to filter.

On rare occasions, I do site-scoped search in google. For example, here's how I'd look for the `list.append()` method if I want to see results from python.org.

```
site://python.org list.append
```



site://python.org list.append



[All](#) [Images](#) [Videos](#) [Maps](#) [Books](#) [More](#)

Tools

About 11,100 results (0.37 seconds)

<https://docs.python.org/tutorial/datastructures>

5. Data Structures — Python 3.10.5 documentation

The list data type has some more methods. Here are all of the methods of list objects: `list.append(x)`. Add an item to the end of the list.

<https://docs.python.org/library/stdtypes>

Built-in Types — Python 3.10.5 documentation

Appending 'j' or 'J' to a numeric literal yields an imaginary number (a complex number with a ... for a complete list of code points with the Nd property.

Abstractions

A good abstraction is like a Sean Paul song - I don't know what's going on but I like it.

I have no idea what he's saying, but the melody sounds good and he's confident, so it works.

It's the same when using abstractions. It only works if you can have confidence that you'll use this black box that was written by you or someone else and it will just work.

You can only feel confident doing something when you think you understand it. You can only truly understand something, if you can filter out the unnecessary details, so you can focus on what's important at that point in time.

Abstractions enable us to ignore unnecessary details, so we can think about the solution to a complex problem in isolation.

Let's look at an example of a fairly complex function that uses abstractions to hide most of the complexity.

The function handles a user's request to change their email.

Note: this is pseudocode (skim over)

```
function requestEmailChange (authToken, newEmail) {  
  const emailIsValid = validateEmail(newEmail);  
  
  if (!emailIsValid) {  
    throw new Error('The supplied email is invalid');  
  }  
  
  if (emailExists(newEmail)) {
```

```

    throw new Error(`A user with email ${newEmail} already
exists`);
}

const confirmationCode = generateConfirmationCode();

const user = getUser(authToken);
user.emailChangeCode = confirmationCode;

sendEmail({
  receiverAddress: newEmail,
  emailSubject: `You have requested an email change for
${site}`,
  htmlEmailBody: getEmailVerificationTemplate ({
    site: site,
    name: `${user.first} ${user.last}`,
    confirmationLink: `${site}${route}?
email=${email} &code=${user.emailChangeCode} `,
  }),
});

return res.status(200);
}

```

There are quite a lot of moving parts. However, the idea of using abstractions is to focus on what functions do, and not on how they do it.

For example, the **validateEmail** function takes an email address as a parameter, validates the email and returns the result.

```

const emailIsValid = validateEmail(newEmail);

```

The function could be implemented in a million ways. It could use the nastiest regular expression, but that's not something we have to focus on to be able to use the function.

Similarly, the **emailExists** function takes an email address and checks if the email exists.

```
if (emailExists(newEmail)) {  
    throw new Error(`A user with email ${newEmail} already  
exists`);  
}
```

The function could be making a query to a relational or a non-relational database, it could be reading from the file system, but that's not something we have to focus on. What's important is that the function does what its name suggests.

As long as we know what arguments the function takes and what it does, we are able to use it.

When using abstractions, the interface is more important than the implementation.

The `generateConfirmationCode` function doesn't take any arguments and returns a confirmation code.

```
const confirmationCode = generateConfirmationCode();
```

The function could use the most advanced algorithm in the world to generate a confirmation code that can never be cracked, but we don't have to know about its implementation to be able to use it.

The `getUser()` function takes an authentication token as a parameter and somehow fetches the corresponding user.

```
const user = getUser(authToken);
```

Just like with the `emailExists()` function, it's not important how exactly the user is retrieved. What's important is that the function does what the name

suggests and until we have a reason to change its behavior, we don't have to worry about it.

We also have a `sendEmail()` function.

```
sendEmail({
  receiverAddress : newEmail,
  emailSubject: `You have requested an email change for
${site}`,
  htmlEmailBody: getEmailVerificationTemplate ({
    site: site,
    name: `${user.first} ${user.last}`,
    confirmationLink: `${site}${route}?
email=${email} &code=${user.emailChangeCode}`,
  }),
});
```

What a function named `sendEmail` does is quite obvious. How exactly it does it is not so obvious.

We could be using a service to send emails, or we could have our own Simple Mail Transfer Protocol server.

I couldn't tell you how exactly sending an email works, but fortunately I don't have to know to be able to send an email.

Even if you knew how the Simple Mail Transfer protocol works, you wouldn't want to think about it every time you have to solve a task that involves sending an email. You'd want to focus on the task at hand.

Lastly, we have a `getEmailVerificationTemplate` function that generates some kind of an HTML template containing an email verification link.

```
htmlEmailBody: getEmailVerificationTemplate({
  site: site,
  name: `${user.first} ${user.last}`,
  confirmationLink: `${site}${route}?
email=${email} &code=${user.emailChangeCode}`,
});
```

The function likely uses some complicated markup language for generating beautiful, responsive emails. Luckily for us, we can use the function without knowing about its implementation details.

Imagine if you had to know how your operating system, or any other GUI (graphic user interface) works in order to use it.

It's the same when using built-in methods. You don't have to know how any of the built-in methods in JavaScript or Python are implemented to be able to use them.

If you have to work on the `requestEmailChange` function, you'll likely have to make changes to one of the functions it calls, or maybe you'd just have to pass one of the functions different arguments.

The benefit of using abstractions is that you'll quickly be able to find the code that interests you.

The more complexity you can hide, the higher the chance your code actually works.

There's nothing that gets you stuck like complexity. The only way to manage complexity is to think about tasks in isolation.

One of the most common ways to introduce complexity is to read code line by line.

Let's look at the function from the example again.

Try to imagine what the `requestEmailChange` function would look like if we were to replace all the functions it calls with their actual implementation.

```
function requestEmailChange (authToken, newEmail) {
  const emailIsValid = validateEmail(newEmail);

  if (!emailIsValid) {
    throw new Error(`The supplied email is invalid`);
  }

  if (emailExists(newEmail)) {
    throw new Error(`A user with email ${newEmail} already exists`);
  }

  const confirmationCode = generateConfirmationCode();

  const user = getUser(authToken);
  user.emailChangeCode = confirmationCode;

  sendEmail({
    receiverAddress: newEmail,
    emailSubject: `You have requested an email change for ${site}`,
    htmlEmailBody: getEmailVerificationTemplate ({
      site: site,
      name: `${user.first} ${user.last}`,
      confirmationLink: `${site}${route}?email=${email} &code=${user.emailChangeCode}`,
    }),
  });

  return res.status(200);
}
```

It wouldn't be a pretty sight if we replaced the functions with their implementation. Try making changes to that bad boy. That would be a nightmare.

When you read code, you shouldn't do it line by line. You should extract logic into well-named functions that do one thing only.

You want to allow yourself to minimize all the noise that you don't care about at any point in time.

By using well-named functions in our `requestEmailChange` function, we are able to shift our attention to details we care about.

If a moderately complex function doesn't make use of good abstractions, you'd have to read the entire function to find the two lines of code that interest you at that point in time.

I am not great at naming functions and variables, but I often notice that taking a few minutes to come up with a concise, accurate name ends up saving me time.

I no longer have to read the function's code to know what it does. All I have to do is look at the function's name.

Similarly, if a variable has a descriptive name, I don't have to follow the code from its declaration to its current value to know what its purpose is.

Function names should be like the subheadings in an article - they tell you what the function does and if you need to know more, you can refer to the implementation.

If you find yourself getting confused by what a function does, it probably isn't named well, or does too many things. Find a better name for the function or split it into multiple functions that do 1 thing well.

You don't necessarily have to know how the function does what it does.

And even if you do, you shouldn't think about it unless you are directly working on the specific function.

Not having to think about all of the moving parts is how we manage complexity when programming.

To be able to keep complexity low, it's important that these functions are responsible for doing 1 thing and doing it well.

If you see a function named **doXandY**, you should most likely have 2 functions - **doX** and **doY**.

We looked at how to read code that uses abstractions. To write an abstraction, all you have to do is write a function that does a single thing well.

The goal is to provide a good interface to the consumer of your function, so that they don't have to know about any of the inner workings to be able to use it.

Once they read the function's name, they should know what it does.

Once they hover over the function in their IDE, they should know what parameters the function takes.

Let's look at an example of creating abstractions to reuse code and manage complexity.

The following function creates a presigned URL and sends it to the user. The idea is that the server signs a URL that allows the user to upload a file to a remote data store (e.g. Amazon's Simple Storage Service, aka S3).

The flow is as follows:

1. The user wants to upload their avatar to a remote data store
2. Our frontend (the browser) requests a presigned URL from our server
3. Our server comes up with a unique filepath, so that the user doesn't override someone else's avatar
4. Our server generates a presigned URL and sends it to the frontend
5. The user uploads their avatar to the remote data store

This is the version of the function that doesn't use abstractions.

Note: this is pseudocode (skim over)

```
function getPsignedUrl () {
  // 1. Get file path (random string of length 10)

  let filePath = '';
  const characters =
    'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!
    -.*() ';

  const length = 10;

  const charactersLength = characters.length;
  for (let i = 0; i < length; i += 1) {
    filePath += characters.charAt( Math.floor(Math.random() *
    charactersLength));
  }

  // 2. Create presigned URL

  const params = {
    Bucket: BUCKET_NAME,
    Key: filePath,
    Conditions: [
      // allow images of up to 5 MB max
      ['content-length-range', 0, 5000000],

      // only allow user to upload images
      ['starts-with', '$Content-Type', 'image/'],
    ],
    // number of seconds before presigned URL expires
    Expires: 15,
  };

  const presignedUrl = createPresignedUrl(params);

  return res.status(200).send({presignedUrl, filePath});
}
```

The function is quite hard to read. There are a lot of things going on.

If you had to make a change, you'd have to read the entire function.

Let's try to hide some of the complexity using abstractions.

```
function getPresignedUrl () {
  const filePath = getFilePath();

  const presignedUrl = getPresignedUrl();

  return res.status(200).send({presignedUrl, filePath});
}

// don't need to know about this to use the function
function getFilePath () {
  let filePath = '';
  const characters =

  'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!
  -.*() ';

  const length = 10;

  const charactersLength = characters.length;
  for (let i = 0; i < length; i += 1) {
    filePath += characters.charAt( Math.floor(Math.random() *
charactersLength));
  }

  return filePath;
}

// don't need to know about this to use the function
function getPresignedUrl () {
  const params = {
    Bucket: BUCKET_NAME,
    Key: filePath,
    Conditions: [
      // allow images of up to 5 MB max
      ['content-length-range', 0, 5000000],

```

```

    // only allow user to upload images
    ['starts-with', '$Content-Type', 'image/'],
  ],
  // number of seconds before presigned URL expires
  Expires: 15,
};

const presignedUrl = createPresignedUrl(params);

return presignedUrl;
}

```

We extracted 2 functions to declutter our `getPresignedUrl` function.

The `getPresignedUrl` function is now quite easy to read.

```

function getPresignedUrl () {
  const filePath = getFilePath();

  const presignedUrl = getPresignedUrl();

  return res.status(200).send({presignedUrl, filePath});
}

```

The `getFilePath` and `getPresignedUrl` functions have a single responsibility and can be reused throughout our codebase.

We won't have to think about how these functions are implemented the next time we use them. Their names clearly describe what they do.

In a real world scenario, the functions you extract will likely take arguments.

Using function arguments is how we define flexible abstractions.

You can think of an abstraction as a black box. If you have to think about the parts of the black box as the consumer of the API, then that's not a good abstraction.

Most of the time when I'm writing hard to understand code, I'm using bad abstractions.

The solution often is to write better abstractions or pick a third-party package that offers better abstractions.

Confidence is the most important thing to look at for whether you're doing something right or wrong when programming.

It is possible to be confident and ignorant and write poor-quality code. However, in my experience, most developers know when something is off (most of the time).

Whether you're writing code, testing code, designing a UI, deploying your project, confidence is what you should be looking at for whether you have something to address.

If you feel bad about something, take a moment and write down exactly what bothers you about this part of your project.

Can you address the things you don't like about X? Should you be doing X at all, or should you use a third-party library or a service?

The best way to manage complexity is to use good abstractions.

You shouldn't spend your resources on thinking about all the moving parts all of the time. Your resources are better spent on defining the task and only focusing on the things that relate to solving it.

If a problem requires me to think about multiple things at once to be able to solve it, then I want to think about as few things as possible.

Abstractions enable us to filter out irrelevant details from the solution, so we can manage fewer lines of code.

Low-code that works is better than more code that works.

A great example of a needed abstraction is how I have to edit configuration files and learn CLI commands to update my scroll speed on Ubuntu. Here is [my config file](#) in case you're interested.

Ideally, the details would be hidden behind a GUI (graphic user interface), so I can click on a button and do the thing.

This applies to everything. The easier you make it for people to use your product or service (or function), the more people will use it, and the less customer support you'll have to do.

Imagine if you had to learn everything about a computer game in order to play it. That would suck. The game wouldn't be very popular.

Have you ever been on a sales page and ready to buy the product or service, but you can't find the buy button? There are testimonials, infographics, but the one thing I'm actually looking for I can't find.

That's exactly what working with code that doesn't use good abstractions feels like.

This is also the main reason successful companies are successful - they just let you do the thing. The default behavior is for everything to just work.

Most companies that suck, don't just let you do the thing. There are always surprises and things you have to know about. Things don't just work.

No one likes the word "prerequisite". You know a solution is bad when it has a lot of "prerequisites".

The more things I have to know about and keep in mind to be able to use the solution, the less I want to use the solution.

If your colleagues can call your `send_email`, `upload_file` or `refresh_auth_token` functions without having to know about their implementation, they'd be very happy about it.

Abstractions help us get rid of distractions.

The most common way to remove distractions when coding is to define a well-named function that does 1 thing only.

When you use a good third-party module, you get to remove distractions for free.

When you use a good third-party service, you basically pay to remove distractions.

The takeaway is that programming is about solving problems and managing complexity.

To solve a problem is not that difficult most of the time. What's difficult is to solve the problem in a straightforward way.

Good abstractions enable us to solve complex problems in a straightforward way.

Take the time to understand error messages

Programmers who care more improve quicker.

There is a very good reason for why we get (most) errors.

There is a certain feeling I get every time I have to google the same error message in a short period of time.

I know that I should probably spend 5 minutes and try to understand the error message.

There are 3 common ways to understand how a library or a programming language works:

- use it for a long period of time read the source code (usually in an
- `src` or a `lib` directory) read the tests (usually in a `test` or a
- `tests` directory)

The more of the 3 you do, the more comfortable you'll feel using the language or library.

The source code is often difficult to read and might span multiple programming languages, multiple different modules located in different repositories, etc.

However, the `tests` are usually quite readable and descriptive.

In essence, tests are used to check if the expected behavior aligns with the actual behavior.

Errors are the output that's produced when we fail a test that's implemented by the library.

In other words, errors occur when we use a part of the library or the language in a way that it wasn't intended to be used.

The most common causes for getting an error are:

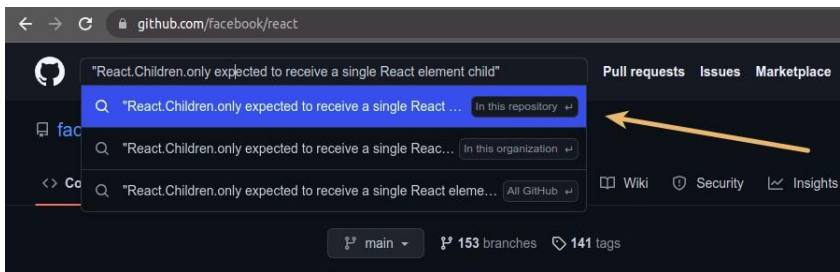
- misunderstanding how a specific method or class should be used mixing
- up data types (happens mostly in untyped languages) not validating user
- input properly

Many programming languages and open source libraries expose a `.json` file with all (or most) of the error codes, e.g.

- [React.js](#)
- [TypeScript](#)

Since it's not always easy to understand exactly why we get an error, I like to look for the code that throws the error in the Github repository of the library.

For example, I can open the [GitHub page for React.js](#) and do a repository-scoped search for the text of the error message.

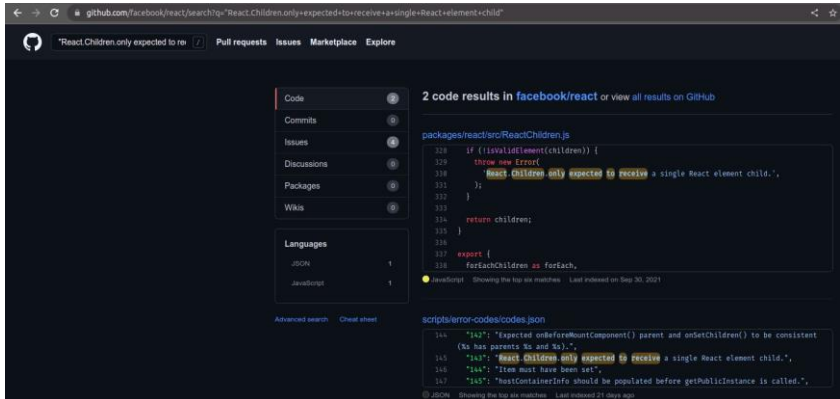


Note that you don't have to type in the entire error message. Some error messages contain variables (e.g. types, method names or property names), so your query might be a little different than the actual message.

It's quite common that the code that throws the error doesn't contain the error message you got word-for-word.

Make sure to click on the "In this repository" filter to look for code only in the related repository.

If your error message contains spaces, which it most likely does, wrap it in quotes when querying.

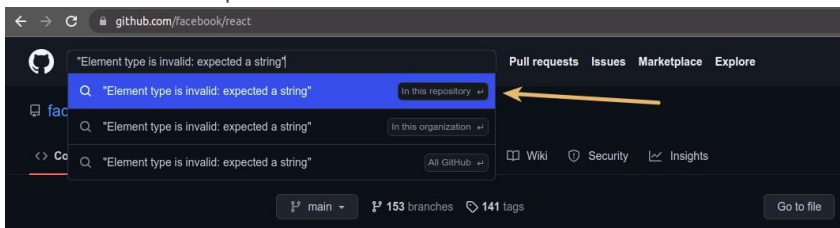


You can now read through the code that throws the error.

