HOW TO GET

Your First Dev Job

Learn to debug the hiring process and optimize your applications

by Johannes Kettmann @ Profy.dev

Introduction

At the beginning of a career in software development, you can be overwhelmed by all the things you seemingly have to learn and know. You may feel like the only person who doesn't understand or know something. You may feel stupid and inadequate.

If you do please know that even experienced developers still feel this way from time to time. There's always a new framework or library that you "should" learn and know about. But it gets better and you'll get used to it. In every project I work on there are situations where someone tells me: "Oh... why didn't you do it another way?" and provides a solution in a minute... and I spent an entire day creating something that's worse.

The same goes for job postings. Don't get intimidated by requirements for skills and experience. Apply anyways. Don't fall for imposter syndrome. Read a job posting like this: Of the required skills the first 30% are important, the rest is nice to have. And even if you don't know all of the important skills or don't meet their experience requirements, still apply. You never know...

This book consists of 7 chapters. Here is an outline of what to expect.

Table of Content

About The Author

Hi, I'm Johannes. I'm a self-taught full-stack JavaScript developer. I took a somewhat complicated route to become a professional programmer. My first baby steps were in Arduino and Android development in my mid-twenties and I moved to web development with React sometime later.

Along the road, I tried to succeed with some "start-ups"... all failed. I picked up my first freelancing job... and didn't get paid. My first job interviews were horrible: I didn't have a clue about professional programming and sucked at technical interviews.

But at some point thanks to my "start-up" projects I had a nice portfolio and gained something you might call experience.

Since then I didn't have any problems finding new job opportunities. Now I'm working as a contractor. I often can pick from multiple offers. Most importantly I have the luxury and freedom to take several months of vacation each year.

As a contractor, I switch my jobs regularly and have been through lots of interviews. At the same time, I was often part of the hiring process inside the companies that I was working for. I've been on both sides of the table.

I hope I can make your path a bit easier by sharing the experience and insights I gained.

If you're interested I'd be happy to connect. You can find me here on Twitter or LinkedIn.

Chapter 1

The Hiring Process

If you're already applying for jobs as a software developer and didn't have much luck so far you might be wondering what you're doing wrong.

Before we have a look at the process that allows you to turn rejection into valuable feedback let's make sure that we're on the same ground and get a clear picture of what it needs to get hired. The hiring process varies from company to company. But there is a typical path every dev has to go. We will shine a light on each step involved.

Note: If you're thinking of applying for a job at a FAANG company there are much more detailed resources about their specific hiring processes.

Steps of a typical hiring process

1. HR or recruiters
2. Hiring manager
3. Coding test
4. Technical interview

When I was applying for my first jobs just thinking of a technical interview or a coding test was enough to freak me out and send my heart rate through the roof. Me coding in front of professional developers?!?

So don't worry if you feel the same. You're not alone.

Since it can be intimidating let’s think of the whole process as a game. There are four levels you have to pass with tiny enemies, some mini-bosses, and then the big final boss in the last level. As you overcome these enemies you will pass from one level to the next until you finally discover the secret treasure.

So let's jump right into the first level!

Level 1: HR or recruiters

The HR department or recruiters are the first enemies. Not hard to beat but they have the power to reject your application.

These people will have a look at your resume once they receive your application. If your resume looks nice and it's readable you already have increased your chances. They will probably check your skills again to see if you meet the requirements. Your background and a nice but short summary may have an impact as well.

Once they consider you as a potential candidate you will typically have a short phone call of about 15-30 minutes. HR personnel or recruiters are usually non-technical people so they won't ask deep questions. They may poke around and throw out some buzzwords ("do you have experience with redux?") but they won't exactly know what that is. They might not even be aware that JavaScript and Java aren't the same things. So don't get nervous. You can throw back some buzzwords ("yeah, I'm familiar with writing reducers and actions and the Flux pattern in general"). That should be enough.

But be sure to be nice. Smile while you're talking. These people are the first gatekeepers to your potential job. They might even put in a good word for you with the hiring manager. Leaving a good first impression is very important here.

Level 2: The hiring manager

The hiring manager is the first considerable mini-boss. Sometimes not so mini in fact.

They are the person who makes the final decision about whether or not you get the job. Depending on the company's size they are more or less technical and may have more or less deep knowledge of the required technologies (aka your skills). They could be the CTO, an engineering manager, or a team lead.

On this level, you will typically have another phone call. This time a bit longer (around 30-60 min) and more in-depth. But mostly the questions asked here will be comparably easy. They will talk about the company and project you would be working on and allow you to ask some general questions in return. Don't be surprised if they don't know the details of the tech stack or workflows within the team. Likewise, you will be asked to talk about yourself and your experiences. So it's wise to prepare a short storyline about one of your projects. Then you'll be able to talk confidently and show your skills without the need to order your thoughts first.