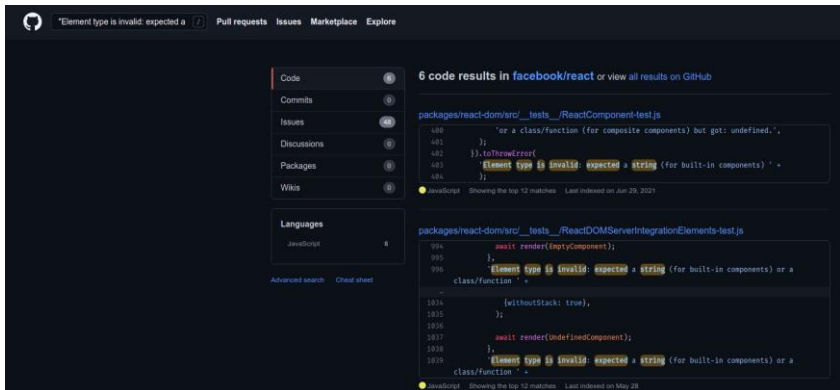
Most open source libraries have code that's easy to read with lots of useful comments. If you can't understand the source, try reading the tests (if there are any).

There will usually also be code that tests whether the error gets thrown. In other words, code that simulates the behavior that is supposed to cause the error.

Here is another example:



The results also contain test cases that show why and when the error is expected to be thrown.



Most modern libraries and programming languages have accurate and descriptive error messages.

Once you have encountered most errors that a package or a language throws, you will have a very good understanding of how it works.

Understanding why you got the error is more important than solving it.

Solving errors is easy, all you have to do is google for the error message and most of the time the solution pops up immediately.

However, solving the same error many times, over a long period of time, is difficult and inefficient.

It's not only the time it takes to solve the error, it's also the time it takes to get the error.

The time you invest to understand error messages will give you a better grasp of the technology and is a great long-term investment.

The confidence you get from understanding how the technology works is what determines whether you're going to have fun writing code, or you're going to dread every single moment.

What's written in this chapter doesn't apply to all errors. Some errors simply don't make sense. These are errors mostly around tooling and unmaintained packages.

There are also errors we don't care about. These are errors that occur when using technologies we aren't interested in.

You shouldn't spend more time than necessary on errors you get due to bugs in tooling and packages. Blindly googling around for a solution is all that's needed.

A good way to know if a software is cared for is how much effort goes into writing actionable, clear error messages.

I watched a talk [Guillermo Rauch](#) gave. He said something along the lines of "View error messages as an opportunity for education".

It only makes sense that library authors want to educate us with clear, actionable error messages.

That helps them in multiple ways:

- reduces customer support (less Github issues, less general questions)
- increases user adoption
- improves developer experience
- gives developers a better understanding of how the technology works, which in turn increases the likelihood they'll contribute to the project

A good error message is one that's obvious and helps you stay in your IDE (you don't need to Alt-Tab to google stuff).

The tin foil hat question is: Are the error messages of some libraries and services terrible on purpose?

I guess the more confusing you make it, the easier it is to sell consulting.

Be honest with yourself

I make mistakes every day and I still hate it. Every time I do, I subconsciously try to justify my decision.

I can already imagine how someone is going to fact-check me about something I wrote in this book and I'm going to look like an idiot.

I especially hate it when other people tell me I'm wrong.

I write blog posts, and if you do something long enough, you're probably going to make mistakes.

Sometimes I get emails that point out flaws in my articles and it's quite hard for me to be objective because I want to always be right. My initial reaction is to be on the defensive and look for issues in the proposed solution.

Instead, much more productive would be to try to present to myself the best possible case for why their solution is better than mine.

Are they onto something? Is the best version of their solution better than the best version of my solution?

The aim should be to find the truth, not to be right.

I never ask if "I like it" or "I don't like it." I think "this is what it is" or "this is what it isn't." - Richard Feynman

One could easily dismiss other people's solutions. Especially, if the solution is not a 100% finished or has flaws.

However, much more productive is to ask yourself if there is an interesting part of their solution, a part that can be used to define a more complete, better solution to the given problem.

It's especially difficult to be objective if the person presenting the information is not being very nice.

Unfortunately, even if that's the case, that doesn't make the information not true. The way the information is presented is irrelevant to whether it's true.

Arrogant programmers, or just arrogant people in general, might make good points.

For my own sake, I try to not conflict the two things. In the grand scheme of things, it's all about trying to better yourself and benefit, even from opinions of people you don't like.

[Ray Dalio](#) speaks about the "art of thoughtful disagreements".

Here are some of the main points:

- Thoughtful disagreement - the capacity to hold two conflicting things in your mind at the same time.
- Don't be afraid to be wrong.
- Ask yourself: "How do I know whether I'm right or wrong?"
- Find the smartest people (independent thinkers) who disagree with you and listen to their arguments.
- If the person has higher expertise than you, ask questions.
- If you want to learn, you should be asking questions, not arguing.
- Learn to deal with uncertainty.
- Aim to find the truth, not to be right.
- Replace the joy of being proven right with the joy of learning what's true.

- Having a big ego prevents you from acknowledging your weaknesses objectively, so that you can figure out how to deal with them.
- Believing in and working towards things that aren't true, ends up costing you time.

Another thing you should be honest with yourself about is whether you should address the things you don't feel good about when writing code.

In my experience, it's very rare to write erroneous code when I feel confident.

Almost every time I've had bugs in my code, I kind of knew about it. There is a certain feeling you get when you don't understand exactly what's happening in your code.

- Maybe you're using a third-party library that you don't feel great about.
- Maybe you're trying to think about too many things at the same time.
- Maybe you haven't refactored your code for the past 3 months.
- Maybe you you're making changes to your code without having any tests in place.
- Maybe you are in a hurry, so you copy-pasted code from reddit.
- Maybe you're just getting started with technology X.
- Maybe you just had a bad day and you can't be bothered writing code right now.
- Maybe it's just a one-off, super specific thing you have to do, and you don't want to spend too much time on something you're likely never going to have to do again.

You can either choose to address not feeling good about something, or choose to ignore it. Either way, you have to make a conscious decision on a case-by-case basis.

Unfortunately, it's possible to have decades of experience as a programmer and still be not very good at it. Programmers who care, usually improve quicker.

There are many things I try to constantly improve on, and many other things I avoid like the plague.

Both categories are covered more in-depth in the "What to actually learn" chapter.

The second category consists of things I outsource, postpone or ignore.

For example:

- Things that require me to have domain-specific knowledge.
- Things that are more focused on tooling than directly adding value to customers.
- Things that will likely have changed or will not be true in a year.
- Things that don't make sense.

Question decisions made in your and third-party code

I once [wrote an article](#) where I shared my experience using AWS Cognito. AWS Cognito is advertised as "Simple and Secure User Sign-Up, Sign-In, and Access Control".

The article I wrote is more than 4,000 words long and a clear indication that I have no life.

You don't have to know anything about AWS Cognito to be able to follow my thought process and notice some of the things I look at when using third-party software.

In addition, you probably have implemented or used authentication and authorization, so I don't think you'll feel lost even if you haven't used AWS.

Here is a single subheading from [the article](#).

Generally speaking, things to look for:

1. Define the problem (What is the default behavior?)
2. What is the expected behavior?
3. What are the proposed changes?

The change email functionality has been bugged for ~ 3 years

It's very common to implement authentication with email as a username.

Unsurprisingly, AWS Cognito supports this behavior.

How do you want your end users to sign in?

You can choose to have users sign in with an email address, phone number, username or preferred username plus their password. [Learn more.](#)

☒ **Username** - Users can use a username and optionally multiple alternatives to sign up and sign in.

☐ Also allow sign in with verified email address

☐ Also allow sign in with verified phone number

☐ Also allow sign in with preferred username (a username that your users can change)

☒ **Email address or phone number** - Users can use an email address or phone number as their "username" to sign up and sign in.

☒ Allow email addresses

☐ Allow phone numbers

☐ Allow both email addresses and phone numbers (users can choose one)

You wouldn't want someone to register with an email they don't own.

That's not secure and enables a user to reserve emails they don't own and to effectively block the actual email owners.

To combat this, you would need an email verification step (like every other site on the internet). Cognito also provides this functionality:

Which attributes do you want to verify?

Verification requires users to retrieve a code from their email or phone to confirm ownership. Verification of a phone or email is necessary to automatically confirm users and enable recovery from forgotten passwords. [Learn more about email and phone verification.](#)

☒ Email ☐ Phone number ☐ Email or phone number ☐ No verification

Ok, so what's the problem?

1. The user requests an email change, but hasn't yet verified the new email with the verification code.
2. Cognito automatically updates the email attribute in the user pool, even though it hasn't been verified.
3. If the user then logs out, they can only log in with their new, notverified email.
4. The new, not-verified email is already taken in the user pool. This blocks any users who might own that email from registering on your website.
5. The old email the user previously used is now available.

The expected behavior for a completed email change would be:

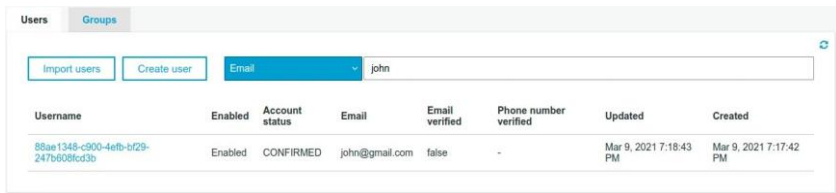
1. The user requests an email change.
2. The user clicks on the link sent to their new email address.
3. AWS Cognito verifies the email and updates the **email** attribute in the user pool.

The expected behavior for an unfinished email change would be:

1. The user requests an email change.
2. The user doesn't click on the link sent to their new email address.
3. AWS Cognito does nothing.

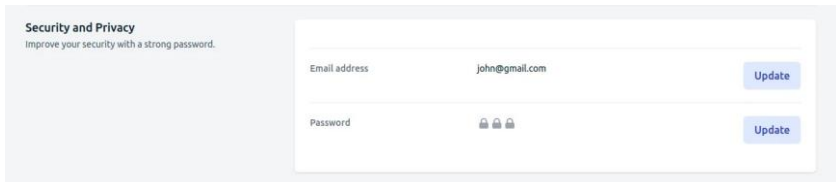
The actual behavior is:

1. Why did Cognito change my email to **john@gmail.com** if I never verified it?



Username	Enabled	Account status	Email	Email verified	Phone number verified	Updated	Created
88ae1348-c900-4efb-bf29-247b6080cd3b	Enabled	CONFIRMED	john@gmail.com	false	-	Mar 9, 2021 7:18:43 PM	Mar 9, 2021 7:17:42 PM

2. Why am I able to log into my application as **john@gmail.com**?



Security and Privacy
Improve your security with a strong password.

Email address	john@gmail.com	Update
Password	🔒 🔒 🔒	Update

I can log in with an email I haven't verified, even though I explicitly selected that I want users to verify their email.

In pseudocode, what the source code actually does:

```
if (user.requestsEmailChange()) {  
    sendConfirmationCode(newEmail);  
    updateUserEmail(newEmail);  
}
```

In pseudocode, what the source code should actually do:

```
if (user.requestsEmailChange()) {  
    sendConfirmationCode(newEmail);  
}  
  
if (user.hasClickedConfirmationLink()) {  
    updateUserEmail(newEmail);  
}
```

End of subheading.

If you can answer the following 3 questions more than a handful of times, you should probably be looking for another library. 1. Define the problem (What is the default behavior?)

2. What is the expected behavior?
3. What are the proposed changes?

This doesn't only happen when the package you use is poorly written. It just might not be what's right for your use case.

Doing this is especially useful when you come across great libraries and services.

You can learn from how other developers have solved problems you encounter.

1. Define the problem.
2. What is my solution to the problem?
3. What is their solution to the problem?
4. Why is or isn't their solution better than mine?

Sometimes, when using software written by some companies, everything just makes sense. Most things just work without you having to think about it. No surprises.

A company by the name of [Vercel](#) (Next.js) comes to mind.

There's a lot to learn from using software written by these companies. Once you use software written by them, it's tough to go back.

Other times, things just don't seem right.

When I first started programming, I didn't think about third-party packages or services too much. I just used what everyone else used.

Most of the popular packages and services solve a problem, but do they solve your problem?

Always question everything that makes you feel like you're doing things wrong.

Anything that slows you down, is inconsistent and doesn't make sense, has to go.

Another lesson learned - don't write long articles about how a service you're using is not very good. Just move on.

What to actually learn

I became a better programmer when I stopped trying to learn everything.

The only way for me to make meaningful progress quickly is to selectively learn what's important and reference all other information in the docs, by googling, or in code snippets on GitHub.

When I started learning learning to code, I wanted to learn everything.

I wanted to remember:

- what parameters each function takes which are
- optional and which are required what each
- function does

And that even for functions that I'd likely never use.

In other words, I wanted to be able to use programming languages and frameworks without the need to refer to the documentation or use my IDE.

If only I could know everything off the top of my head, then I wouldn't even need internet access.

To a certain extent companies and job interviews are to blame. Many interviewers ask dumb questions like "What's the third parameter function X takes?".

Many companies also post job descriptions requiring 30 technologies across the whole stack.

After a while, I realized that spending a couple of hours to define the things I should learn and the things I shouldn't learn is a worthwhile investment.

These things will be different for everyone because everyone has different goals.

However, everyone could benefit from defining and following their own rules for what to learn and what not to learn.

Here are some of my rules.

What to learn

- the fundamentals - This is everything that can be applied to other programming languages or libraries in the same field. These are things that are universally true and are not language or framework-specific
- how the documentation is structured and what it contains. Should I be looking for information in the docs or should I be googling?
- the rest I'll learn by constantly looking stuff up. Whatever pops up often and helps me complete a task, that's what I'll learn (without really trying)

What NOT to learn

- the intricacies of method X (parameters, order, default values, edge cases) - hover over the method in your IDE or use the documentation.
- where to import method X from - use auto imports in your IDE.
- all sorts of boilerplate and initialization code - use snippets in your IDE.
- everything that is not applicable to other similar frameworks and languages. If framework X stops being relevant, is what you've learned applicable to the similar frameworks Y and Z?

Spending time to learn all sorts of magical abstractions that are framework X-specific should be avoided (unless the monetary benefits far exceed losses due to non-transferable knowledge). everything I'm not going to use in the

- next 30 days.
- everything I can use a GUI for (that doesn't need to be automated). If I can get from point A to point B by using a GUI (graphic user interface), I'm not learning the CLI.
- learning about tooling, e.g. module bundlers, linters, transpilers should be avoided like the plague. Only practice "just enough and just in time learning" with these because they often change.

Any meaningful time investment to learn these is like burning money. Look for pre-made solutions. Someone has already solved the problem you're trying to solve.

If I can use module bundlers and all sorts of other tools without having to know, understand or configure them myself, I'm definitely doing that.

I'm ok with not having the optimal setup, because it doesn't make sense for me to spend the time to do largely undifferentiated heavy lifting.

I don't want to spend time learning stuff that changes rapidly and makes very little sense.

It should also be noted that you don't "learn" stuff that doesn't make sense, you remember it (or attempt to).

- one-off things - Is it something so niche that I'm likely never going to do again? If so, I'm looking to leverage someone else's solution at all costs.
- is it something domain-specific or with financial or health implications? If so, I'm looking to outsource and pay to use a legitimate service.

- everything that is a solution to a non-existent problem.

Premature optimizations are the worst.

No one needs a Kubernetes cluster with infinite autoscaling and 100% uptime service-level agreement for their blog. Most people also don't need an infinitely scalable database in case their business blows up overnight.

- everything that doesn't make sense. When working with thirdparty libraries, you're writing code around decisions made by other programmers and sometimes these decisions don't make sense.

In these situations, you have 2 options: you can just copy paste the code from the docs, or you can look for a different library.

The list items above are very generalized. Better decisions can be made on a case-by-case basis.

You can't always know if you'll be able to apply something you're learning. If you realize that you're never going to use something, just drop it and don't waste your time.

You will pick up many of the things in the "What NOT to learn" list over time. Especially methods and APIs that you come across often.

It is hard to know in advance which methods you'll use often, so learning everything about everything isn't a good idea.

If it matters, you'll encounter it enough times and you'll learn it.

Things change too quickly, so it's much more efficient to do "Just in time" learning.

Trying to learn new things is like trying to bail out water from a sinking boat. You're either going to need a big bucket, or you'll need to find a way to patch the hole.

If you do have a big bucket, you could use it to do cooler things than learning stuff you won't apply.

A good question to ask yourself is: "Is there demand for what I supply?".

Because if there isn't, you get to keep all of your supply.

You don't make money for knowing things that you don't get to apply.

Many people know a lot of things, but it's only what you get to apply that matters. For instance, I spent 4 years to get a degree in tourism management. What a waste of time.

I could have literally played World of Warcraft for 16 hours a day, for 4 years instead of going to uni, and I'd be in the same position I'm in today.

"The direction you're heading in matters more than how fast you move"
- Naval Ravikant

If you don't use it, you'll lose it

1. Don't optimize prematurely. Try to leverage tools without actually learning them.
2. Don't bother watching courses or reading books about technologies you aren't using. Watching a course or reading a book about a technology is pointless if you don't apply what you've learned to a project you're working on.
3. Prioritize. Not every technology should get an equal amount of your attention. What's Important is not always urgent.

Learn fundamentals before you learn magic

Learn fundamentals before you learn magic.

If you already know the fundamentals, leverage good abstractions.

Frameworks come and go, fundamentals are forever.

By magic, I mean high-level frameworks that do a lot for you, under the hood.

High-level frameworks are ones that offer relatively high level of abstraction.

One very big mistake I made early on is to try to learn to code by using a high-level framework (Django).

You can do so many things by leveraging high levels of abstraction, but learning the fundamentals of programming is not one of them.

High-level frameworks are terrible for new programmers because, on the one hand, they enable you to do stuff (e.g. file uploads, authentication, forms), so they give you an illusion that you're making progress, but on the other hand, you have so many missing pieces of the puzzle, that you're barely making any progress.

As a new programmer, you should be learning transferable information (fundamentals), and not high-level frameworks.

Obviously, the idea of using high-level abstractions is to not have to know about all the implementation details.

However, it's good to have a general understanding of the task you're trying to solve and an understanding of the foundations that these abstractions are built on top of.

This is one of the reasons frameworks like Express or Flask are as popular as they are.

Express.js is mostly JavaScript and Flask is mostly Python.

If you know the language, you can apply your knowledge without having a big barrier to entry.

If you don't know the language, you can pick it up by using the framework.

How you define "fundamentals" largely depends on what you're learning. A generic definition would be - transferable, universally true knowledge.

For example, some of the fundamentals of programming are:

- Variables
- Data types
- Conditions
- Loops
- Functions Classes
- Scope
- Solving problems
- Managing complexity (abstractions)
- Testing
-

I would also consider getting good at writing pseudocode very important.

What to learn is only a piece of the puzzle. The single most important thing is whether you care or not.

If you are genuinely curious about the problems you encounter over a long period of time, you'll do great.

Ultimately, programming is about solving problems and trying to manage complexity.

If you get annoyed when your solutions are confusing and complex, you'll come up with better solutions.

If you can solve problems and manage complexity better than the other person, you're probably a better programmer.