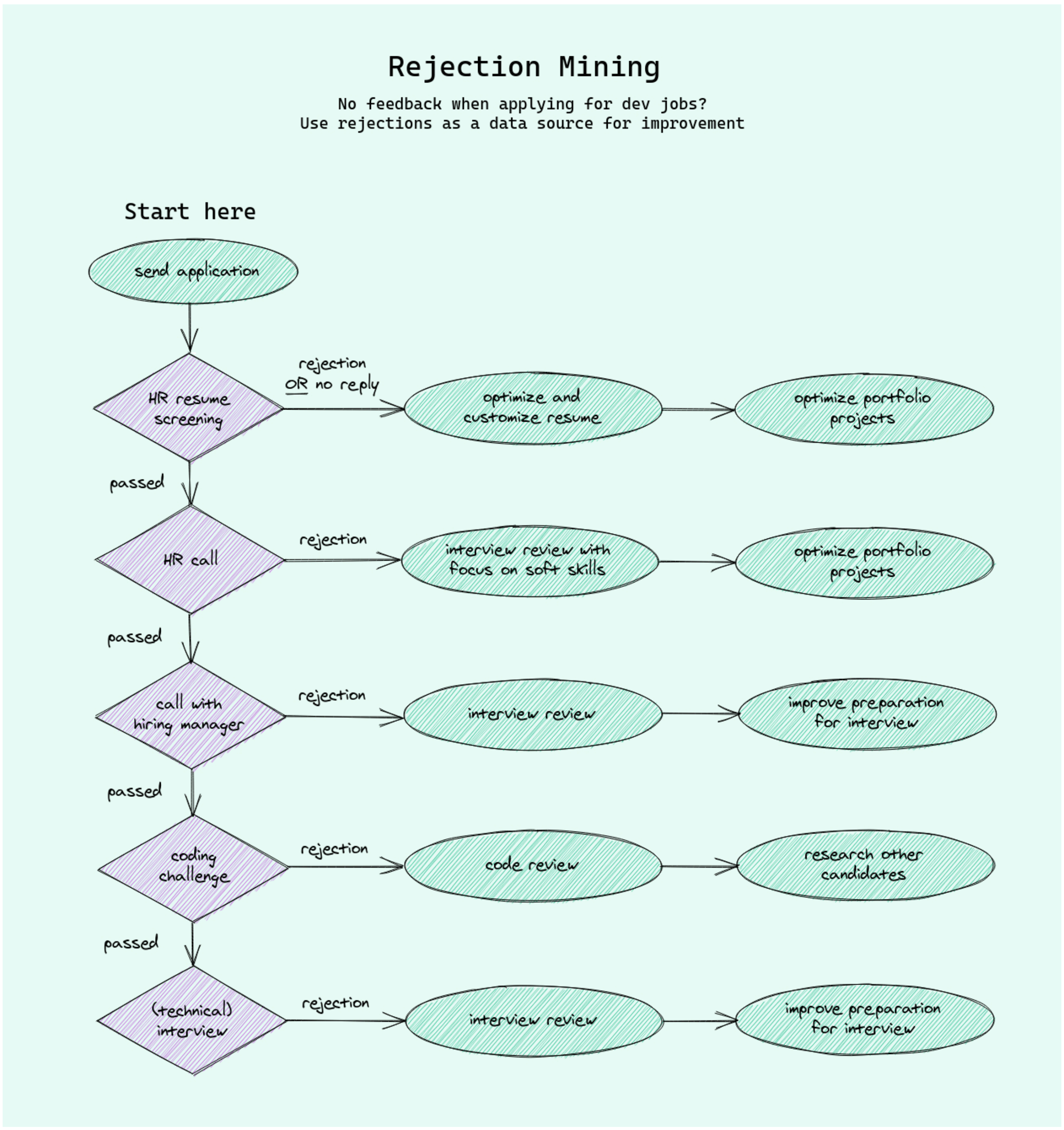
Level 3: The coding test

Naturally, the boss on the next level is a bit tougher to beat.

The coding test can be a nice exercise at home or a stressful life coding session at the office or via screen share.

If you're lucky and get an assignment to work on at home don't be too stingy with your time and hand the results in soon. Clean the code by using a linter. Add an informative readme including instructions on how to install and run the project. Maybe add some details about your technical decisions. If you know how to write tests add some tests to impress the reviewer.

Most importantly: make sure it works! It's frustrating to receive a candidate's code just to find out that it throws errors when you try to run it.


Rejection Mining
No feedback when applying for dev jobs?
Use rejections as a data source for improvement
Start here
send application
HR resume screening
rejection OR no reply
optimize and customize resume
optimize portfolio projects
passed
HR call
rejection
interview review with focus on soft skills
optimize portfolio projects
passed
call with hiring manager
rejection
interview review
improve preparation for interview
passed
coding challenge
rejection
code review
research other candidates
passed
(technical) interview
rejection
interview review
improve preparation for interview

If you find yourself having to code in front of others try not to freak out. This actually can be beneficial for you. The task probably won't be as difficult. Make sure to talk about everything you're trying to do. Guide them through your thought process. That's what they want to see. Here you can show off your communication skills.

If you're nervous that's totally normal. You might just tell them openly. But don't stop talking about what you're doing.

The assignment is often too big to finish. So no worries if you can't implement everything. You're also allowed to use Google for help. Everyone does that in their day-to-day work. Just be sure not to use any technology you don't really know.

I once had an interview with a life coding session via screencasts when React hooks just came out. I had used them before but wasn't familiar yet. Part of the task was to fetch some data from an API and render it. I decided to implement the data fetching with the new useEffect. Of course, I failed. Something didn't work and I wasn't sure why. But the interviewer liked my communication skills. I wasn't programming very fast but I was talking about every step I did. So even though I wasn't able to deliver I still passed the test.

Level 4: The technical interview

The technical interview is the final boss. At least for me, this is the hardest part of the journey. Once you beat this final enemy you'll have a job offer!

There are different kinds of questions like specific technical, general technical, design and architecture, teamwork, and soft skills.

Specific technical questions are more or less difficult questions about specific technologies like JavaScript, React, or CSS. These are knowledge questions.

It's okay not to know something. Don't start guessing in such a situation. Of course, the interviewers want to see that you know some things. At the same time, they want to be sure that you ask your teammates if you're not certain.

General technical questions can be "You enter a URL in the browser and hit enter. What happens?" (Urgh I hate this question) or "How would you describe and ensure code quality?". The interviewers try to get you talking and learn something about the way you think.

They might also try to push you into a corner by digging deeper and deeper to see how you react. Just try to keep your cool and say that you don't know at some point. Again, don't make stuff up. This kind of interview can be uncomfortable but it's designed to be this way.

Design and architectural questions are about sketching a solution to a given problem. For frontend positions, a problem might be to render an array of objects into a grid and you're supposed to explain what kind of components you would create. If you're more experienced they might ask how you would structure a full-stack online shop or food delivery app. There's no one correct solution. Again they want to get to know your thought process.

The team and soft skills questions might be about git or scrum workflows or how you would react and solve a conflict situation. It should be ok not to know much about workflows for entry-level positions though. Just be open and say that you'd like to learn.

Summary

This was it for the first chapter. We talked about the different stages of a typical hiring process:

1. A short interview with HR or recruiters
2. An interview with the hiring manager
3. Coding test
4. Technical interview

Note that many Junior developers report that they get a coding test very early in the process. Often before the very first interview. This might be a way for the company to save time. Conducting interviews consumes a lot of time and they may not want to waste it on developers who are not skilled enough for the job. So they filter the good candidates via a code challenge before investing time into personal interviews.

Now that you have an overview of the hiring process you'll learn a process that can tell you where in the hiring process you're stuck and what to tweak in order to become an outstanding candidate.

Chapter 2

How To Use Rejections As Feedback

When you start applying for jobs you will quickly understand that the whole hiring process feels like a black box. You send an application in. Something happens. And out comes a rejection or, if you're lucky, an invitation to an interview or a coding challenge.

The problem starts when you get rejections and can't seem to pass one of the stages of the process. The reason is that one important thing is missing:

You don't get any feedback

When your application is rejected you're asking yourself what went wrong. But usually, you only receive generic messages like "We've decided to move forward with another candidate". That doesn't provide a lot of material for improvement, right?

But what if you didn't need feedback? What if you knew what to improve in order to get to the next stage of the hiring process? It would certainly make your life easier. You could use each application to become a better candidate.

Of course, there's no magic bullet. You can't determine the real reason why you've been rejected. Sometimes it's just bad luck.

But with the process, you'll learn in this chapter you will have **a tool that points you at potential problems without receiving any direct feedback**. We will make use of the information you already have:

- Where in the hiring process are you stuck?
- What kind of rejections do you get?
- What are your experiences during interviews?

Before we continue let's recall the different stages of a typical hiring process we talked about in the last chapter:

1. Application filters
2. HR or recruiters
3. Hiring manager
4. Coding test
5. Technical interview

You can get stuck on any of these stages. You might get automatic rejections from application filters. Maybe we rarely get to the interview stage and usually get rejected by an HR person.

This tells us something. It can point us to an issue e.g. with our resume.

So in order to get better results from our applications, we can gather data from the rejections. That's why I call this process

Let's have a closer at the process. I prepared the map above so it's easier to get an overview. You can start at the top and follow the arrows.

Note: The first stage of the hiring process **HR or recruiters** *is split into two steps here (the screening of the resume and a phone interview) since a candidate may be rejected after any of these steps*.

In the following sections, we will walk through the map and get short explanations of each step. To not bloat this chapter we won't go into much detail though. That's what the other chapters are for

1. HR resume screening

If you don't get replies or you get rejected by a human you might be stuck at the resume screening by an HR person.

You should focus on optimizing and customizing your resume. Reorder and highlight skills and keywords important for the job, double-check that the formatting is consistent and looks nice. You might want to remove unimportant skills. Try to optimize your resume so that everything important is visible as easily as possible.

If you're certain that your resume is fine have a look at your portfolio projects. Being rejected at this stage can also mean that a technical person already had a look at your code. Add links to your portfolio projects (source and deployed version) on the resume. Make sure that you have a nice readme file and that the code is formatted nicely.

2. HR call

You're stuck here if you get invited to a call with an HR person but don't get to the next stage.

You should focus on improving your interviewing skills. This interview shouldn't be hard to pass from a technical perspective. A technique that might help you is the interview review described at the end of chapter 5 with a focus on soft skills.

The technique applied to the HR call in a nutshell: Step through the interview from hindsight. Ask yourself whether you were friendly on the phone. Did you show interest in the job? Did they ask questions related to company culture and how did you answer them?

Again, if you can't find any issues with the call itself you should have a look at your portfolio projects. Maybe a technical person had a look at your code only after the call and decided not to continue the hiring process with you.

3. The hiring manager

You're stuck here if you don't get to the next stage after a phone interview with the hiring manager.

You should focus on creating a storyline to have something to talk about, researching the company, and preparing some general questions. Again, use the interview review technique (chapter 5) to get a better picture of the issues of your interviews. Be sure to look up terms you didn't know an be better prepared the next time.

4. The coding challenge

You're stuck here if you get a coding challenge and turn it in but get rejected.

You can train coding challenges by solving small tasks over and over again. Find tasks that are similar to the ones that came up in the challenge. Like fetching data from an API and rendering it. Find an open API, initialize a project (for example with create-react-app) and write the code. Repeat until you're comfortable.

If you want to be sure that there are no red flags in your code you can ask for a code review in online communities like Reddit. Make it easy for other developers to review your code. If you can narrow down on potential problems on your own and only share a small piece of code you have better chances for getting a review.

If the coding challenge was a take-home assignment and contained a unique string (like a URL for a custom API endpoint or some mock data) you can try to find solutions of other candidates. The direct comparison might help you understand what you could have done better.

5. The technical interview

You're stuck here if you are invited to a longer technical (usually on-site) interview but don't get an offer.

You should focus on analyzing why the interview didn't go well and improving your weak spots using the interview review technique. There's a variety of types of questions and things that might go wrong in technical interviews. They are for many people the most challenging part of the hiring process. We will have a detailed look in a later email.

Summary

In this lesson, we learned a technique that can help you narrow down on issues you experience during the hiring process.

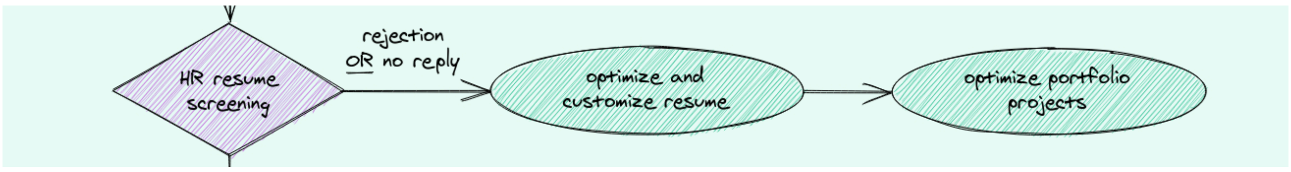
Of course, it's not always as simple as described. You might not get an offer because an interview and the coding tests in combination weren't so great. Sometimes there's just a better candidate... not much you can do about that. But I hope you now have a tool that can help you navigate through the hiring process.