Testing

The very short version of testing is that we do it gain confidence that our software actually works.

It enables us to make changes quicker and without having to manually test all user flows in our application every time we make a change.

A good test is one that tests a specific use case. The test would alert you if the the related functionality breaks.

A useful test is one that tests functionality that your users would know about if it stopped working.

The goal is for the users to never know that something broke before you do.

Over a long enough period of time, this can only be achieved by automated testing.

If a test doesn't give you confidence that a part, or an entire user flow works, it is a wasted test.

It's not only the time spent writing the test that's wasted. It's also the time spent updating it, managing the extra code that doesn't add any value, running it, etc.

The easiest and least obvious way to introduce code that adds no value into a project is with useless tests.

With the broad adoption of agile, it's much less common for companies to implement features that the client didn't ask for. However, most companies still have a requirement of X% code (test) coverage, e.g.

80%.

In short, code coverage is how much of our code is executed when the tests are run.

For example, how many of the functions in your codebase have been invoked during testing. How many of your conditional statements have run during testing, etc.

Putting emphasis on code coverage doesn't make sense because it incentivizes you to game the system and write tests that don't give you confidence.

One of the most common ways to waste time is to chase X% code coverage.

You start thinking in terms of testing functions and classes, and not in terms of testing user flows.

A user flow is the path the user takes in order to complete a task on your website.

Always aiming to get X% code coverage is a terrible idea.

Like any other code, tests also have to be maintained, changed and refactored.

"Code Coverage < Use Case Coverage" - [Kent C. Dodds](#)

My experience has been that the best tests are the ones I didn't have to write.

When using a third-party module or a service, e.g. for authentication, we aren't just taking advantage of the functionality we didn't implement.

We are also taking advantage of all of the tests someone else has written for us.

Some of the benefits of testing include: we spend less time debugging

- we have less complexity to manage we spend less time waiting
- on someone for code reviews
- we can confidently refactor our code without worrying that something might break
-

Refactoring is the process of restructuring code without changing its external behavior.

In other words, the goal of refactoring is to make your code easier to read, understand and reuse, not to change what the code actually does. If you massage untested code enough times, it'll break.

You could spend the time to manually test all user flows every time you refactor your code, or you could spend the time to write the tests.

It's quite obvious that the second option is better if you're going to spend at least a couple of days working on the project.

Not refactoring code, because of worrying that something might break is one of the most common bottlenecks you'll encounter in companies.

Some developers like writing tests, most don't. I certainly don't.

It's much more exciting to implement features that directly add value to your company, than to make sure these features actually work.

Unfortunately, the code we push to production has to be tested.
Fortunately, it doesn't have to be tested by us.

One of the things I consider when thinking about if I should use a service or write the code myself is whether I can reliably test this functionality.

Things like payment processing, time-sensitive financial transactions, sending mass emails, I would much rather outsource.

If the specific functionality is hard to test in a way that would give you confidence, it's best to use a service that is tested or exposes APIs that would allow you to test it in a more reliable way.

Certain things take less time to implement than to test. And whether I'll be able to confidently make changes to the functionality is a whole 'nother story.

The whole point of testing is to gain confidence that your application works.

If you don't feel confident when making changes to a tested codebase, you're wasting your time.

The best code is the code that works and you never have to see or manage.

If I use an authentication service from a reputable company, I can make use of any of the implemented functionality without having to write and manage tests and I'll still be confident that the functionality in my application works.

To be more precise, I'm testing my integration with the service, I'm not testing the functionality provided by the service.

The programmers of company X are responsible for testing that their software works. I am responsible for testing that my integration with their software works.

For example, let's assume that you pay to use an authentication/authorization service that provides "Simple and Secure User Sign-Up, Sign-In, and Access Control".

If you want to make sure the user registration flow works, you would test:

- a success message is rendered when the registration form is submitted correctly

Register for an account

After you fill in the form, you will receive an email with an activation link - click on the link to complete the process.

Email Address
e.g. john-smith@example.com

Password
Must be at least 6 characters

Repeat your Password
Must be at least 6 characters

First Name
e.g. John

Last Name
e.g. Smith

[Resend activation link](#) [Already have an account?](#)

Register

Please click on the activation link, sent to your email address to complete the registration.
Form submitted successfully.

- field errors are displayed when the submit button is clicked with empty values

Register for an account

After you fill in the form, you will receive an email with an activation link - click on the link to complete the process.

Email Address
e.g. john-smith@example.com ⓘ
Email is required.

Password
Must be at least 6 characters ⓘ
Password is required.

Repeat your Password
Must be at least 6 characters ⓘ
Password confirmation is required.

First Name
e.g. John ⓘ
First Name is required.

Last Name
e.g. Smith ⓘ
Last Name is required.

[Resend activation link](#) [Already have an account?](#)

Register



- an error message is rendered when the API throws (e.g. because a user with the provided email already exists)

Register for an account

After you fill in the form, you will receive an email with an activation link - click on the link to complete the process.

Email Address
bobbyhadzhiev@gmail.com

Password
.....

Repeat your Password
.....

First Name
Borislav

Last Name
Hadzhiev

[Resend activation link](#) [Already have an account?](#)

Register

✕
An account with the given email already exists.



You could also test that:

- the email the user supplied is valid the two passwords match
- the password must contain a minimum of X characters
-

This would prevent users from flooding the auth service (by using your form) with invalid data. However, even if you don't test whether the email is valid, the auth service definitely does.

If you were to implement the functionality yourself, you would have to test the validity of the email and whether the password satisfies the requirements on both your frontend (browser) and your backend (server).

You're basically testing that your frontend (what the user sees) integrates with the service correctly.

You wouldn't test:

- supplying an email that already exists throws an error
- supplying an invalid email throws an error
- supplying a password that doesn't satisfy the requirements (e.g. length) throws an error
- a user is successfully created when valid inputs are provided
- a confirmation email with an activation link is sent to the user after account creation
- clicking on the confirmation link updates the user's status to **confirmed**
- making a request with an invalid confirmation code doesn't update the user's status to **confirmed**

- making a request with a valid, but expired confirmation code doesn't update the user's status to **confirmed**

Being confident when implementing or updating functionality is crucial. The code you use has to be tested. However, it doesn't necessarily has to be tested by you.

There are services for things like:

- authentication/authorization
- sending emails newsletter
- management payments
- search functionality e-commerce
- comments content management
- systems data storage anything
- else you can think of
-
- The next chapter expands on when to outsource.
-
- [Kent C. Dodds](#) often talks about how we shouldn't test implementation details.

An implementation detail is exactly what it sounds like - a detail in how the functionality is implemented. These are details that your users don't know or care about. They only care that the user flows work.

"Implementation details are things which users of your code will not typically use, see, or even know about." - Kent C. Dodds

One way to know that you're testing implementation details is when you refactor your code, without changing its actual behavior, and you have to update your tests.

If you change the implementation, but not the behavior of your code, you shouldn't have to make any changes to your tests.

If you test user flows (e.g. user does this, expected behavior is this) instead of how the functionality is implemented, you won't have to change your tests every time your implementation of a feature changes.

A feature is a unit of functionality that satisfies a requirement.

How your functions and classes implement a specific feature isn't important. What's important is that the feature works.

You shouldn't have to change your tests if you rename class properties or state variables.

"The more your tests resemble the way your software is used, the more confidence they can give you." — Kent C. Dodds

You should test user behavior.

For example, on the frontend (browser):

- if a user clicks the submit button without filling the required fields in the contact form, an error should appear on the screen
- if a user fills in a login form with the correct credentials, they get redirected to the home page
- if a user fills in a login form with incorrect credentials, an error should appear on the screen

On the backend (server):

- if a user attempts to access a product that doesn't exist, the server returns a 404 not found response
- if a user tries to update their avatar without a valid JWT (auth) token, a 401 unauthorized response is returned
- if a user updates their address successfully, the changes are reflected in the database and a 200 success response is returned

Notice that we are testing user flows, or a part of a user flow.

A user flow is the path the user takes in order to complete a task on your website.

If a user reads some of the list items above, they wouldn't be totally lost.

Most of the time, testing the backend is simpler.

It mostly consists of making http requests with valid or invalid credentials, valid or invalid JSON data, and checking whether the actual response was the expected response.

The short version of what to test is - prioritize testing the functionality that is important to your users and your business (which should be the same, most of the time).

If you have an e-commerce store and users aren't able to add items to their shopping cart, then you don't have an e-commerce store.

Similarly, if you have an e-commerce store and your checkout flow doesn't allow users to complete their purchase, then you don't have an e-commerce store.

If the pagination or the filters on your e-commerce store break and you end up displaying less products than you actually have, then that's a problem because visitors are not going to buy what they can't see.

On the other hand, if the functionality for adding comments or rating products breaks, that's not great, but visitors could still purchase from your store.

If customers can't see their order history, that's definitely not great, but it doesn't prevent visitors from making a purchase.

If you want to read more about writing tests that give you confidence, check out [Kent's material](#).

Focus on adding value

There's nothing more important than adding value. Everything else is a means to an end.

It's not important how exactly you'll implement the feature, what stack you'll use, or if you'll just outsource it and pay for a service.

What's important is that the feature is there, it does what your customer wants and it works.

The best way to get promoted, get a salary raise and have leverage is to add value.

Why do a third of the websites on the internet use Wordpress with all of its inefficiencies? Because WordPress makes it easier for programmers to add value.

By adding value, I mean adding value to the customer of whatever it is you're selling.

For example, implementing features, fixing bugs, making changes to existing functionality, and whatever it is your client decided would improve their experience or their business.

Before you start learning something new, ask yourself two things:

1. How will I use this to add value? If you don't have a good answer, you probably shouldn't learn it.
2. Can I use this to add value without having to learn it? If the answer is yes, you probably shouldn't learn it.

If you're an athlete, winning solves everything. If you are a programmer, adding value solves everything.

In general, you're only allowed to be rude if you can add a lot of value. The more value you add, the more you can get away with. This goes to show how important it is to add value.

There are many developers (myself included) who write less than ideal software and still make a living.

Why is that? Because they add value. They have found someone to add value to. Even though their software is less than ideal, they add value in one way or another and that's what most companies focus on (making money).

This is not necessarily bad, it's just how the world works.

On a case-by-case basis, you might want to think about whether you want to focus on adding value, writing high-quality software, or both.

Ideally, you would want to add value by writing high-quality software every time, but that doesn't always make business sense.

If I need a WordPress site and know nothing about programming, I'd probably go with the developer who charges \$1,000, and not with the one who charges \$5,000. All I care about is that the site does X, Y and Z.

Most managers don't want to know about the quality of the software. They want to know if it does what they need it to do and how fast you can add new features.

Asking most managers about time for refactoring is like asking them for paid leave. The most important thing is implementing new features that (seemingly) work, as fast as possible.

It is largely up to these managers whether you'll get promoted and make more money.

Doing things that don't directly add value is kind of like playing defense on a sports team. You might do a great job, but the person who scored is going to make more money than you.

You could be great at what you do, but if you can't use what you know to add value, it's kind of pointless.

You should either join another company (find someone to add value to) where they use the same technologies, or you should focus on learning things you can use to add value.

Value is anything that makes you or your company money, directly or indirectly.

For example, you could spend 2 months trying to learn all the intricacies of the **git** CLI, or you could simply use a GUI (graphic user interface) to get the job done without knowing (almost) anything about **git**.

If you've spent months learning everything about **git**, but only use 6 commands on a day-to-day basis, you've wasted your time.

You should only spend your time learning everything about **git** if you intend to use your knowledge on a daily basis.

If you don't focus your attention on the shortest path to adding value, you risk overextending yourself and learning things you won't be able to apply.

Of course, there's no way to only learn things you can apply and to apply everything you learn.

However, learning things you won't be able to apply is a much bigger problem in programming than in any other field.

With things constantly changing, new frameworks being hyped every day, companies posting job offers that require you to know 30 technologies across the whole stack, it's sometimes hard to stick to only learning things that you can apply to add value.

One thing you should definitely learn is the fundamentals of programming and the fundamentals of the specific technologies you're using.

How you define "fundamentals" largely depends on what you're learning. A generic definition would be - transferable, universally true knowledge.

You are guaranteed to use the fundamentals, and it's highly unlikely the fundamentals are going to change.

We don't get paid to write code, we get paid to add value.

It's not necessarily the person who works the hardest, or writes the most lines of code the person who produces the most value.

You have to be good at what you do and more importantly, you have to work on the right things.

"The direction you're heading in matters more than how fast you move"
- Naval Ravikant

The hardest working people in a company don't get paid the most money.

You don't only add value by implementing features yourself. Sometimes it's much easier to add value by integrating with services. The end result is what matters.

If there is a service that does what the customer needs and it makes financial sense to use it, then use it.

Integrating with services scales much better than implementing the functionality, writing the tests and managing the code yourself.

Pick a realistic hourly rate and outsource things that cost less than your hourly rate. If there is a suitable solution out there, use it.

Over time I've learned that it takes money to make money. Even if you aren't directly paying to integrate with a service, and you implement everything yourself, you're still paying with your time.

You're paying with your time to implement the functionality, to test it, to manage it, etc. Sometimes you even have to acquire domain-specific information prior to all the other stuff.

I'd like to take this chance to tell you how much I hate learning domainspecific information.

When I get asked "Do you want to work for company X? You're going to be working on the most advanced accounting software in the world", the only response I can think of is "I would rather lose my car keys, my driver's license, my debit card, my ID, my passport and my phone".

"I'm never gonna financially recover from this" - me after losing all my stuff

Don't get me started on insurance.

I don't know if you've noticed, but I'm always trying identify things that I don't like.

When I find something I don't like, I tend to look for solutions.

This has helped me become a better programmer in the long term.

Code is overhead. The more code you have to think about and work around, the slower you move, the slower you add value, the less money you make.

You can only be good at so many things. Use third-party packages and services where it makes sense. Your time is worth a certain amount of money. If you can pay someone else less money to solve your issue, you should.

If it makes sense to outsource, it makes sense to outsource.

As a developer, you can't really scale yourself that much. You can always get better but there are limitations. You can hire others, but using a service is usually much cheaper and less time consuming.

When to outsource:

- Pick a realistic hourly rate and outsource things that cost less than your hourly rate.
- You can't be bothered doing X.
- It doesn't make business sense for you to do X.
- Doing X requires you to have domain knowledge.
- Getting X wrong could lead to legal trouble.

The goal is not to do everything yourself and do it well. That's not practical.

The goal is to feel confident about every aspect of your application.

There are services for things like:

- authentication and authorization
- sending emails newsletter
- management payments
- search functionality e-commerce
- comments content management
- systems data storage anything
- you can think of
-
- When it comes to using these services, the question really is - "Will outsourcing the task cost less than your hourly rate?"
- Assume you value your time at \$150/hour for the next example.

If a service that enables you to do X costs \$100 per month, but will take you 5 hours to implement and 1 hour per month (\$150 / month) to manage, you should probably use the service.

You can't scale your time, but you can scale your use of good services.

The goal is to be able to scale commonly occurring tasks.

For example, assume that managing functionality X would cost you \$100/month to outsource, or 1 hour (\$150) worth of work if you were to do it yourself.

If you work with clients, you could outsource functionality X in 100 projects, pay \$100 per month and charge your clients \$150 per month for a profit of \$5,000.

You would also get to do something else with your 100 hours, which you value at \$150/hour = \$15,000. You can scale this as quickly as you can find new clients.

On the other hand, if you decided to do the work yourself, you'd have to spend 100 hours (2.5 work weeks) of your time to manage the service for your 100 clients. You'd make a profit of \$15,000, but you'd spend 100 hours (\$15,000). You'd also soon be maxed out as this isn't scalable.

The only way to go far is to go fast - [Bob Hadzhiev](#)

In other words, the only way to go far is to leverage good abstractions.

Integrating with third-party software is much more scalable than writing the software yourself, testing it and managing it.

And that's assuming you don't have to spend time to learn a domain or a technology to implement and manage the code.

The less code you have to manage, the easier it is to wrap your head around it and make changes.

The less code you have to manage, the less things that can go wrong, the less things that will go wrong.

Obviously, this only applies if you scale something that works. Scaling something that doesn't work, e.g. introducing the same bugs and inefficiencies in multiple projects, is a very bad idea.

A poor programmer is a poor programmer

A mexican politician by the name of Carlos Hank González once said "A poor politician is a poor politician".

That's a great line and it applies to everything.

What's the point of being good at your job if it's not reflected in your salary?

Because if you are good at your job, it probably is reflected in someone else's salary.

The money you're generating has to go somewhere, it doesn't just disappear.

The goal of a company is to pay you as little as possible and squeeze the most value out of you. It's just business.

It's just business. No hard feelings, we can still be friends.

If you're a long-term oriented person, your goal probably is to become a better programmer in order to be able to make more money.

Your goals and the goals of the company don't necessarily align.

Sure, it would be good for them if you became a better programmer because they'd get more value out of you.

However, a company's goal is not for you to get better. They'd like you to be better, so you can get more work done in your 8 hours.

However, they don't spend time thinking "How do I make Alice a better programmer". They spend time thinking "How do I get the most value out of Alice".

It's your responsibility to spend time thinking "How do I become a better programmer while getting a fair compensation".

This is not a team sport. You have to do what's right for you. You can't pay your bills with recognition, friendships or social media followers.

One of the most commonly asked interview questions is: "What are your salary expectations?".

I've also been asked "How much do we need to pay you for you to be motivated?".

Don't get me wrong, I'm not about to start my own company and pay everyone a million bucks. That wouldn't work very well.

Solving the problem is not always possible or practical. At least, not solving the problem for everyone.

However, you could try to solve the problem for yourself.

Every company in the world wants to pay you just enough. It's just business and it's only fair for it to be only business on both sides.

I've worked for much less money than I deserved for my contributions. I quickly realized that this was a terrible idea. It demotivated me and ended up costing me a lot more than just the money I didn't make during that time.

You could be programming to make the world a better place. Even then making money doesn't hurt.

Or maybe you just love to write code and you would do it for free. Even if you had \$100 million in the bank, you'd wake up and start writing code.

Nevermind going to a tropical island, playing golf, or buying a camel, you'd just be writing code with \$100 million in your bank account.

If you want to change the world or just like writing code, I can't relate.

I'd much rather have camel money.

"Money is not going to solve all of your problems, but it's going to solve all of your money problems." - [Naval Ravikant](#)

I'm not very smart, so I'm probably not going to change the world in any way. On the other hand, there are many not very smart people who make good money.

Frameworks, languages, tools and services are all temporary, retirement is forever.

If you work for a large company and you feel like you're being taken advantage of, and you're not compensated fairly, you have 2 options:

1. Look for a different job.
2. Make it as painful as possible for them to lose you.

The first option is quite obvious and much better than the second. However, not all people are in a position to switch jobs.