In the next lesson, we'll have a more detailed look at how to improve your resume.

Chapter 3

The Resume

In the previous lesson, you learned a process to use rejections for your benefit. Remember the map that visualized the process? Today we will talk about the first to step.



You should focus on improving your resume if you

- receive immediate rejections
- don't get any replies to your applications.

So let's talk about what a resume should contain and how you might improve yours.

Sections of a resume

1. Personal details
2. Summary
3. Skills
4. Experience
5. Personal projects
6. Education

The purpose of your resume is to get you to the interview stage. To achieve this goal it's most important to **make it as readable as you can**. Imagine a person seeing your resume for the first time: is everything relevant quickly readable and ideally pops into your eyes immediately?

If you're applying for your first software development job fit the complete resume on a single page. Once you gained some experience you can add another page. Restricting yourself to a short resume helps you remove unnecessary clutter.

Note: In some countries, you'd rather use a CV instead of a resume. In comparison to a resume, a CV is a longer document that contains the history of your whole career. It's more objective and isn't customized as much. Europeans tend to apply with a CV, US-Americans with a resume, and Australians might use both. So inform yourself what's common in the country you're applying in. For the most part, the advice in this lesson is valid for a CV as well. If you're interested in a detailed comparison you can read this article.

Now let's have a look at the above sections one by one.

1. Personal details

This is simple. Double-check your contact details like email or phone number. Don't use a cryptic email address. Add relevant links for example to your portfolio, GitHub or LinkedIn. You might include links to StackOverflow (if you have posted some answers) or your blog or Medium profile (if you write about job-related topics).

2. Summary

I'd consider the summary section as optional. Many people won't even read it. If you switch careers though this is an opportunity to explain why.

If you want to write a summary you can use it as an overview of your skills and experience. For a junior developer position, you could write something like this:

"Front-end developer with a strong personal portfolio. Well-versed in the front-end ecosystem and quick to adopt and master new technologies that arise."

Make sure not to use common generic terms. Everyone is "highly motivated" or has "strong communication skills". Try to mirror words and requirements from the job posting. If they want a front-end developer, give them a front-end developer. If they want a quick learner, say that you are. Keep the summary to 1-2 sentences. Don't write a complete paragraph.

One last thing: Don't make yourself sound like a seasoned programmer if this is your first job. Don't oversell.

Note: I took the above example from a Reddit post. So don't copy-paste it just like this. Others have done so before...

3. Skills

The skills section is very important to make your relevant skills visible at first glance.

You can list the skills in order of the requirements in the job posting. You can also highlight important skills. Including your level of expertise for each skill can be helpful as well.

Restrict the list to around 10 skills. Generic or unspecific terms like "web development" don't belong here since they add clutter. Git, on the other hand, is a skill you want to mention. The same goes for Agile workflows if you have experience with it.

Another important point is to spell the technologies correctly. For example, write React instead of ReactJS. A quick Google search will help you verify the right names.

4. Experience

As mentioned before your resume should fit on a single page, especially for your first dev job. Listing only relevant experience saves a lot of space. It removes clutter and makes the resume more readable.

If you worked for example in UI design or as a project manager before you may mention these jobs. Still take care to keep it relevant and short.

If you built a personal app that runs in production, has users, and is intended to make money you should add it here as well IMO. This kind of app is mostly more than a small pet or fun project. You typically have some analytics tools and error monitoring installed. Once you have real users with different operating systems and browsers you'll see all kinds of problems. So this is usually a different kind and much deeper experience than a common portfolio project. As a role in the project, I would just mention "Founder".

I founded a couple of "startups" myself. All of them ended up in the experience section of my resume. I'm pretty sure that they were the reason that I was able to get my foot into the door.

If you did freelancing or contracting projects before you can also add them here. Sometimes these projects are very short-term so it might look strange. As if you were fired or couldn't get along with the team. So as a role you can, for example, say "Freelance software developer".

Some people already have worked as professional developers but they signed an NDA with their employer. This is no problem. Just write a couple of bullet points, as usual, explaining what you did or achieved, your responsibilities, technologies that were used, workflows like scrum, and so on. There should be lots to write about without naming specific details of the project.

5. Personal projects

If you don't have any experience as a software developer to mention in the above section you might want to switch the "Experience" and "Personal projects" sections.

Make sure to add links to the deployed app as well as the source code. Make it as easy as you can for any person to check out your stuff.

You can shortly explain what the app does. But keep it short, like 1-2 sentences. And of course, mention the tech stack.

You will get more tips for your personal projects in the next lesson.

6. Education

If you have a college degree of any kind make sure to put it on your resume. Any degree is a plus. It tells the employer that you're invested in learning and can finish what you started.

If your degree is unrelated keep it short. Otherwise, you can mention relevant classes. Especially, if you don't have anything to write in the experience section.

People who have studied computer science or similar can also move this section above the "Experience" and "Personal Projects" section.

Summary

This was the third chapter. We talked about the sections of a resume and what to write in each. I hope you found some of the tips helpful. The most important thing to keep in mind: make your resume readable and remove unnecessary clutter.

If you find it hard to come up with a good design there are tons of nice templates online. You can also use a free resume builder like flowcv.io. Below is an example resume created with this nice little tool.

In the next chapter, we will talk about how to stand out with your portfolio projects.

Jane Doe

React Developer

123 Maine St., Niche, Conneticut, 06068 jane@doe.com 432 - 547-2714

doe.com in janedoe janedoe

Skills

- React
- Redux
- JavaScript
- styled components
- HTML
- CSS
- Node.js
- Express

Profile

Front-end developer with a strong personal portfolio. Well-versed in the React ecosystem and quick to adopt and master new technologies that arise.

Projects

Reddit Timer

- Shows the best weekday and time to submit a post to Reddit. Uses the data of the top posts of the last year to create a heatmap as visualization.
- *React, Redux, Hooks, styled components, Axios*
- Source code

Remote Dev

- Job board for remote developers. Allows companies to create, update, and delete job listings.
- *React, Redux, Hooks, CSS modules, Node.js, Express, PostgreSQL*
- Source code

Todo App

- Simple Todo application that allows CRUD operations and stores items in a database.
- *React, CSS modules, Node.js, Express, MongoDB*
- Source code

Education

Ryerson University, *Bachelor of Science, Biology*
09/2016 – 07/2019 | Toronto, Canada

Chapter 4

The Portfolio Projects

If you optimized your resume but still don't get to the interview stage, be it a phone interview or on-site, there might be a problem with your portfolio projects.

Your personal projects are the best way to convince your future employer of your skills. Especially if you don't yet have professional experience.

When you think about creating a project you should always keep in mind what an employer wants to see. As a frontend developer, for example, you will have a couple of responsibilities. Most importantly you will write business logic and CSS on a daily basis. You will keep an eye on code quality and at least from time to time write documentation.

Your projects should be focused on showcasing these skills. So don't bother implementing lots of features at first. Developers who review your code often won't have the time or patience to look at all those features anyway.

If you're still building your GitHub portfolio check out the Reddit Analytics project on Profy.dev. It has all the features mentioned above. But it's not only a project that will look great on your portfolio. You also learn how to work on a professional team. Here is how it works in a nutshell:

- You implement the app based on professional designs and tasks.
- For each task you create a Pull Request on GitHub and request a review by the Profy bot.
- You receive a list of useful advice on how to improve your code.
- You compare your code to the implementation of an experienced professional React dev.

Whether you build the Reddit Analytics project or any other app you'll find this chapter useful.

We will look at some simple and advanced ways to create outstanding projects for your portfolio. On the next page you'll get a checklist that you can use to improve your existing projects. In the rest of the chapter I'll describe each point on the list in more detail.

The Portfolio Project Checklist

Profy.dev

- [ ] Double check that the app works
- [ ] Deploy a running version
- [ ] Add links to deployed app and source code in resume
- [ ] The app is easy to use
- [ ] Don't hide the app behind a login
- [ ] Well-structured and informative readme
- [ ] Clean code formatting
- [ ] Write custom CSS
- [ ] Somewhat complex logic
- [ ] Mobile responsiveness
- [ ] Pin your GitHub repos
- [ ] Don't use tutorial apps
- [ ] Semi-unique app
- [ ] Write tests
- [ ] Good Git workflow

1. Double check that the app works

That sounds kind of ridiculous but I've seen it multiple times. Either you enter the URL and there's only a "white screen of death" or you try to run the source code and only see errors. Make sure to test everything yourself before applying for a job.

2. Deploy a running version

Being able to take a look at the app is important for non-technical people. It will also make it easier to understand how it works for developers who review your source code. It's important that its response time is not totally slow, so don't use a free Heroku plan. When the app is not opened for some time Heroku needs to restart the application which takes quite some time (aka cold-start). Make sure people don't get bored and close the app before having a chance to look at it. I'd suggest Vercel or Netlify since it's super easy to deploy React apps from GitHub there. Takes only a few clicks.

3. Links to deployed app and source code in resume

Make it as easy as possible for anyone looking at the resume to check out your projects. Imagine having limited time to review a pile of applications. You don't want to be forced to enter a URL manually or scroll through a list of unordered projects on GitHub.

4. Easy to use

The UX doesn't need to be overwhelmingly great. But a new user should be able to understand what the project is doing. Think about someone who has never seen the app and doesn't know how it works. Will they understand what to do? Is it clear where they need to input data etc?

5. Don't hide the app behind a login

Again imagine a person with limited time. You don't want to force them to create an account before being able to access your app. If you need a login make sure to note the user credentials in the resume or prefill the login form.

6. Well-structured and informative readme

This should at least contain instructions to install and run the app as well as a link to a deployed version. You can use the readme to show off your skills and ability to communicate. Add sections where you explain your technical decisions and the structure of the code. You can also include a link and description to a place in your code with custom CSS (see 8) and some more complex business logic (see 9). Developers reviewing your app often won't have the time to step through the complete source code. So guiding them to the beautiful places may be advantageous.

7. Clean code formatting

This is really simple but yet a lot of junior candidates don't have a nicely formatted code base. Some files may have four spaces indentation, some only two. Use a tool like Eslint or prettier and format your code automatically.

8. Write custom CSS

It's okay to use a UI framework like bootstrap, material-ui, you name it. It's way easier to build an app that looks nice without a lot of design skills. A nice looking app can be a good way to leave a good impression after all. But your day-to-day work as a developer will most likely include writing lots of custom CSS. So be sure to write the styles of some feature yourself. Add some mobile responsiveness if you like. Also, see point 6.

9. Somewhat complex logic

Another big part of your responsibilities will be writing business logic. So make sure you have at least one feature where you implement something more complex than iterating over an array and rendering the contained objects. Transform some data. Make use of some array functions like map, filter or reduce. Write this code as readable as you can. Also, see point 6.

10. Mobile responsiveness

CTRL+Shift+I, that's how easy it is for the reviewing developer to test the mobile responsiveness of your app. And nowadays it's an essential topic for companies to not upset the Google search engine. So be sure that your app is not totally broken on mobile devices.

11. Pin your GitHub repos

Assume that someone who wants to check out your skills might end up on your GitHub profile. The default order of the repositories there is by popularity. Which doesn't mean a lot when you don't have popular repos. But you can select which projects should appear in this list by clicking "customize your pins".

12. Don't use tutorial apps

Everybody watches tutorials and a lot of people implement these apps. Many people also list them in their portfolio. This makes it likely that the person reviewing your projects already saw the same thing over and over again and recognizes it as belonging to a tutorial.