There are 2 main reasons for why a company underpays developers:

1. They can find someone else to do your job for the same amount of money
2. They think they can find someone else to do your job for the same amount of money

If they can actually find a replacement, tomorrow, then you have to learn new things that would enable you to separate yourself by adding more value, or by adding value with specialized knowledge that is not as easy to acquire.

If they think they can find a replacement tomorrow, you have to show them why they're incorrect. You can do that by taking on more responsibilities, being more assertive and adding value with specialized knowledge.

If they can replace you, then they don't have to pay you a lot.

Or to be more precise, if they can replace you, they'll underpay you or they will replace you.

The only way to make a fair wage working for a company is to add value in ways others cannot. Or even if they could, they wouldn't do it for any less.

A job is a paid internship

Always, but especially when you're a salaried employee, you should take the time to understand the problem at hand and not just blindly copy and paste solutions.

You can probably solve 99% of the problems you'll encounter in your projects by blindly copy-pasting code from stackoverflow. However, this is a very short-term strategy that will end up costing you.

Being a programmer is a marathon and is all about getting better and becoming more productive. This is only possible by getting stuck and unstuck thousands of times.

Try to minimize the tasks that take time and don't make you a better programmer.

You should think of a job as a paid internship.

A job = pay + becoming a better programmer

If either of the two is not satisfactory, you should probably be looking for a new job.

If you aren't becoming a better programmer in your job (e.g. because they give you repetitive, unautomatable tasks), it has to be balanced out with a higher salary.

Time you spend doing things that don't make you better is the definition of trading time for money.

It is trading hours for dollars.

Spending time doing boring, unautomatable tasks that don't make you a better programmer is no different than doing nothing.

Sure, you'll make more money per hour, but you aren't increasing your leverage in any way.

Your employer has no reason to increase your salary. In fact, they probably feel confident that they can replace you quite easily.

Everything you do in your job, you do for 2 reasons:

1. to get better - includes gathering analytics and testing things that help you reach useful conclusions. Not just directly getting better at language or framework X.
2. to make money - if you aren't getting better, then you better be making a lot of money. Trading time for money doesn't make sense unless you're compensated well.

Stand out when applying for jobs

There are so many resources for how to break into tech.

Let me preface this by saying that I don't know what I'm talking about.

You'd be much better off consuming content from [Leon Noel](#).

I am by no means a specialist in this, but there are a couple of things I'd like to share.

When applying for a job with a company, review their website (or service)

Using constructive criticism, tell them what they're doing wrong and what they can improve (functionality, design, SEO, security, performance, conversions, whatever you specialize in).

Constructive criticism provides specific examples and actionable suggestions for making improvements.

What the review consists of largely depends on what you specialize in.

Whatever field you specialize in, try identifying issues or opportunities and tell them how you'd make improvements.

To be on the safe side, you'd want to mostly include obvious flaws (bugs, errors) in their application and ways for them to make improvements.

Opinion-based suggestions are much more tricky, because maybe they had a reason to do things a certain way that you might not know about.

In other words, more facts, less opinions.

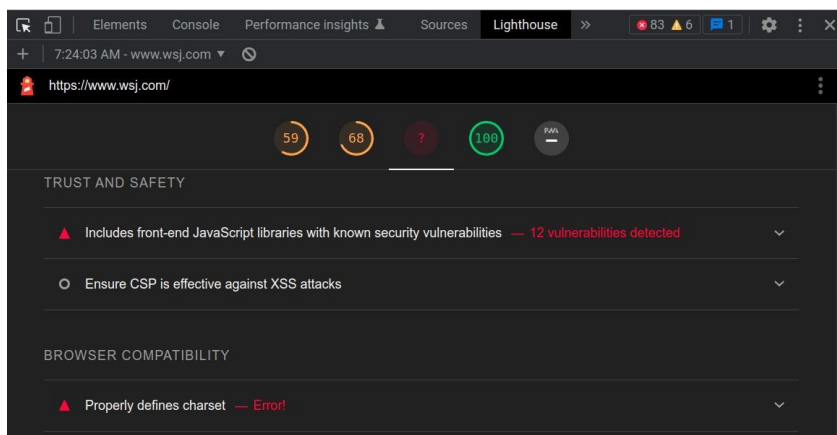
You can bring up opinion-based stuff during the interview: "Hey, I noticed you use X to do Y, is there a reason you didn't do Z?".

You also don't have to do this for every company you apply with. If their website or service works great and you have nothing valuable to add, there's no point in doing this.

If you are a penetration tester, you'd evaluate the security of the website (legally).

If you are a designer, you'd look at the website's design.

If you are a general web developer, you can run the Lighthouse tool in Chrome to get some ideas.



Whatever field you are in, you can google most common X mistakes, e.g.

"Most common web design mistakes".

You can even check what technologies their website is built with. There are sites like builtwith.com that show very detailed information about the technologies a website uses.

As a side note, something I often notice is informational websites using WordPress.

Most of the time, there is no good reason to use a database if all you have is a static page written in JavaScript, CSS and HTML, a blog and a contact form.

You can just upload the static files to a CDN (Content delivery network), and not have to worry about security vulnerabilities, database management, server patching, having to scale unexpectedly, some Denial-of-service attacks, etc.

Using a CDN is cheaper and less time consuming (assuming you do things right).

Usually, the less things that can go wrong, the less things that will go wrong.

This is something I'd ask about during a job interview.

Not every applicant is going to tell a company about a problem or an opportunity they didn't know they had.

If you provide constructive criticism and ways for them to improve their business when applying for a job, they'd assume that you'll keep doing the same after you join the company.

You don't have to be a 10x programmer to find things that can be improved.

For example, if you're a designer, you can google "most common web design mistakes", and chances are you won't have to go to the second page of google to find some mistakes the company you're applying to makes.

If you can't find a great way to apply this when looking for a job, you should still adopt the mindset of constantly looking for inefficiencies and opportunities.

This is how people usually come up with business ideas. If you can identify a real problem, you can start your own business.

You can also ask a friend programmer to help you with the review.

Once you know what to look for, you will be able to define a step-by-step process that you can follow.

You should keep the same mindset after joining a company.

- Always look for inefficiencies.
- Always look for things that don't feel right.
- Always look for things that prevent the company from making progress.

Depending on what you specialize in (e.g. design, general web dev, testing), you'll want to define a streamlined process that you can follow when doing audits.

This doesn't have to be something super formal. You're not going to invoice them for your work.

If you notice that X is broken, e.g. their mobile menu, simply inform them, send them a screenshot or a **gif** and tell them how you'd go about fixing it.

After a while, you should start noticing patterns of some of the common mistakes companies make with their websites or services in your field of expertise.

If you get good at this, you may even offer companies to do consulting for specific issues.

Solving the same problem in different companies sounds like a good business to be in.

If it's boring and makes money, you're probably doing something right.

Programming is about solving problems. It's quite impressive if you can solve problems from the get-go.

Start a blog

The best way to get a remote job is to start a blog and write about the technologies you're interested to work with.

This will be much easier if you work with niche technologies, e.g. AWS.

If you establish yourself as an authority on the topic, you'll likely get more job offers than you know what to do with.

When that happens, hire \$20 an hour developers on Upwork, tell them to pretend they're you and pocket the difference.

This is the best way to break into HR.

No one expects you to do this.

"What's common produces only common results" - [Ray Dalio](#)

By the way, I'm just kidding. Or maybe I'm half-kidding, because that's kind of what HR (Human resources) actually do. They try to get competent developers to work for as little money as possible to benefit themselves and the company.

Most companies have a predefined budget for a position. You never really know how much of this budget goes towards paying you.

The person who is tasked to make you a job offer could also be a fellow developer. Either way, they represent themselves and the company, you represent yourself.

To get back to the topic of creating your own blog, something interesting to note is that [my blog](#) gets many more views on topics such as JavaScript and Python, but I mostly get AWS job offers. AWS-related topics make up a fraction of my traffic.

I wrote a bunch of articles about an AWS service called CDK. These articles bring in very few views every month, but I have probably been offered CDK work from more than a hundred different companies.

Obviously, that doesn't mean that you should write articles about the most niche technologies you can find.

Write articles only about technologies that:

- you are interested to work with are in
- demand

It would be more difficult to separate yourself from the pack if you write articles about mainstream languages (e.g. JavaScript and Python).

However, this still shows that you're learning in your free time.

If I'm looking to hire a developer, I'll be much more inclined to hire someone:

- with a personal blog who tweets
- about programming

If you care about a technology, it shows in the articles you write.

Show that you are invested

If I'm an employer, I want to know that you're invested.

I want to know that programming is not just a hobby for you. It's not something you forget about after you finish work.

Whether that's accurate or not, you should try to convince every company you apply to that you love writing code in your day job and in your free time.

They should think that you're invested and you'll likely be thinking of solutions to problems in your free time.

It's not about this company or any other company. You'd be doing this in every company because that's what you do.

The word "invested" is very important here. If I'm looking to invest in your business, I want to know that you have invested in the business yourself.

I want to know that you have something to lose if things go wrong. The more you have to lose, the better. The bigger your investment, the better.

Your investment in this scenario is your time. It's the time you'll spend getting better at programming that I don't have to pay you for.

The more time you spend getting better at programming without me having to pay you, the better.

Here are a couple of ways to show companies that you are invested:

- working on side projects tweeting about programming,
- posting on linkedin having a blog or a youtube channel
- expressing opinions rather than just facts and tips (shows them you think about this stuff)
- going to conferences or giving talks
- buying a keyboard without a number pad
- leaving a comment under every programming video you watch. It doesn't have to be anything meaningful, an emoji is fine
- only wearing clothes (e.g. t-shirts and hoodies) with logos of programming languages and frameworks

- putting a lot of stickers on your laptop. You don't have to use the technologies, you just need the stickers
- contributing to GitHub every day. You can use a bot for this. Tell everyone when you get to 1 year
- saying that you are a "Google developer expert", "AWS Hero" or "Microsoft Technology Associate". Chances are they won't sue. Make sure to front-load the title with the company name as the first word. The next words largely don't matter

Personally, I'd go with "Netflix Certified Solutions Architect". It's quite accurate and has just the perfect length.

OK, maybe I got a little carried away towards the end, but you get the point.

You don't necessarily have to explicitly state that you write code all day every day. You can just give them the impression that you do.

Even if you aren't as good as some of the other applicants, your level of investment might be what gets you the job.

It should be noted that being invested doesn't mean being desperate. You are invested in yourself, in becoming a better programmer. You aren't invested in working for company X (desperate).

It's not about the particular company, it's about your love for programming. You'd do this with \$100 million in the bank. This is your identity.

Being stuck on a problem keeps you up at night. You're constantly looking to solve problems and improve solutions to already-solved problems.

Being invested in yourself and being confident is the opposite of being desperate.

Companies will think twice about lowballing you in contract negotiations if you're invested in yourself and confident.

Maybe I should test this by doing a split test with 2 book titles:

- All in programming
- Half-assed programming

Many moons ago, when I was a university student, I used to watch football (soccer if you're an American).

I watched a football match and the local team lost. Then I went to the nightclub the same day and the entire team was there.

They were dancing, drinking overpriced, diluted alcohol and listening to terrible music (unfortunately, I was too).

I thought to myself, "Wait, they just lost the game and went partying the same night? That doesn't make any sense". It felt like I was more invested in the outcome of the game than the players themselves.

I thought "They lost, but they've made tens of thousands of dollars and can just go partying the same night, while I've wasted 2 hours of my time and now I'm bummed out for the next 2 hours. How does that make sense?".

This was the last football game I ever watched.

Let's be honest, if I'm gonna be bummed out that they lost, I want them to also be bummed out. I'd much rather watch an MMA fight because if I'm rooting for a fighter and they win, great, I get a dopamine hit.

If I'm rooting for a fighter and they lose, that's much easier for me to digest. I know they've done everything in their power to prevent it from happening.

If you lose an MMA fight, you get beat up in front of millions of people and lose half your paycheck. Stakes are high.

Every employer wants you to be invested. They want you to be invested with your time and your identity.

If you show them that you are investing your free time to become a better programmer, that you take pride in writing high-quality software and have a bit of an ego, then they know that you'll do what needs to be done for them to get their money's worth and then some.

By the way, I have to confess that I lied about something.

I lied about it taking me 1 year to write this book and it bothers me.



I wrote the majority of this book in the past month.

I guess I lied to give people the impression that I've put a lot more effort into this.

I guess I lied because I think the book is good and it only needs to be given a chance.

There's nothing more humbling than your expectations diverging from reality.

It was either that, or to have something to write about.

In the end, it's their decision to make you a job offer or not. If it makes sense for them to offer you the job, they'll offer you the job.

The cards are stacked against you throughout the entire process.

Optimize every action you repeat hundreds of times a day

You should spend a week and think about every little action you repeat hundreds of times a day.

Spend time to optimize and automate repeatable, predictable, boring tasks that take time and don't make you a better programmer.

I don't think of things in terms of: "I did X 100 times today" and it took me a total of 1 minute.

I think of things in terms of "I'll do X 36,500 times this year and it'll take a total of 365 minutes (~6 hours).

Snippets

I use snippets and keyboard shortcuts for 2 main reasons:

1. to save myself time
2. to feel as comfortable as possible when writing code. Everything has to be as effortless as possible.

I don't want to have to spend time thinking about a movement, a combination of keys or anything other than the business logic I'm writing.

It's very easy to come up with bad snippets or to create bad keyboard shortcuts. The goal is to be efficient, not to be fancy.

It's hard to optimize if you're a generalist and use different technologies (e.g. code snippets for 3 languages and frameworks), but there are almost always some easy wins in general categories.

Snippets aren't just used to save you a couple of keystrokes. I also use snippets to not have to remember the syntax for methods, classes, boilerplate and other initialization code from third-party libraries.

This keeps me in a flow state as I don't have to constantly switch between my IDE and the browser.

You can use snippets for everything that you don't want to have to remember.

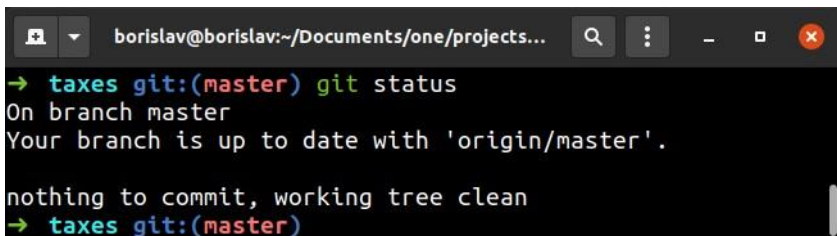
It's the same with shell aliases. A shell alias is a snippet, but in the terminal.

I use aliases not only to shorten commands, but also to make them easier to remember.

This also enables me to follow the same conventions when using different CLIs (command line interfaces).

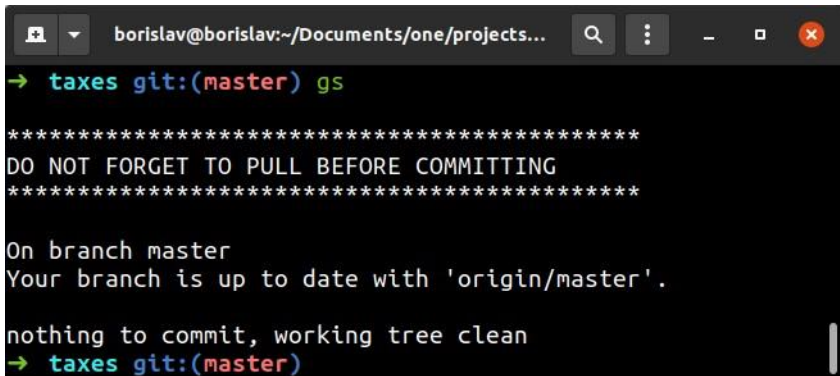
Here are some of [my shell aliases for tools you probably use](#).

Here is an example of an alias for the command `git status`. This is the long, non-aliased version.

A terminal window with a dark background. The title bar shows the user 'borislav@borislav' and the path '~/Documents/one/projects...'. The prompt is '→ taxes git:(master)'. The command 'git status' has been executed, resulting in the output: 'On branch master', 'Your branch is up to date with 'origin/master'.', and 'nothing to commit, working tree clean'. The prompt '→ taxes git:(master)' is shown again at the bottom.

```
borislav@borislav:~/Documents/one/projects...  
→ taxes git:(master) git status  
On branch master  
Your branch is up to date with 'origin/master'.  
  
nothing to commit, working tree clean  
→ taxes git:(master)
```

And this is the alias.

A terminal window with a dark background. The title bar shows the user 'borislav@borislav' and the path '~/Documents/one/projects...'. The prompt is '→ taxes git:(master) gs'. The output is a multi-line message: a line of asterisks, 'DO NOT FORGET TO PULL BEFORE COMMITTING', another line of asterisks, 'On branch master', 'Your branch is up to date with 'origin/master'.', 'nothing to commit, working tree clean', and a final prompt '→ taxes git:(master)'.

```
borislav@borislav:~/Documents/one/projects...
→ taxes git:(master) gs
*****
DO NOT FORGET TO PULL BEFORE COMMITTING
*****
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
→ taxes git:(master)
```

Notice that the alias prints a message before issuing the command.

The more I can simplify things for myself, the better.

If I've spent the time to come up with a convention for something, or I've spent the time to learn something, I want to use it everywhere I can.

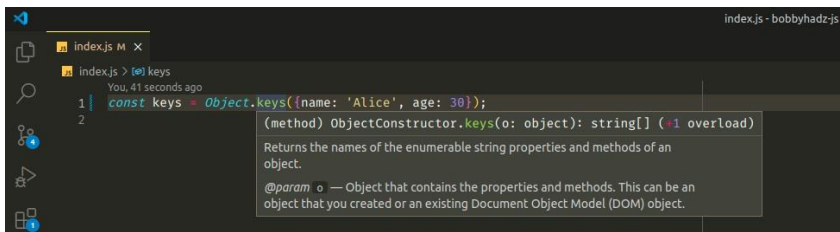
For example, I use VIM in my IDE (Visual Studio Code), in my terminal (zsh) and in my browser (Google Chrome).

Linters, Code formatters, Typed Languages

Simple and obvious stuff like a code formatter and a linter can save you tens or hundreds of hours over a long enough period of time.

Another quite obvious thing is to watch youtube videos and listen to podcasts in 1.5-2.0 speed.

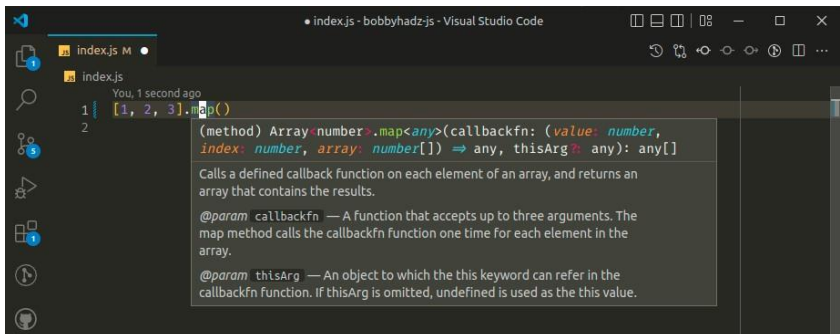
Using linters and type checkers, or even IDE default typings helps me solve many issues directly in my IDE without having to switch to my browser.



Even if you don't use a typed language (e.g. TypeScript), you can still take advantage of your IDE's built-in types for languages like JavaScript and Python.

If you haven't yet, you should spend an hour or two to hover over builtin methods and to get comfortable reading types.

Most of the time, you shouldn't have to google method names. The information you need to be able to use the method is most likely available in your IDE.



When you hover over a built-in method, you often get information about its parameters, its return type and a short description.

Don't learn the intricacies of method X (parameters, order, default values, return type, edge cases).

Instead, hover over the method in your IDE to get information about its parameters, its return type and a short description.

Use the documentation when you need to handle edge cases, e.g. method X does this when a negative value is provided for its third argument.

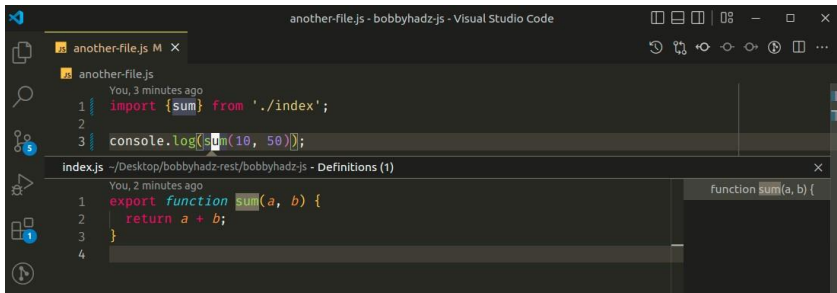
Typed languages like TypeScript:

- help me fix bugs that are in the making give me
- confidence when writing code eliminate the
- need for writing some tests

The less context switching I have to do, the more I can keep my focus on writing business logic.

Another thing that helps me stay focused is to use the "Peek definition" functionality most modern IDEs have.

In VSCode I have it bound to **Ctrl + Shift + F10**.



It enables me to look at a function's implementation without having to open the file where it is defined.

Keep track of things

When I first started programming, I had a simple `.md` file (could also be `.txt` or notion) that I used to update every day, answering the following 3 questions:

- What did I accomplish today?
- What do I plan to work on tomorrow?
- What (if anything) is preventing me from making progress?

This helped me identify patterns and saved me a couple of minutes every morning, because I already knew what I had to do that day.

Unfortunately, I've gotten lazy and I don't do it anymore, but I think there is some value in doing it.

The third question is quite important - "What (if anything) is preventing me from making progress?". I try to always be mindful about things that don't feel right when working on a project.

Most of the time, when something doesn't feel right, I'm doing something wrong.

Right now, my process consists of having a single `.txt` file that contains notes about everything I don't want to forget. Not very organized, I know.

Keyboard shortcuts and navigation

Let's start with a couple of obvious things:

- there's no reason for you to have a number pad on your keyboard. It's just an extra 2 inches you have to move your hand to reach your mouse.

2 inches doesn't sound like a lot, but you'll move your right hand an extra

2 inches hundreds of thousands of times for no reason. • previous/next page buttons on a mouse are quite useful.

There is a shortcut or keyboard combination for everything you can think of.

Modern IDEs are fairly flexible. If you can think of something and define a query, you'll find a solution quite quickly.

The more comfortable you are when writing code, the more time you'll spend in a flow state.

What works for me might not work for you, but I've seen massive return on investment from learning VIM. Some of the benefits include:

- not moving my right hand over to the arrow keys a million times a day
- I can use VIM in my IDE (Visual Studio Code), in my terminal (zsh) and in my browser (Google Chrome)
- the default keyboard shortcuts cover almost everything I need to do

I'll post a list of the keyboard shortcuts (not VIM-specific) I most often use.

This is just meant to give you ideas. What works for me might not work for you.

Here are some things to note though:

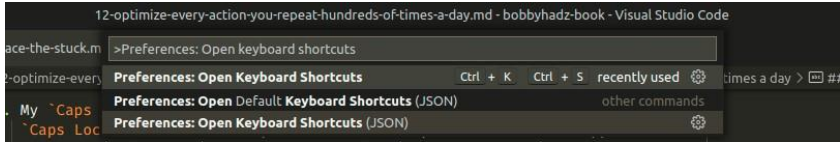
1. My **Caps Lock** key is set to **Ctrl**, so I don't click the actual **Ctrl** key. **Caps Lock** is much better positioned and I optimize for what happens 99.9% of the time (using Ctrl), not 0.1% of the time (using Caps Lock).

When you see **Ctrl** in a keyboard shortcut, I really meant **Caps lock**.

2. I use VSCode for my code editor, but that shouldn't matter as you can customize your keyboard shortcuts in any IDE.
3. Keyboard shortcuts are very opinion-based. What works for me might not work for you.
4. I use VSCode VIM extension (but you don't have to).

5. The text to the right of each keyboard shortcut is the name of the command in VSCode.

If you use VSCode, you can open your keyboard shortcuts with **Ctrl + Shift + P**, and typing in **Preferences: Open keyboard shortcuts**.



You can filter by the command names (the text to the right of the key combinations) if you decide to change some your VSCode keyboard shortcuts.



Some of my most commonly used keyboard shortcuts

I don't have a great way to add **gifs** to a **pdf**, so I've added a **gif** for each keyboard shortcut to an [article on my website](#).

It's best to check out the [article](#) and scroll past the list items.

1. **CTRL + Shift + 0** - View: Move Editor into Next Group
2. **CTRL + Shift + 9** - View: Move Editor into Previous Group
3. **CTRL + Shift + J** - Add Cursor Below
4. **CTRL + Shift + K** - Add Cursor Above

5. **Alt + D** - Add Selection To Next Find Match
6. **Ctrl + F2** - Change All Occurrences
7. **Alt + W** - Close Window
8. **Ctrl + Shift + ** - Collapse Folders in Explorer
9. **Ctrl + Shift + D** - Copy Line Down
10. **Ctrl + Shift + U** - Copy Line Up
11. **Ctrl + '** - Expand Selection
12. **Ctrl + ;** - Shrink Selection
13. **Ctrl + N** - File: New file
14. **Ctrl + Shift + N** - File: New folder
15. **CTRL + L** - focus left window (extension.vim_navigateCtrlL)
16. **CTRL + H** - focus right window (extension.vim_ctrl+h)
17. **CTRL + J** - Terminal: Focus Terminal
18. **CTRL + K** - View: Focus Active Editor Group
19. **CTRL + S** - File: Save
20. **CTRL + Shift + S** - Save All
21. **CTRL + Shift + [** - Fold
22. **CTRL + Shift +]** - Unfold
23. **CTRL + Shift + -** - Fold All
24. **CTRL + Shift + =** - Unfold All

25. **F12** - Go to Definition
26. **Ctrl + P** - Go to File
27. **Ctrl + Shift + Enter** - Insert Line Above
28. **Ctrl + Shift + Enter** - Insert Line Below
29. **Alt + J** - Move Line Down
30. **Alt + K** - Move Line Up
31. **Ctrl + Shift + F10** - Peek definition
32. **Ctrl + .** - Quick Fix
33. **Ctrl + Space** - Trigger Suggest
34. **F2** - Rename Symbol
35. **Ctrl + Shift + H** - Search: Replace in Files
36. **Ctrl + Shift + P** - Show all Commands
37. **Ctrl + F1** - Show Hover
38. **Ctrl + Shift + E** - View: Show Explorer
39. **Ctrl + B** - View: Toggle Primary Side Bar Visibility 40. **Ctrl + Tab** -
View: Open Next Editor
41. **Ctrl + Shift + Tab** - View: Open Previous Editor
42. **Ctrl + Shift + T** - View: Reopen Closed Editor 43. **Ctrl +**
Shift + F - View: Show search
44. **Alt + 1, Alt + 2, Alt + 3** - Open Editor at index

45. **Ctrl + 1**, **Ctrl + 2**, **Ctrl + 3** - Focus Nth Editor Group

Working with tags (Emmet)

1. **Ctrl + 5** - Emmet: Expand Abbreviation
2. **Alt + E** - Emmet: Go to Matching Pair
3. **Alt + S** - Emmet: Split/Join tag
4. **Alt + U** - Emmet: Update Tag
5. **Alt + A** - Emmet: Wrap with abbreviation
6. **Alt + R** - Emmet: Remove Tag

The less you have to move your hands, the better. Commonly performed actions have to be effortless.

Think long-term and optimize things that you have to repeat often (tens or hundreds of times a day).

There is a browser extension called [Vimium](#) which is quite convenient for navigating in your browser.

Here is a [2 minute youtube video](#) of how it works.

Note that the **Vimium** extension sets a bunch of keyboard shortcuts in your browser. Therefore, it might override website-specific keybindings (e.g. keybindings specific to [github.com](#)).

Search for code examples on GitHub

Searching for code examples on GitHub is quite useful when the documentation for a library you're using isn't very helpful.

Most of the time, I don't want to have to read the source code to be able to use a class or a function.

A code snippet is worth a thousand words. Chances are someone has already figured out how to use a specific function or class, and it's in a bunch of open source repositories on GitHub.

When you can't find what you're looking for in the docs, you can either google or search for code snippets on GitHub.

Google is not as good as Github at finding specific code snippets (especially when it comes to more niche third-party libraries).

You can find anything on Github - code snippets, passwords, API keys, literally everything.

The next time you get rate limited by an API, just look for a key on Github. There are many API keys that have been open-sourced.

When using unfamiliar libraries, the fastest way to get something done is to search for code snippets on Github.

Sometimes I get asked questions like "You've shown how to do X, but I need to do a variation of X".

If I had to take a guess, probably around 20% of the library-specific questions I get asked could get solved with a GitHub query.

Disclaimer: There is never a guarantee that the code you'll find in a random GitHub repository works. You still have to read it and make sure it suits your use case.

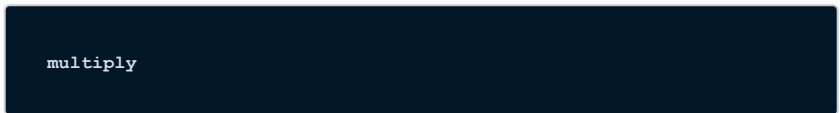
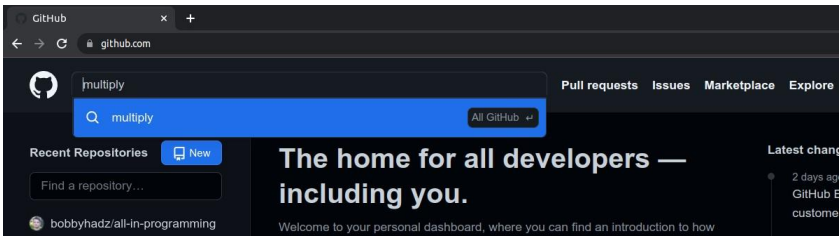
You can search for code snippets in a specific GitHub repository or in millions of public repositories.

Here are some things to note when [searching for code on GitHub](#).

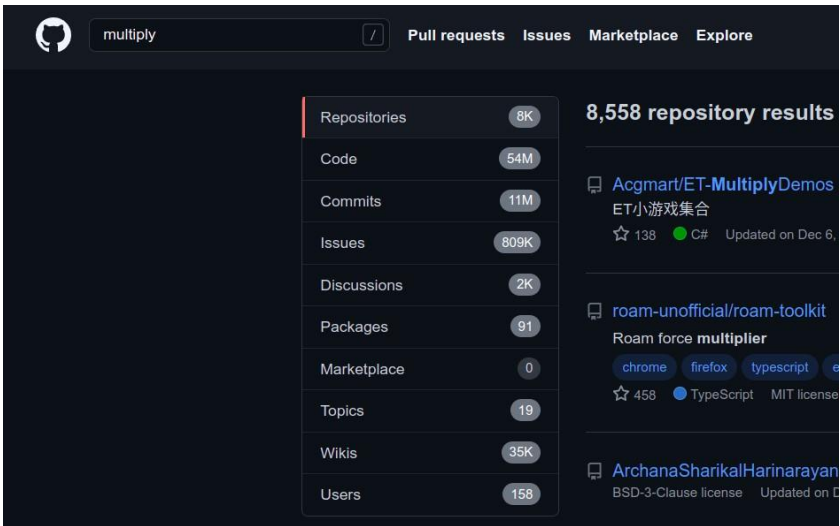
- Search is not case-sensitive.
- Only the default branch is indexed for code search.
- Only files smaller than 384 KB are searchable.
- Only repositories with fewer than 500,000 files are searchable.
- Only repositories that have had activity or have been returned in search results in the last year are searchable.
- You can't use the following wildcard characters as part of your search query: . , ; / \ ' " ` = * ! ? # \$ & + ^ | ~ < > () { }] [@. The search will simply ignore these symbols.
- Search results can show at most two fragments from the same file, but there may be more results within the file.

You can search globally or scope your query to a particular user, organization or repository.

To search globally, type your query in the search field and select "All of GitHub".



Chances are you won't get a very helpful result with a global query. You can look at some of the available filters in the left sidebar.



The filters at the top of the left sidebar allow you to scope the search to repositories with the specified name, code samples with the specified query, commits, users, etc.

The filters at the bottom of the sidebar allow you to scope the search to a specific programming language.

The screenshot shows the GitHub search interface. On the left, a 'Languages' filter is expanded, showing a list of programming languages and their corresponding number of results. On the right, a list of search results is displayed, each with a repository icon, the repository name, a brief description, and star/like counts.

Languages	
Java	1,122
JavaScript	1,024
Python	971
C++	849
C	579
HTML	393
C#	358
Verilog	300
Swift	249
VHDL	185

Repository	Description	Stars	Language	Updated
alecthomas/units	Helpful unit multipliers and functions for Go	88	Go	Updated on Dec 18, 2021
oneApple/MultiThreadServer	线程池+epoll服务器	17	C++	Updated on Aug 29, 2013
marufaytekin/MatrixMultiply	Sparse matrix multiplication for hadoop	13	Java	Updated on Apr 12
hcsp/multiply-divide-with-bit-operation	Java basic practice for beginners: calculation	1	Java	Updated on Jun 19

Advanced search Cheat sheet

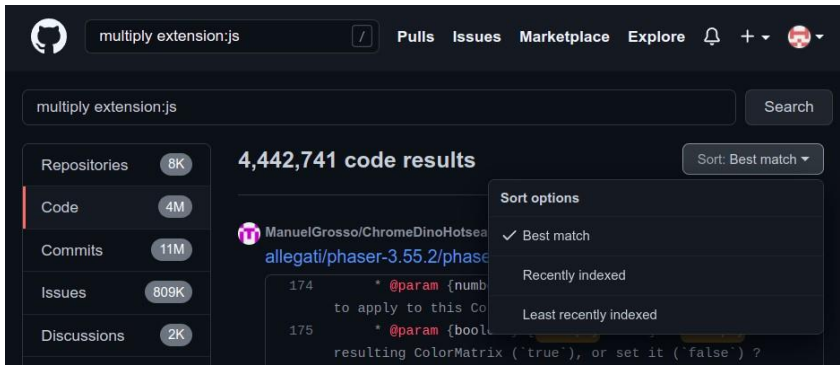
What you'll most commonly do is select **Code** in the top filter and select your preferred programming language in the bottom filter.

The following query looks for mentions of the word **multiply** in files with a **.js** extension.

```
multiply extension:js
```

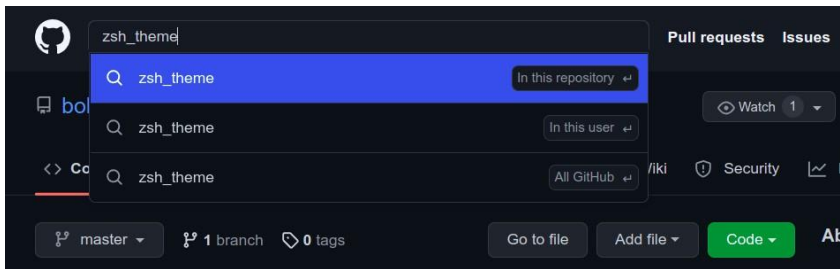
By default, the results are sorted by "Best match" and this is what you'll most often use.

You can also set the sort order to "Recently indexed" to get the most recently updated files at the top.



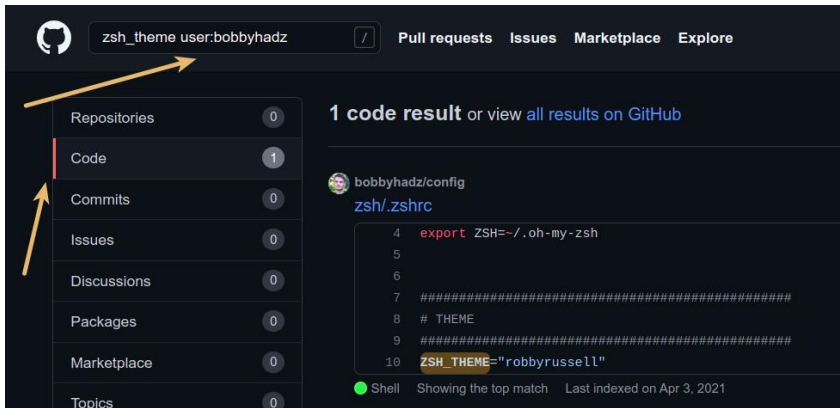
To search scoped to a specific repository, open the repository, e.g.

[bobbyhadz/config](#), type a query in the search bar, and select "In this repository".