Even worse, a lot of people don't say that they implemented this app with a tutorial. Don't do this. It feels like you're lying about your achievements and diminishes any trust in you.

After all, following a tutorial is relatively easy. Even if you customize the app afterward. So writing your projects from scratch is a better option.

13. Semi-unique app

Everybody builds ToDo apps, weather apps, or yet another YouToob. These all might prove that you have the required skills and even get you a job. But if there are lots of applicants they might not be enough.

For a reviewer, it's never clear how much of the code was your own doing and how much you copied from other sources you found on the internet (similar to the tip about tutorial apps).

Maybe you ask yourself: "How should I find a good idea?" One way of finding good ideas is to scratch your own itch. Imagine you like surfing. If you would like to know where the best surf spots are, just create a small app for this. It doesn't matter if such an app already exists. It's still way better than another ToDo app.

14. Write tests

Adding tests to your portfolio project is one of my favourite tips. It's not hard to do (after you get used to them) and it shows a great deal of maturity and professionalism. Especially self-taught developers often have no experience with testing.

How should you approach testing? An excellent library for testing frontend apps is testing-library. There are versions for all major frameworks available. Alternatively, you can write end-to-end tests for example with Cypress. If you're new to testing you can find an in-depth beginner's guide to testing React apps here.

If you write tests, make sure to mention that in the Readme and the project summary in your CV.

15. Good Git workflow

Using a good and common git workflow is another sign of professionalism. When I check repositories of job candidates I usually take a look at the commit history as well. Often you only see one "initial commit". Maybe a couple of bug fixes. That doesn't look very professional.

For your personal projects (which you probably host on GitHub) you can use a simple workflow with issues and PRs. Use expressive and clear branch names and commit messages. You can also squash merge the commits of one PR to keep the history clean.

Again, it might be a bit scary to work with Git in the beginning but you'll quickly get used to it. If you know the basics of Git already but aren't sure how to use it in a professional team yet here is a free and interactive course of mine that can help you out.

And once more: if you use a proper Git workflow mention it in the Readme.

Tips if you want to apply to your dream company

Maybe you have a particular company in mind that you'd like to apply to. It may be the company of your dreams. But for your first job it could just be any company that looks like a good fit.

The following two tips can highly increase your chances of being considered for a job:

- Build an app using the company's tech stack
- Implement a potential new feature for the company

Combining both of these tips show a great deal of dedication and willingness to walk the extra mile. You can also prove that you have the skills to be a valuable asset to this particular company.

If you don't know what tech stack the company uses, you can research the company's GitHub profile or find employees on LinkedIn and check their skills.

A friend of mine applied for his first developer job like this. He is a self-taught web dev who has studied architecture before. Nothing related to software development. Yet, he applied to exactly two jobs and got an offer.

This technique doesn't work well if you need to apply very fast. There might be a lot of competition for a job and if you apply after a couple of days you might not be considered anymore.

But if you can build a small application quickly or you send out a cold application without a public job offering this technique might work very well.

Wrapping it up

This was the fourth chapter. I hope you found some of the tips for optimizing your portfolio projects helpful. You received many smaller tips that you can quickly apply to your existing projects.

But if you want to increase your chances of getting a job drastically consider the two more advanced options

1. Write automated tests

2. Use a proper Git workflow

If you’re new to testing you can start with this beginner’s guide to testing React apps.

Chapter 5

The Interviews

In the last chapter, we had a look at ways to improve your portfolio projects. By following the advice you hopefully will soon get more invitations for interviews, the topic of this chapter.

Job interviews are probably the most frightening steps of the hiring process. You're never sure what to expect. Will you make a good impression? Will there be algorithm questions? Will you know all the answers to technical questions?

As you may remember there will be multiple interviews throughout a typical hiring process. Let's recall the different types:

1. Phone interview with HR or a recruiter

2. Phone or on-site interview with the hiring manager

3. Phone or on-site technical interview

For bigger companies, you might need to go through more stages. There might be multiple technical interviews with different teams. Sometimes you have to talk to people with different roles, like marketing, design, development, and product management. This might take a whole working day including team lunch.

My experience is that most companies will stick to a shorter process though.

Before the interview

Before the interview, you can do a couple of things to improve your chances of making a good impression.

1. Do your research

Find information about the company. Have a look at their website and try to understand what their product is doing.

Sometimes the application you're supposed to work on is hidden behind a login and you can't get access without paying money. Anyways, there should be marketing pages with information and maybe some screenshots. There might also be technical documentation.

See if the company has a GitHub account. Perhaps you can learn more about their tech stack or some open-source repositories.

Some companies have technical blogs to attract talent. These are wonderful opportunities to get a step ahead of other candidates since the developers themselves describe the inner workings of the company. They might give insights into tools they use, the tech stack, or the reasoning behind technical decisions.

In some cases, the HR person or recruiter will tell you the names of the interviewers on the next stage. Make sure to note these names and research them. Do they have a LinkedIn or Twitter account? Maybe you can find their personal blog which will give you insight into their thinking. Even simply familiarizing yourself with their picture will make you more comfortable during the first important moments of the interview.

2. You should be prepared to talk about your previous experience or your personal projects.

If you have professional experience as a software developer prepare a short story about the most interesting projects you were working on. What technologies did you use there? How big was the team and how did it collaborate? Did you implement something or did something happen that you are especially proud of? What did you learn?

Naturally, the interviewers will expect you to have deep knowledge of your personal projects. After all, you wrote the code. If it has been some time since you last touched your projects be sure to refresh your memory. Try to remember why you made a certain decision.

It can be especially helpful to find things that you would do differently from hindsight. This might be a question in the interview. **Even if they don't ask for it you can still bring it up.** Being able to criticize yourself shows character, maturity, and learning ability.

3. Practice interviews. Especially if you're a nervous type of person.

Common advice is to practice writing (pseudo) code on a whiteboard or paper. You can find tons of coding interview questions online. While you're thinking about the solution and writing down the code speak out loud and explain what you're trying to achieve.