You can also scope your query to a specific user:

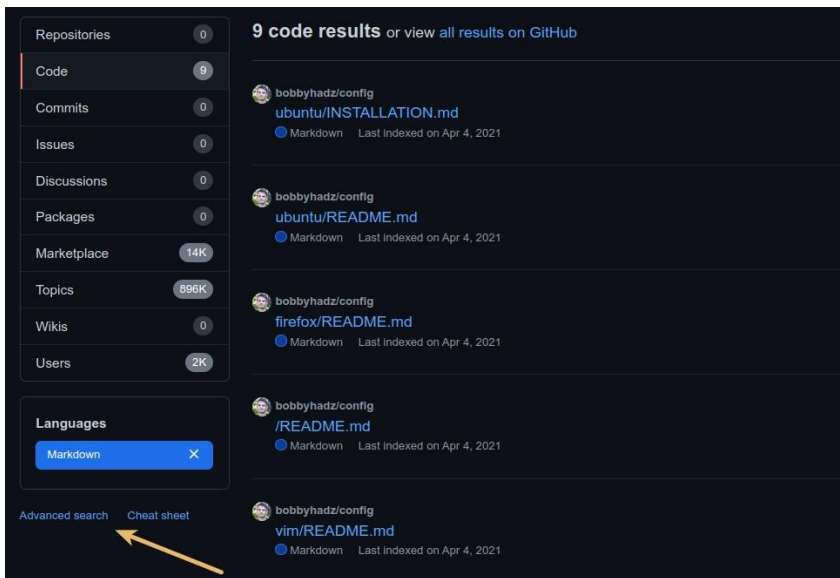
```
zsh_theme user:bobbyhadz
```

Make sure to select the correct filter in the left sidebar (e.g. **Code** or **Repositories**).

If you don't want to have to remember the syntax, use the "Advanced Search" GUI (Graphic user interface).

Click on the **Advanced Search** link below the filters in the left sidebar.



And fill in only the necessary fields.

Advanced search

language:Markdown language:Markdown

Search

Advanced options

From these owners

github, atom, electron, octokit

In these repositories

twbs/bootstrap, rails/rails

Created on the dates

>YYYY-MM-DD, YYYY-MM-DD

Written in this language

Markdown

Repositories options

With this many stars

0..100, 200, >1000

With this many forks

50..100, 200, <5

Of this size

Repository size in KB

Pushed to

<YYYY-MM-DD

With this license

Any license

Return repositories

not

including forks.

Let's look at some examples.

Search for the term **CreateBucketCommand**, click on **Code**, and set the language to **JavaScript**.

```
CreateBucketCommand language:javascript
```

The screenshot shows a code search interface with the following components:

- Search Bar:** Contains the text "CreateBucketCommand language:javascript" and a "Search" button.
- Left Sidebar:**
 - Repositories:** A list of repository types with counts: Code (257), Commits (1), Issues (1), Discussions (1), Packages (0), Marketplace (0), Topics (0), Wikis (0), and Users (0).
 - Languages:** A list of programming languages with counts: Markdown (27), C++ (25), HTML (3), JavaScript (selected with a blue highlight and an 'X' icon), Scala (10), and PHP (352).
- Main Content Area:**
 - Header:** "257 code results" and a "Sort: Best match" dropdown.
 - Result 1:**
 - Repository: `cloudcastapp/aws-sdk-examples`
 - File: `src/node/s3/createBucket.js`
 - Code Snippet:


```
1 import { S3Client, CreateBucketCommand, waitUntilBucketExists } from "@aws-
2 sdk/client-s3";
3 const s3Client = new S3Client({
4   // profile: 'your-profile',
5 });
6 // Search for CreateBucketCommandInput to see input
7 // to pass to CreateBucketCommand
```
 - Metadata: "JavaScript Showing the top two matches Last indexed on Jul 6, 2021"
 - Result 2:**
 - Repository: `bobkoby51/intro-aws`
 - File: `13_training/01_storage-s3/src/index.js`
 - Code Snippet:


```
1 import { S3Client, PutObjectCommand, CreateBucketCommand } from "@aws-
2 sdk/client-s3";
3 const params = {
4   // ...
5 };
6
7 const s3Client = new S3Client({ region: params.Region });
8 try {
9   const data = await s3Client.send(new CreateBucketCommand({ Bucket:
10     params.Bucket }));
```
 - Metadata: "JavaScript Showing the top two matches Last indexed on Mar 8"

Note that the listing shows a maximum of 2 results in each file, even though the file may contain more than 2 matches.

The command searches for mentions of the word `CreateBucketCommand` in JavaScript files.

In general, it's easier and more intuitive to use the `extension` search qualifier, e.g. `extension:js`.

Here is a [list of all of the available languages](#). Number 9 might shock you.

Definitely don't google "[language 9 name] urban dictionary".

You wouldn't believe which company developed this programming language.

The following example uses the `extension` qualifier to search for files ending in `.js`.

```
CreateBucketCommand extension:js
```

Repositories

0

Code

244

Commits

1

Issues

1

Discussions

1

Packages

0

Marketplace

0

Topics

0

Wikis

0

Users

0

Languages

JavaScript

CreateBucketCommand extension.js

Pulls

Issues

Marketplace

Explore

Sort: Best match

244 code results

cloudcastsapp/aws-sdk-examples

src/node/s3/createBucket.js

```
1 import { S3Client, CreateBucketCommand, waitUntilBucketExists } from "@aws-  
sdk/client-s3";  
2  
3 const s3Client = new S3Client({  
4  
5   // profile: 'your-profile',  
6 });  
7  
8  
9 // Search for CreateBucketCommandInput to see input  
10 // to pass to CreateBucketCommand
```

JavaScript

Showing the top two matches

Last indexed on Jul 6, 2021

bobboby5/Intro-aws

13_training/01_storage-s3/src/index.js

```
1 import { S3Client, PutObjectCommand, CreateBucketCommand } from "@aws-  
sdk/client-s3";  
2  
3 const params = {  
4  
5  
6  
7  
8  
9  
10  
11   const s3Client = new S3Client({ region: params.Region });  
12   try {  
13     const data = await s3Client.send(new CreateBucketCommand({ Bucket:  
    params.Bucket }));
```

JavaScript

Showing the top two matches

Last indexed on Mar 8

Note that the extension is whatever is after the last dot. For example, you can't filter for files with a `test.js` extension, but you can filter for files with a `js` extension.

Important: use quotation marks for queries with whitespace (multiword queries).

```
"new CfnAuthorizer"
```

The screenshot shows the GitHub search interface with the query "new CfnAuthorizer". The search results are filtered by the "Code" tab, showing 135 results. The left sidebar shows the "Languages" section with TypeScript selected. The search results list three repositories:

- cpmech/cloud**: `az-cdk/src/constructs/AuthorizerConstruct.ts`. The code snippet shows:


```
17 const cognitoArn = `arn:aws:cognito-
18 idp:${region}:${account}:userpool/${props.cognitoUserPoolId}`;
19 this.authorizer = new CfnAuthorizer(this, 'Authorizer', {
```
- chessdbal/Hercules**: `packages/infrastructure/lib/api/api-authorizer.ts`. The code snippet shows:


```
15 super(scope, id);
16
17 const cfnAuthorizer = new CfnAuthorizer(this, 'Authorizer', {
18   name: 'CognitoAuthorizer',
```
- tothalex/aws-rust-stack**: `src/api.ts`. The code snippet shows:


```
44 const apiAuthorizer = new CfnAuthorizer(
45   props.scope,
46   `${props.name}-authorizer`,
47   {
48     restApiId: apiGateway.restApiId,
```

Even then, the [GitHub docs](#) state:

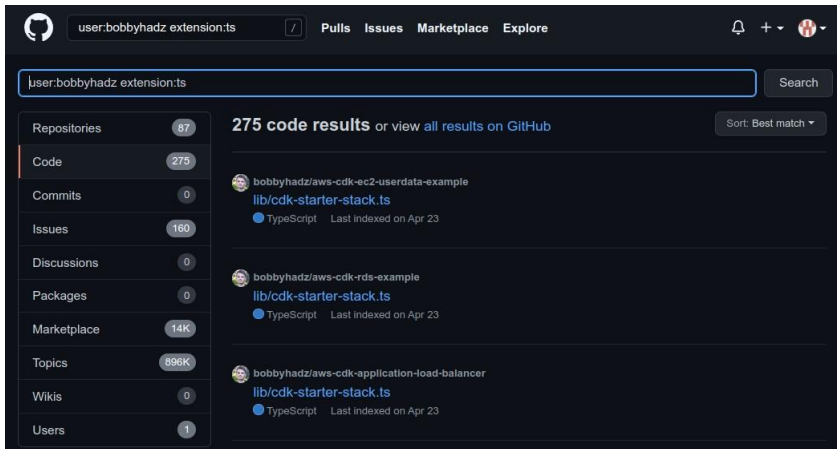
Some non-alphanumeric symbols, such as spaces, are dropped from code search queries within quotation marks, so results can be unexpected.

To search the code in all of a user's / organization's repositories, use the **user** or **org** qualifier. To search the code in a specific repository, use the **repo** qualifier.

You can use the **user** or **org** qualifiers to search the code in all repositories of a specific user or organization.

Here is an example that matches code from **bobbyhadz** in files that end in **.ts**.

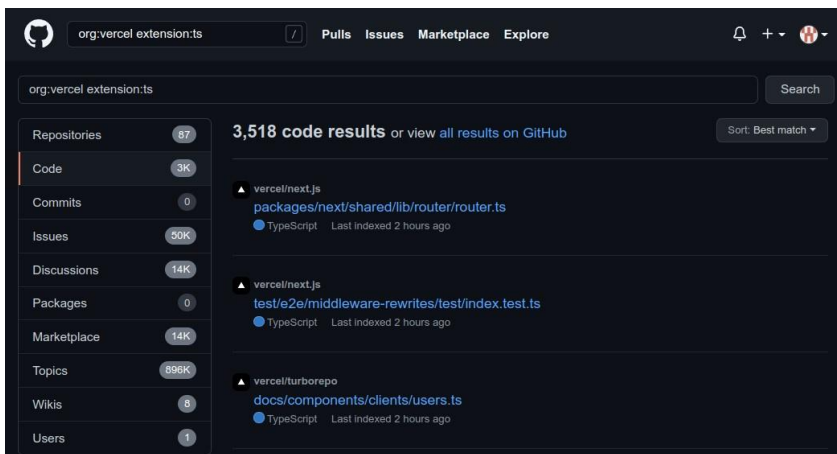
```
user:bobbyhadz extension:ts
```



The screenshot shows the GitHub search interface with the query 'user:bobbyhadz extension:ts'. The left sidebar displays filters: Repositories (57), Code (275), Commits (0), Issues (160), Discussions (0), Packages (0), Marketplace (14K), Topics (896K), Wikis (0), and Users (1). The main area shows '275 code results' with a 'Sort: Best match' dropdown. Three results are visible, all from the 'bobbyhadz/aws-cdk-ec2-userdata-example' repository, showing 'lib/cdk-starter-stack.ts' as a TypeScript file last indexed on Apr 23.

This matches code from the **Vercel** organization that ends in **.ts**.

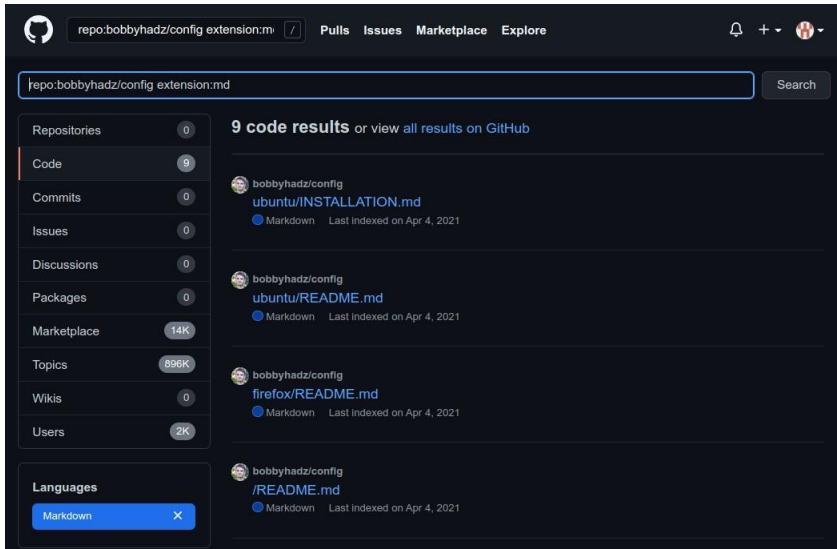
```
org:vercel extension:ts
```



The screenshot shows the GitHub search interface with the query 'org:vercel extension:ts'. The left sidebar displays filters: Repositories (87), Code (3K), Commits (0), Issues (50K), Discussions (14K), Packages (0), Marketplace (14K), Topics (896K), Wikis (8), and Users (1). The main area shows '3,518 code results' with a 'Sort: Best match' dropdown. Three results are visible, all from the 'vercel/next.js' repository, showing various TypeScript files last indexed 2 hours ago: 'packages/next/shared/lib/router/router.ts', 'test/e2e/middleware-rewrites/test/index.test.ts', and 'docs/components/clients/users.ts'.

You can also match files scoped to a specific repository.

```
repo:bobbyhadz/config extension:md
```



Here is an example that matches files named `codebuild.yml` with the word `pre_build`.

```
filename:codebuild.yml pre_build
```

The screenshot shows the GitHub search interface with the query 'filename:codebuild.yml pre_build' in the search bar. The left sidebar shows filters for Repositories (0), Code (60), Commits (3M), Issues (0), Discussions (8), Packages (0), Marketplace (0), Topics (0), Wikis (67), and Users (0). The 'Languages' section has 'YAML' selected. The main area displays '60 code results' sorted by 'Best match'. Two results are shown:

- Result 1:** shaikmohammed12/OneDineTest /aws_codebuild.yml. The code snippet shows a `pre_build` phase with a command to echo 'Nothing to do in the pre_build phase...'.
- Result 2:** mmaall/photo-deduplicator test/cicd/codebuild.yml. The code snippet shows a `pre_build` phase with commands to echo 'Command phase finished', 'Entered the pre_build phase...', and 'Pre-build phase finished'.

Both results are marked as 'YAML' and show 'Showing the top two matches'.

GitHub search also gives you the ability to exclude keywords when filtering.

Here's how you'd look for files with the word `pre_build` that don't contain the word `post_build`.

```
filename:codebuild.yml pre_build NOT post_build
```


filename:codebuild.yml pre_build NOT post_build

11 code results

Sort: Best match

juverpc/testjuver
/codebuild.yml

```
1 version: 0.2
2
3
4 phases:
5   pre_build:
6     commands:
7       - echo $CODEBUILD_SOURCE_VERSION
8       - echo $MODEL
9       - echo $PIPELINE_NAME
```

YAML Showing the top match Last indexed on Jun 16, 2021

nhsconnect/prm-infra
pipeline_definition/assumerole_codebuild_apply.yml

```
1 version: 0.2
2
3 phases:
4   pre_build:
5     commands:
6       - chmod a+x ./utils/*
7       - eval $(./utils/aws-cli-assumerole.sh -r $ASSUME_ROLE_NAME)
```

YAML Showing the top match Last indexed on Mar 25, 2021

Advanced search Cheat sheet

Note that the **NOT** qualifier can only be used for string keywords. It doesn't work for numerals or dates.

You can also prefix any search qualifier with a **-** to exclude all results that the qualifier matches.

```
CfnAuthorizer extension:ts -filename:authorizer.ts
```

CfnAuthorizer extension.ts -filename:authorizer.ts

268 code results

Sort: Best match

Repositories 0

Code 268

Commits 11

Issues 28

Discussions 1

Packages 0

Marketplace 0

Topics 0

Wikis 0

Users 0

Languages

TypeScript 268

Advanced search Cheat sheet

LarouTech/foto-architecture-stacks

lib/foto-apiGateway-stack.ts

```

7 import { CognitoProps } from "../interfaces/cognito-props";
8 import { CfnAuthorizer } from "aws-cdk-lib/aws-apigateway";
9
10 interface CustomProps extends Route53Props, FotoLambdaProps, CognitoProps {
11   ...
12   this.requestValidator(fotoApi, 'this.validator');
13 }
14
15 //Cloud formation Authorizer
16 const cfnAuthorizer = new CfnAuthorizer(this, 'Authorizer', {

```

TypeScript Showing the top two matches Last indexed on Jan 24

iRoachle/cdk-token-authorizer

lib/api/CustomApi.ts

```

1 import {
2   CfnAuthorizer,
3   HttpApi,
4   HttpApiProps,
5   HttpMethod,
6   HttpRouteKey,
7   ...
8 } from "aws-cdk-lib/aws-apigateway";
9
10 export class CustomHttpApi extends HttpApi {
11   public readonly authorizer: CfnAuthorizer;
12
13   constructor(scope: Construct, id: string, props?: CustomHttpApiProps) {

```

TypeScript Showing the top two matches Last indexed on Apr 17, 2021

You can prefix any search qualifier with a **-** to exclude all results that are matched by the qualifier.

For example, **-language**, **-filename**, **-path**, **-extension**, **-user**, **repo**, **-org**.

I often use the **-path** qualifier.

For example, if I navigate to the [Express.js](#) repository page and search for **render**, scoped to the repository, I get a bunch of test files.

render

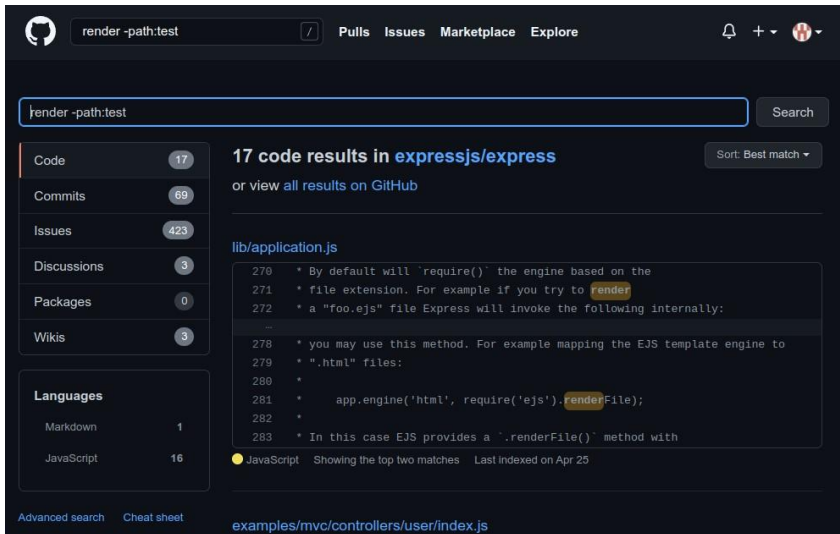
The screenshot shows the GitHub search interface with the query 'render'. The left sidebar displays statistics for the search results: Code (22), Commits (69), Issues (423), Discussions (3), Packages (0), and Wikis (3). Below this, the 'Languages' section shows Markdown (1) and JavaScript (21). The main content area shows '22 code results in expressjs/express' and a link to 'view all results on GitHub'. A code snippet from 'test/app.render.js' is displayed, showing a Jest test for the 'render' method. The snippet includes imports for 'path' and 'tpl', a 'describe' block for 'app.render', and a 'function' definition. The code is as follows:

```
5 var path = require('path')
6 var tpl = require('../support/tmpl');
7
8 describe('app', function(){
9   describe('.render(name, fn)', function(){
10     ...
11     var app = createApp();
12
13     app.locals.user = { name: 'tobi' };
14
15     app.render(path.join(__dirname, 'fixtures', 'user.tpl'), function
16       (err, str) {
```

Below the code snippet, it says 'JavaScript Showing the top two matches Last indexed on Feb 15'. Another code snippet for 'test/res.render.js' is partially visible at the bottom.

The tests are located in a `test/` directory, so they can be excluded by setting the `-path` qualifier to `test`.

```
render -path: test
```



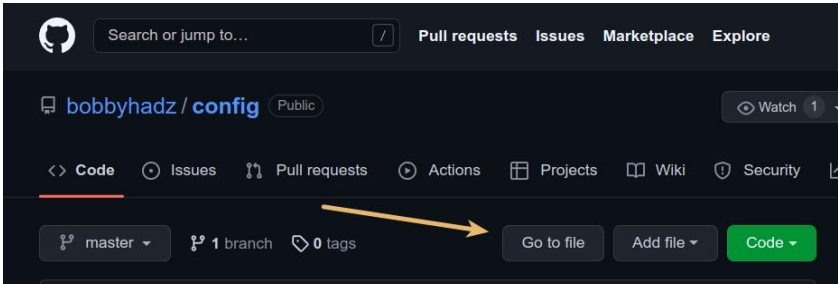
Search for Files on GitHub

There are 2 ways to search for files on GitHub:

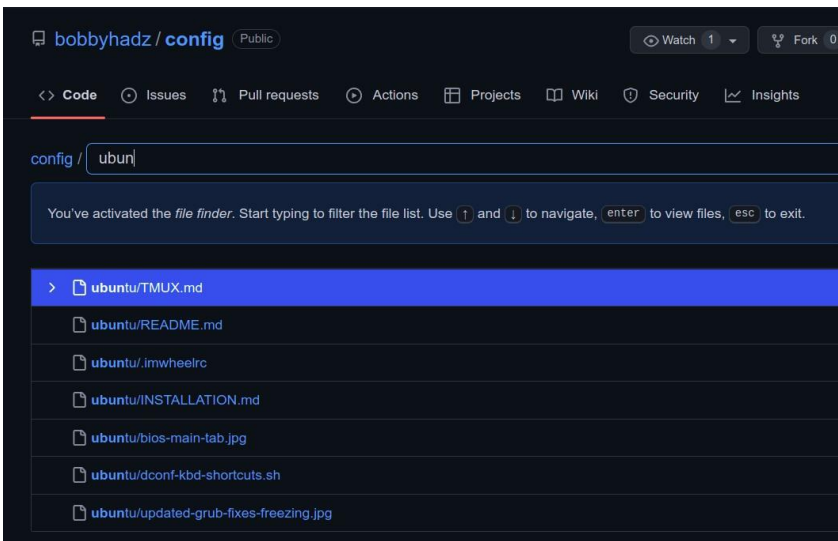
- use the **filename** search qualifier to search for a file in one or more repositories
- use the file finder GUI to search for files in a specific repository

To use the file finder GUI to search for files in a specific repository:

1. Open a GitHub repository, e.g. [bobbyhadz config](#).
 2. Click on the **Go to file** button located above the list of files in the repo
- Alternatively, press **t** on your keyboard to open the file finder



3. Start typing the name of the file you're looking for



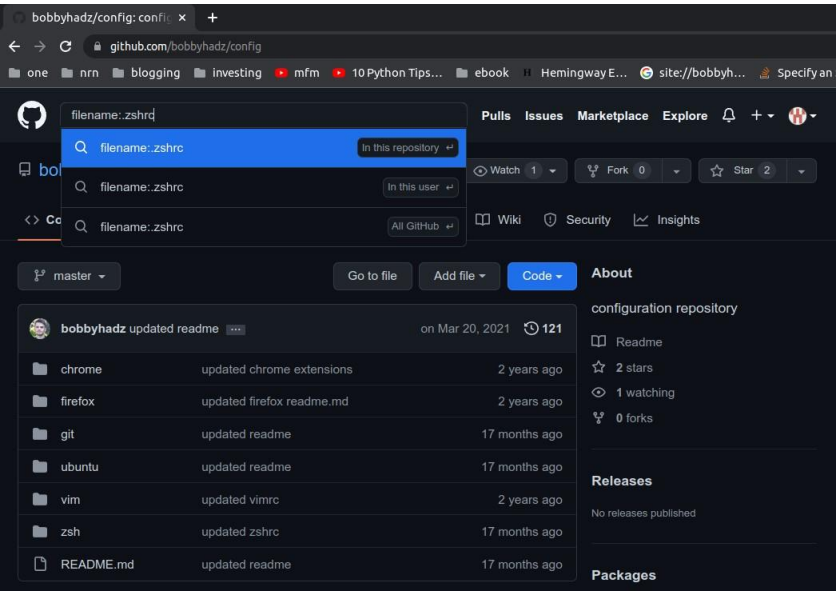
4. Select the file you are looking for.

Here are some examples of using the **filename** search qualifier.

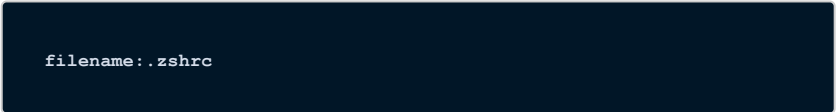
If you are on a page of a [specific repository](#), you'll be able to search for code:

- in the repository

- in all of the user's repositories
- in all of GitHub



I'll search for files named `.zshrc` scoped to a [specific repository](#) in this example.



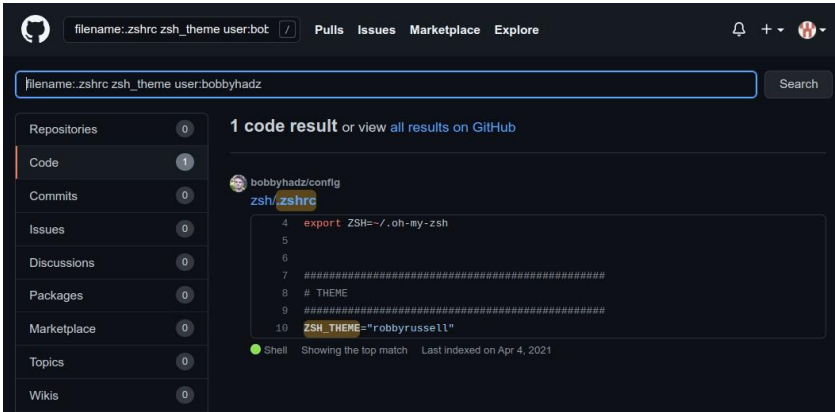
The following example matches files named `.zshrc` containing the word `zsh_theme`, in all of GitHub.

```
filename:.zshrc zsh_theme
```

The screenshot shows the GitHub search interface. The search bar contains the query `filename:.zshrc zsh_theme`. The left sidebar shows navigation options: Repositories (2K), Code (48K), Commits (62K), Issues (1K), Discussions (38), Packages (0), Marketplace (0), Topics (0), Wikis (538), and Users (0). Below this, the 'Languages' section shows Markdown (52) and Org (5). The main content area displays '52,569 code results'. The top result is from user 'jc9361/dotfiles' for the file `zsh/.zshrc`. The code snippet shows the definition of `ZSH_THEME` with various options like 'random', 'agnoster', 'nanotech', 'gallois', 'theunraveler', and 'wedisagree'. A second result is partially visible from user 'jens-eidot' for the file `files/.zshrc`, showing `export ZSH=$HOME/.oh-my-zsh` and `export ZSH_THEME="Soliah"`.

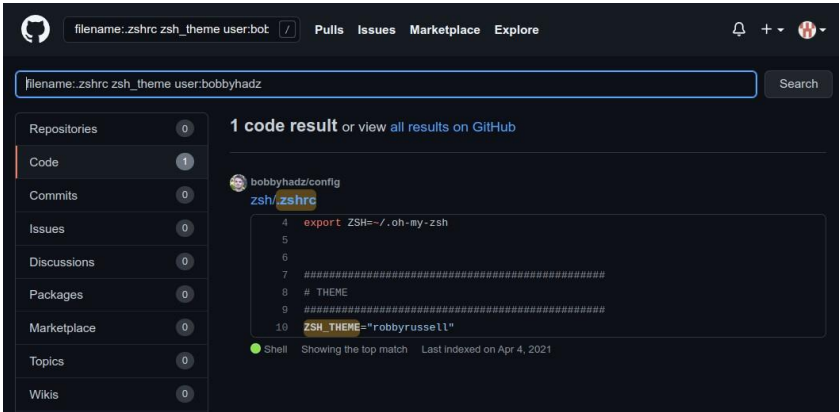
The following example matches files named `.zshrc` containing the word `zsh_theme`, scoped to the user `bobbyhadz`.

```
filename:.zshrc zsh_theme user:bobbyhadz
```



The following example matches `markdown` files named `.zshrc` containing the word `zsh_theme`, in all of GitHub.

```
filename:.zshrc zsh_theme language:markdown
```

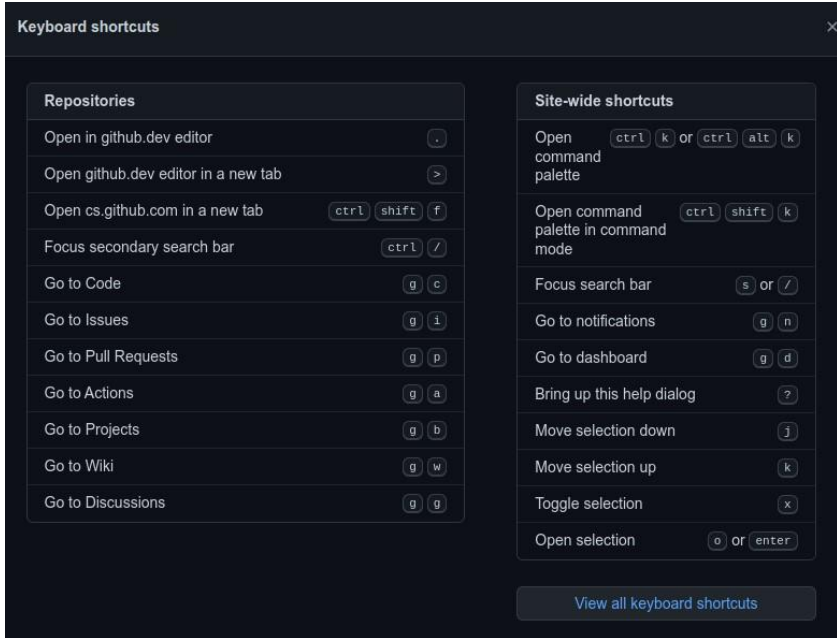


You can also [query for dates](#), e.g. to only match repositories with the word X that were pushed to before, after or between specific dates.

I haven't had to use these.

Useful Github keyboard shortcuts

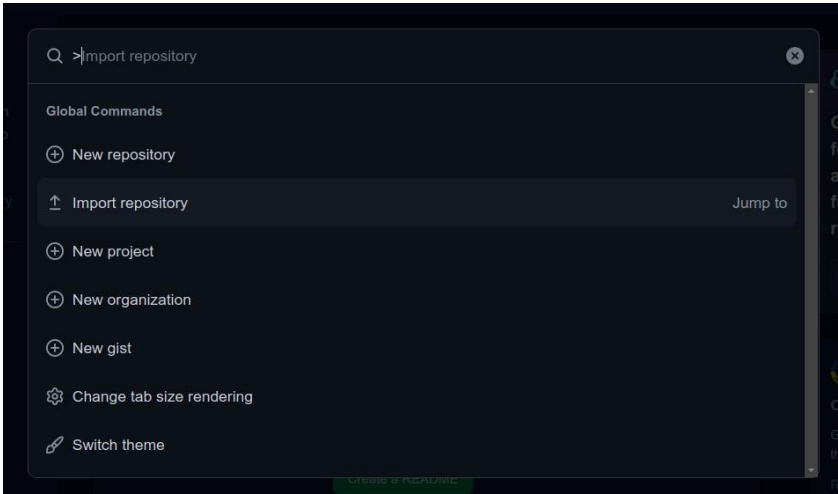
You can type **?** to show the available [keyboard shortcuts](#) for the specific Github page you are on.



Site-wide shortcuts

These are shortcuts you can use on any GitHub page.

- You can press **s** or **/** on any page to focus the search input field at the top
- **Ctrl K** (windows and linux) or **Command K** (Mac) to open the command palette.



Repositories

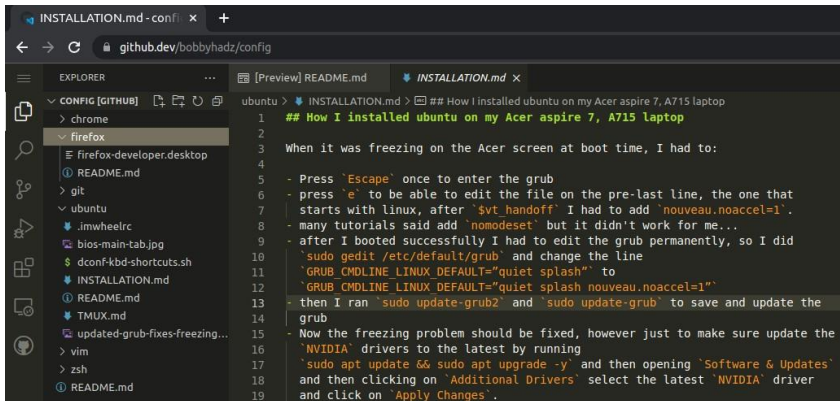
These are shortcuts you can use to navigate a [specific repository](#).

- **G C** - go to **Code** tab
- **G I** - go to **Issues** tab
- **G P** - go to the **Pull requests** tab

Source code editing

- [Open a repository](#) (or a pull request) and press dot **.** to open the repository (or the pull request) in the web-based editor.

This will open a VSCode editor with the contents of the repository directly in your browser.



- **Open a file** and press **E** to edit the file. This will fork the repository automatically.

Source code browsing

- **Open a file** (not in Edit mode) and press **L** to navigate to a specific line.
- **Open a repository** and press **T** to open file finder
- **Open a repository** and press **W** to switch to a different branch link to line
- or multiple lines in a github file

To create a permalink to a code snippet:

1. **Open the file.**
2. Click on the start line number.

The screenshot shows the GitHub interface for the repository 'bobbyhadz / config'. The file 'config / zsh / .zshrc' is selected. The file content is displayed in a dark-themed editor. A yellow arrow points to the three dots menu icon on the left side of the code editor.

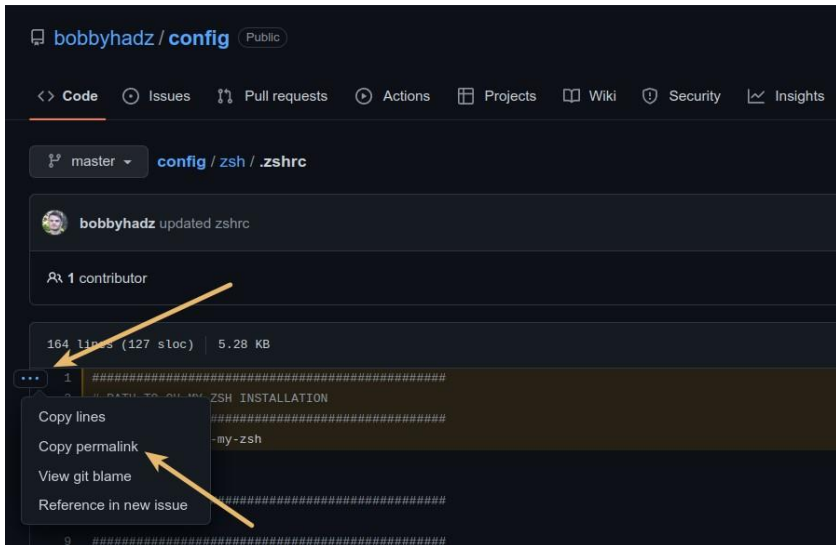
```
1 #####
2 # PATH TO OH-MY-ZSH INSTALLATION
3 #####
4 export ZSH=~/.oh-my-zsh
5
6
```

3. Press **CTRL Shift**.
4. Click on the end line number.

The screenshot shows the same GitHub repository and file. A yellow arrow points to the three dots menu icon on the left side of the code editor.

```
1 #####
2 # PATH TO OH-MY-ZSH INSTALLATION
3 #####
4 export ZSH=~/.oh-my-zsh
5
6
```

5. Click on the three dots and then click "Copy permalink".



If you need to permalink to a file in a specific commit, press **Y**. This will permanently link to the exact version of the file in that commit.

This will update the URL to contain the commit hash, e.g.:

<https://github.com/expressjs/express/blob/2c47827053233e707536019a15499ccf5496dc9d/lib/view.js>.

Here is a list of all the [available GitHub keyboard shortcuts](#).

Read open source code

There are some conventions that most Github repositories follow:

```
src/           # (or `lib`) - source files for the package test/
# (or `tests` or `spec`) - tests files
.gitignore     # specifies intentionally untracked files
CONTRIBUTING.md # how to contribute to the package
README.md      # what the package is about
LICENSE        # who can use the code / for what purposes
```

You'll spend most of your time in the following 2 directories:

- `src` (or `lib`) when reading source files
- `test` (or `tests`, `spec`, `__tests__`) when reading the package's tests

Here are 2 good repositories to look around:

- [Express.js](#) - if you use JavaScript
- [Flask](#) - if you use Python

You most often read open source code to:

- gain confidence that you know what a method actually does
- debug an error message you got while using the package. There has to be some code that throws the error when a certain condition is met •
be able to change what the module does
- use undocumented functions or classes. Avoid doing this if possible, it might make version updates more difficult

Even though the directory structure of most open source projects is straightforward, you will most often use the search input field to look for a particular function name, class name or an error message, scoped to the specific Github repository.

Here is an example that searches for files that contain the word `render` in the [Express.js](#) repository.

render

The screenshot shows the GitHub search interface for the term 'render'. The top navigation bar includes the GitHub logo, the search term 'render', and links to 'Pulls', 'Issues', 'Marketplace', and 'Explore'. On the left sidebar, there are filters for 'Code' (22), 'Commits' (69), 'Issues' (423), 'Discussions' (3), 'Packages' (0), and 'Wikis' (3). Below these are 'Languages' with 'Markdown' (1) and 'JavaScript' (21). The main content area displays '22 code results in expressjs/express' with a 'Sort: Best match' dropdown. It shows a snippet from 'test/app.render.js' with the following code:

```
5 var path = require('path')
6 var tmpl = require('../support/tmpl');
7
8 describe('app', function(){
9   describe('render(name, fn)', function(){
10     ...
11     var app = createApp();
12
13     app.locals.user = { name: 'tobi' };
14
15     app.render(path.join(__dirname, 'fixtures', 'user.tmpl'), function
16       (err, str) {
```

At the bottom, it indicates 'JavaScript Showing the top two matches' and 'Last indexed on Feb 15'. Below the code snippet, the file path 'test/res.render.js' is visible.