Speaking out loud is very important here. Mostly when we code we solve problems in our head. Especially, when we're not used to pair-programming or using a rubber duck. Getting used to explaining our thought process to others will drastically increase your chances during interviews.

You can do the same with common interview questions that don't require you to code. Just find a list of, let's say, interview questions for JavaScript and answer them out loud. If you don't know the answer to all of the questions, even better. This will happen in a real interview anyways. So better to get used to it already.

Another technique you can use is to imagine an interview situation or replay a past interview in your head. Your psychology is a major factor that might help or prevent you from getting a job. Getting used to the feelings and the stress that may arise can help you to stay cool.

And, of course, some websites offer mock interviews. interviewing.io or pramp.com seem like good options. Also, have a look at codesignal since they have a lot of resources for interview practice. All these websites are free for developers since they make money by finding good candidates for companies.

The last tip about practicing interviews is to apply for jobs that you don't really want. You'll be much more relaxed than in an important interview. Still, you can gather experience in real interview situations.

The interviews

In general, you should always try to imagine having a normal conversation with another human being. Think of the interviewer as a nice person you have a nice chat with. Personality is important. So the interview shouldn't be a dry painful process. But this also depends on the interviewer, of course.

As we saw above there are three different interviews. Let's talk about each of them in detail.

1. Interview with HR or a recruiter

The interview with HR is usually a short phone call of about 15-30 minutes. The person on the other end of the line is usually non-technical. They won't ask deep technical questions but rather poke around a bit by asking about some buzzwords.

As a React developer, for example, you might be asked if you have experience with Redux. The HR person won't exactly know what Redux is. Just throw some terms back like "*Yeah, I'm familiar with writing reducers and actions*". Even if you don't have experience but know a bit about it you can say something like *"I haven't worked with it before but I know that it's used for state management in complex applications"*. This should be enough to make them happy in most cases.

The most important thing to remember is that you need to be friendly. These are the first gatekeepers on your way to a job. They have the power to reject your application. Even if they ask a question that seems stupid keep in mind that they are strangers to software development.

2. Interview with the hiring manager

As you may remember, the hiring manager is the person who decides whether you get the job or not. They often have technical knowledge. They might have been a developer themselves before. Maybe they even are the team lead who is looking for a new coworker.

This interview is typically another chat on the phone of around 30-60 minutes.

They will introduce you to the project and allow you to ask questions. Don't be surprised when they don't know the details of the team's tech stack or workflows. Depending on their role within the company this is totally normal.

You'll get a chance to talk about yourself and your projects. Having a short storyline prepared can be helpful at this point.

You might need to answer deeper technical questions. Buzzwords won't be enough here. At the same time, these questions won't be too hard yet.

In this interview, it's even more important to be nice. The interviewer might be someone you'll work with daily. So they'll be much more picky character-wise and they will make sure you're a good fit for the team.

3. The technical interview

The technical interview is mostly on-site. You'll often get to know a bunch of people and have a tour through the office.

This is also the most difficult part of your journey. The final boss.

Since this is such an important part let's have a whole section dedicated to it.

The technical interview

In my experience, there are different types of technical interviews with two extremes: one that feels like an interrogation and one that feels like a nice chat.

In any case, you will start with an introduction round where all the interviewers introduce themselves and their roles. Then you'll have a chance to tell them a bit about yourself. You can shortly say where you're from and maybe mention your favorite hobby. If you're lucky you have shared interests and can kill a couple of minutes talking about those.

Then you'll talk a bit about your experience or projects. Don't oversell your experience. Be honest with your strengths and weaknesses. And try not to start a long monologue when talking about your projects. If the interviewers seem to lose interest just ask them if you should go into more detail or if they heard enough.

At some point, the question-answer part starts. There are different kinds of questions which we will have a look at now.

1. Technical questions

Technical questions can be very specific knowledge questions, open-ended questions or algorithms, and data structure questions.

Examples for a specific knowledge question are *"How does inheritance work in JavaScript?"* or *"What kind of lifecycle methods do you know in React?"*. Basically, you know the answer or you don't.

If you don't know the answer at all just say *"I don't know"*. Nothing is worse than starting to guess. The interviewer knows the answer so they can see when you're bullshitting.

Actually, it's important to admit your knowledge gap. In a real team, you want to have people who are open and ask for help when needed. Somebody pretending to know is at least annoying and in the worst-case dangerous.

If you have a hunch about the answer but aren't sure you could answer the inheritance questions like *"I'm not sure but it has something to do with prototypes."* If you know the answer partially you can say *"Lifecycle methods? I know componentDidMount and componentDidUpdate. There are more but I can't remember right now."*

If you know how to find the solution that's also good to point out. *"I'd have to have a look at the documentation. There's a nice page about all available lifecycle methods if I remember correctly."*

Finding solutions to a problem and reading documentation will be a big part of your daily work. With this answer you at least let the interviewers know that you read the documentation before and that you know how to search for answers.

An example of an open-ended question is "You enter a URL into the browser and hit the return key. What happens?" In this case, the interviewer wants to see your thought process. So just start somewhere and don't be too surprised (like I was the first time I heard this question). It doesn't really matter how detailed your answer is. You could start with the browser sending a request through your machine's operating system which hits DNS servers… bla bla. You could even talk about electrons and electromagnetic waves. You will probably get follow-up questions during your answer.

Algorithm and data structure questions are the ones that are most prominent in forums and blog posts. For self-taught developers, these CS questions are often a horror scenario. I know they are for me. Especially when you're working with JavaScript you may never have heard of a linked list, a hash map, or bubble-sort. People are studying hard to prepare for these questions on leetcode or hackerrank.

In my experience, not so many companies ask these kinds of questions though. After all, you won't encounter such problems in your daily work in most companies.

Of course, if you want to work for Google, Facebook, Uber, etc. you should know that stuff. I was asked about linked lists at an interview with Zalando (big e-commerce player in Germany). If your interviewers worked at one of these companies before they might also be into these questions.