Once you find the function or class you are looking for, you can click on its name for more search-based code navigation.

For example, if you open [the view.js](#) file and **CTRL F** to find **function View**, you can click on the function's name to view its definitions and its references.

```
48 * @param {object} options
49 * @public
50 */
51
52 function View(name, options) {
53   var options = options || {};
54
55   this.name = name;
56   this.options = options;
57   this.render = function() {
58     // ...
59   };
60   if (this.options.render) {
61     this.render();
62   }
63
64   var fragment = this.render();
65   if (this.options.render) {
66     // ...
67   }
68   this.text = this.defaultEngine[0] === ' ' ? ' ' + this.defaultEngine : this.defaultEngine;
69   return this.render();
70 }
```

DefinitionsReferences

Present in 2 files

lib/view.js

52 function View(name, options) {

test/app.render.js

64 function View(name, options){

210 function View(name, options){

234 function View(name, options){

264 function View(name, options){

339 function View(name, options){

Github is going to show you where the function or method is defined and where it's being referenced in the repository.

Whether you'll be able to decipher what the code does depends on the quality of the code and your experience level.

If you can't understand what a function does, you can check if it has tests by using a repository-scoped search with the function's name.

The tests will show you how the function should and should not be used.

Most open source projects write relatively clean and easy to follow code (unless they are written in multiple languages) and write comments above difficult to understand code.

You can click on the different functions and methods to open their definitions if you want to dive deeper. Alternatively, you can use the search input to look for the functions or methods scoped to the specific repository.

The next chapter covers how to make contributions to open source repositories (for beginners).

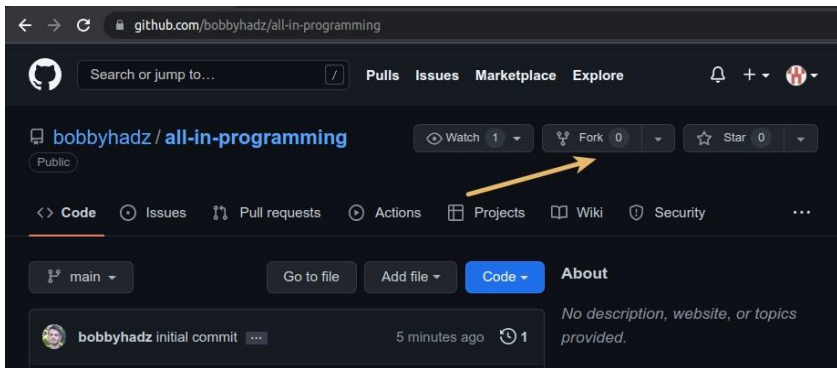
Contribute to open source code

Let's look at an example of contributing to an open source repository.

The repository we will be contributing to is [bobbyhadz/all-in-programming](#).

Here are the steps you have to follow to make your first pull request. The steps are also available in the **CONTRIBUTING.md** file in the root of the repository.

1. Fork [the repository](#).



The screenshot shows the GitHub interface for creating a new fork. At the top, there's a navigation bar with the GitHub logo, a search bar, and links to Pulls, Issues, Marketplace, and Explore. Below this, the repository name 'bobbyhadz / all-in-programming' is displayed, along with buttons for Watch (1), Fork (0), and Star (0). A secondary navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, and Security. The main heading is 'Create a new fork', followed by a brief explanation: 'A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.' Below this, there are two input fields: 'Owner' with a dropdown menu showing 'bobbyhadz2' and 'Repository name' with a text input containing 'all-in-programming' and a green checkmark. A note states: 'By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.' There is an optional 'Description' field. A checkbox labeled 'Copy the main branch only' is checked, with a note: 'Contribute back to bobbyhadz/all-in-programming by adding your own branch. Learn more.' A warning icon and text say: 'You are creating a fork in your personal account.' At the bottom, there is a blue 'Create fork' button, which is pointed to by a yellow arrow.

Search or jump to...

Pulls Issues Marketplace Explore

bobbyhadz / all-in-programming

Public

Watch 1 Fork 0 Star 0

<> Code Issues Pull requests Actions Projects Wiki Security

Create a new fork

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Owner ^{*} Repository name ^{*}

bobbyhadz2 / all-in-programming ✓

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

Description (optional)

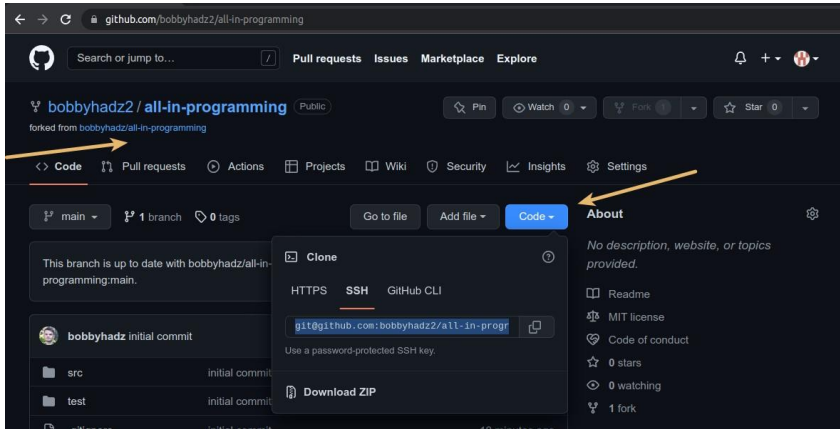
☒ Copy the `main` branch only
Contribute back to bobbyhadz/all-in-programming by adding your own branch. [Learn more.](#)

ⁱ You are creating a fork in your personal account.

Create fork

2. Git clone the fork using the SSH URL. Make sure to replace the **YOUR_GITHUB_HANDLE** placeholder with your actual username.

```
git clone git@github.com:YOUR_GITHUB_HANDLE/all-in-programming.git
```



3. Change to the directory of the repository.

```
cd all-in-programming
```

4. Pull from the original repository.

```
git pull git@github.com:bobbyhadz/all-in-programming.git -v
```

5. Create a new branch and switch to it. Make sure to pick a name that represents a function that hasn't been added to the **src** folder.

```
git checkout -b feat/add_5_and_5
```

6. Install the dependencies.

```
npm install
```

7. Run the tests before making any changes

```
npm run test
```

Alternatively, you can run the tests in watch mode to have them re-run every time you make a change.

```
npm run test:watch
```

8. Create a new file in the `src` directly (e.g. `src/sum5And5.js`) and define a `sum` function that adds 2 numbers. You can use any of the other `sum` functions as a template.

```
// src/sum5And5.js

export function sum5And5 () {
  return 5 + 5;
}
```

9. Add a test for the function in the `test` directory, e.g. under `test/sum5And5.test.js`. You can use the `other test files` as a template.

```
// test/sum5And5.test.js

import {sum5And5} from '../src/sum5And5' ;
```

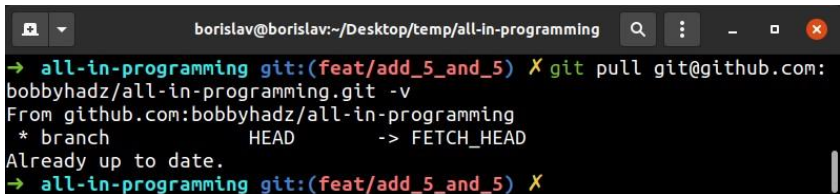
```
it('returns 10 when adding 5 and 5' , () => {  
  expect(sum5And5()) .toBe( 10 );  
});
```

10. After you finish, re-run the tests to make sure all tests pass with the changes you've made.

```
npm run test
```

11. Pull in changes from the original remote repository to stay up to date and avoid merge conflicts.

```
git pull git@github.com:bobbyhadz/all-in-programming.git -v
```



A terminal window screenshot showing the execution of a git pull command. The window title is 'borislav@borislav:~/Desktop/temp/all-in-programming'. The prompt is '→ all-in-programming git:(feat/add_5_and_5)'. The command entered is 'git pull git@github.com:bobbyhadz/all-in-programming.git -v'. The output shows the remote repository being fetched and the local branch being updated. The final status is 'Already up to date.' and the prompt returns to '→ all-in-programming git:(feat/add_5_and_5)'.

```
borislav@borislav:~/Desktop/temp/all-in-programming  
→ all-in-programming git:(feat/add_5_and_5) X git pull git@github.com:  
bobbyhadz/all-in-programming.git -v  
From github.com:bobbyhadz/all-in-programming  
* branch      HEAD      -> FETCH_HEAD  
Already up to date.  
→ all-in-programming git:(feat/add_5_and_5) X
```

12. Add and commit the changes you made.

```
git add .  
  
git commit -m 'add sum5And5 function'
```

```
borislav@borislav:~/Desktop/temp/all-in-programming
→ all-in-programming git:(feat/add_5_and_5) X git add .
→ all-in-programming git:(feat/add_5_and_5) X git commit -m 'add sum5And5 function'

[feat/add_5_and_5 e34d5d7] add sum5And5 function
2 files changed, 10 insertions(+)
create mode 100644 src/sum5And5.js
create mode 100644 test/sum5And5.test.js
→ all-in-programming git:(feat/add_5_and_5)
```

13. Run the `git push` command and look at the error message.

```
git push
```

```
borislav@borislav:~/Desktop/temp/all-in-programming
→ all-in-programming git:(feat/add_5_and_5) git push
fatal: The current branch feat/add_5_and_5 has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feat/add_5_and_5
→ all-in-programming git:(feat/add_5_and_5)
```

The error message will tell you what command you need to run to push the current branch and set the remote as upstream.

14. Run the command from the error message. You only have to run this command the first time you push to the remote branch.

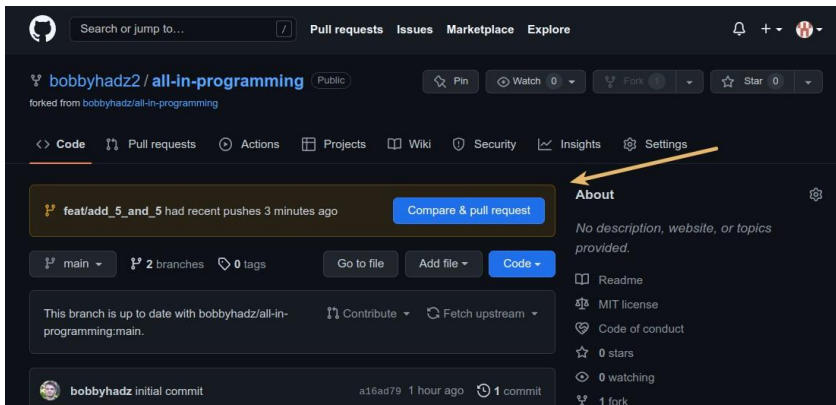
```
git push --set-upstream origin feat/add_5_and_5
```

```
borislav@borislav:~/Desktop/temp/all-in-programming
→ all-in-programming git:(feat/add_5_and_5) git push --set-upstream or
igin feat/add_5_and_5
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 12 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 640 bytes | 640.00 KiB/s, done.
Total 6 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feat/add_5_and_5' on GitHub by visit
ing:
remote:      https://github.com/bobbyhadz2/all-in-programming/pull/new/
feat/add_5_and_5
remote:
To github.com:bobbyhadz2/all-in-programming.git
 * [new branch]      feat/add_5_and_5 -> feat/add_5_and_5
Branch 'feat/add_5_and_5' set up to track remote branch 'feat/add_5_and
_5' from 'origin'.
→ all-in-programming git:(feat/add_5_and_5)
```

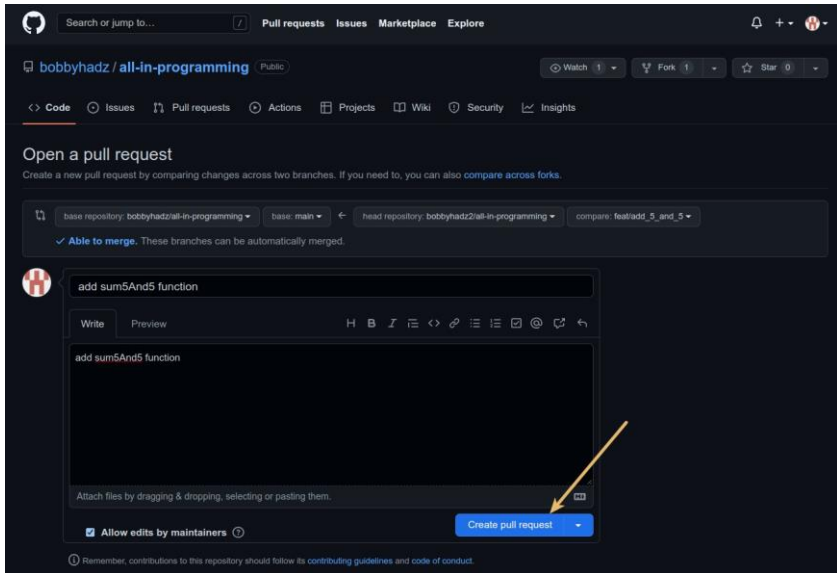
15. The next time you need to push, you can simply run the `git push` command.

```
git push -v
```

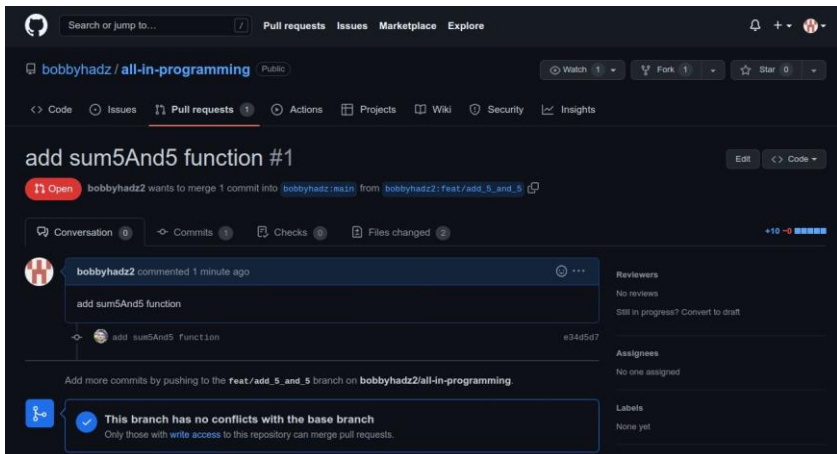
16. Open your Github fork and make a Pull request to the `main` branch of the original repository.



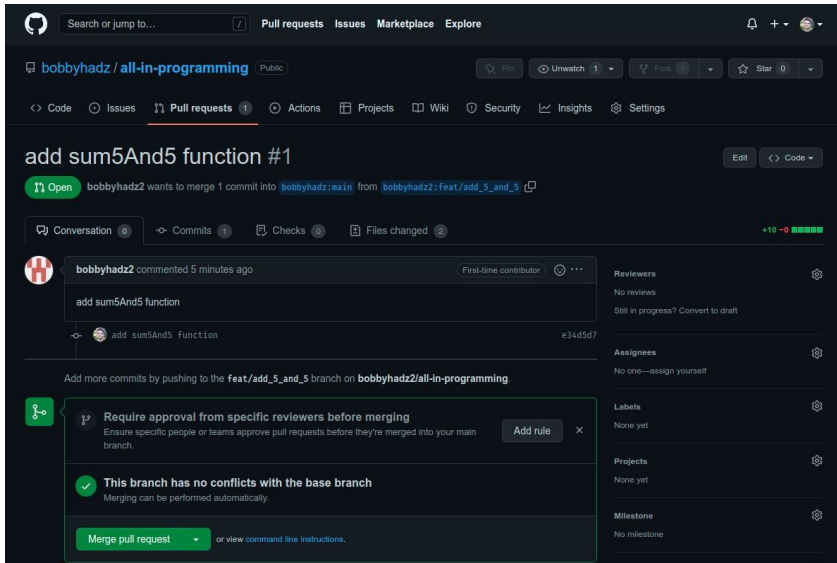
17. Optionally, leave a comment and click on the "Create Pull request" button.



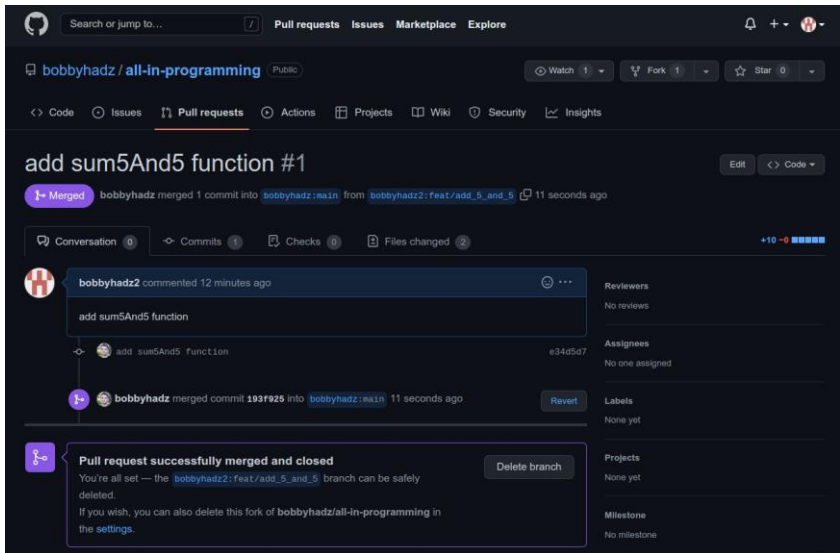
18. This is what an open pull request looks like for the person who made the contribution.



19. This is what the pull requests looks like for the maintainer of the repository.



20. I will merge your pull request as soon as possible!
21. This is what it looks like after I've merged your pull request.



If you want to learn more about contributing to open source repositories, check out:

- [Kent C. Dodds](#)
- [CJ from Coding Garden](#)

Shout-out to [CJ from Coding Garden](#) who is by far the best programming streamer I've ever watched.

It's very difficult to write code and answer questions in real time. He does a great job.

Outro

Thank you for reading!

Please, don't take what's written in this book as "advice". I am not qualified to give advice to anyone about anything.

What works for me might not work for you.

Take what's written in this book as food for thought and a source of ideas.

Do your own research, draw your own conclusions.

I'll leave you with a book recommendation.

It's a free book called "The Almanack of Naval Ravikant: A Guide to Wealth and Happiness".

You can download it for free by going to navymanack.com and clicking on the "DOWNLOAD FREE PDF" button.

Out of everything I've read, this is the book I got the most value out of.