But for many mid-sized companies or startups that are not based in Silicon Valley, these questions are not relevant. I encountered one data structure question once at Zalando and one algorithm question with an interviewer who previously worked at Zalando. That's all.

You might not want to waste a lot of time on grinding algorithms and data structures on leetcode. If you're unsure whether or not you should prepare for this kind of question, you could even ask the HR people or the hiring manager directly how these interviews usually work. If you're in contact with an external recruiter, even better. They are very interested in getting you a job, after all. They'll be very open in sharing all the knowledge they have with you.

If you know the names of the interviewers upfront you can also check their work history. Did they work for one of the big tech companies before? If yes, you might expect some questions in this direction.

A lot of interviews that I attended recently were more and more relaxed in fact. There was often a very limited amount of technical questions and much more focus and team compatibility and the kind of questions we will talk about in a bit. Nobody will be able to assess your real technical capabilities in a 60-minute interview anyway. And interviewers are increasingly aware of that.

One last note about technical interviews: Some interviewers use technical questions to push you into a corner and see how you react. The sole purpose is to see how deep you can go and what you will do when you reach your limit. This is the kind of interview that feels like an interrogation.

This is a technique I learned from a previous coworker. It's not very nice since the interviewee most probably gets nervous and feels like they suck. It's still good to know about it since it might be easier for you to keep calm. Just don't get nervous and start making stuff up.

2. System design questions

System design questions are not related to the looks of an application but its system architecture. Basically, you're asked how you would build application XYZ including servers, databases, and so on. This kind of question is aiming to learn something about your thought process.

You don't need to implement anything or write pseudocode. It's more about what kind of technologies you would choose, what features you would implement, how you would model your data, and how the different parts of the system would interact with each other.

If you apply for a pure frontend position and you don't have experience with backend development or vice versa it should be fine if you design only part of the whole system. Nobody will expect an entry-level applicant to give a complete answer here.

In most cases, the interviewers will guide you. They might ask you to clarify or explain some thoughts in a bit more detail. If you don't know further you'll always free to ask them for guidance. Actually, that might be well received.

Let's have a look at an example for a design question:

"How would you build a food delivery app like Uber Eats?"

I would start by gathering some requirements. *A food delivery app has multiple sides. There are the customers who order food, there are restaurants, and the couriers who transport the food from the restaurant to the customer. I would maybe ask how many users/restaurants/couriers are to be expected. How many languages do we need to support?* And so on...

With these requirements, we can start to design: *As a frontend, I would at least expect a web app and mobile apps for the customers, a web app or mobile app for the restaurants and a mobile app for the couriers.*

The first question might come up: "What technologies would you use to build these apps?"

For the web apps, I would choose React, because I'm most experienced in it and it's still state of the art.

For the restaurants, I wouldn't build a separate mobile app at first since the app is probably not too complex. I would go with a … I don't remember the name. What is it called again when you can add a website to your smartphone's home screen? Ah yeah, progressive web app. At the same time, I'm not sure… Maybe I should ask some restaurants first what they prefer.

For the mobile customer and courier applications, I wouldn't build a native app. I also wouldn't go for React Native since I heard that it's hard to maintain. I would try to build a small prototype with Flutter and see how it goes. But I have no experience with mobile development, so don't take my word for it.

The next question: “What kind of features do you think are necessary to make the customer website work?”

…

I guess that’s enough. There is not a single correct answer to a system design question. The important part is to gather requirements first, ask for verification of these requirements and explain your decision-making process. If you’re stuck you can ask for the interviewer's opinion or let them clarify. This can lead to a small professional discussion.

3. Questions about teamwork, workflow, and soft skills

The last category basically contains everything else. Especially popular are questions about teamwork to understand what kind of a team player you are and about workflows like agile development and your opinions about it.

I would suggest you read specifically about agile development and Scrum as these are often listed as (optional) requirements. You don’t need to understand these topics fully. Most teams anyways don’t implement them by the book.

As the last point, I’d suggest you read a bit about Git and typical workflows with pull requests and feature/bugfix branches. Again, you don’t need to be experienced with this but it’s always good to be familiar with the terminology.

I won’t go into details about the answers. But just to give you some image of what might expect you here are some examples from interviews that I was involved in (in brackets I mention things that you might want to look up).

- How many people were in the biggest team you worked with?

- How would you solve a conflict in your team?

- Can you explain what agile development means? (waterfall vs agile, cross-functional teams, ...)

- Do you know what Scrum is? Have you ever worked in a Scrum environment?

- What does code quality mean for you? (maintainable code, readable code, ...)

- How do you achieve it? (Code reviews, good variable and function naming, tests, single responsibility, DRY principle...)

- What are code reviews good for? (Code quality, preventing bugs, knowledge sharing...)

After the interview

The interview is over. Finally! Your brain is fried and you're exhausted. But don't run off already. I have two more tips for you.

1. Ask questions

The interview is mostly finished with the following sentence: "Are there any questions from your side?"

A candidate asking good questions after (or during) the interview seems so much more interested in the company and the job they're applying to. But as I said, your brain is probably fried at this point. There might be no mental capacity left to think about anything interesting to ask.

That's why you should prepare some questions upfront and memorize them. Maybe even write them down on a piece of paper. There is no shame in reading your questions from a paper. It just shows that you prepared for the interview.

Here are some suggestions. You can ask

- about the company

- about the development process

- for a tour in the office

- what kind of technical debt they have

- what the best and worst things working there are

Especially the questions about the development process, technical debt, and best/worst things working there can be enlightening.

Specific things to ask about the development process are if they use agile methodologies like Scrum, if they work in sprints, if they work in cross-functional teams, and if they make use of pair programming. This way you can check if they work in a modern development environment. This is especially important if you're an entry-level developer and want to improve yourself.

The questions about technical debt and best/worst things are the counter-part to the famous question about the strength and weaknesses of a candidate. It can be very interesting to watch the interviewers when thinking about an answer.

2. Review the interview

After each interview write down everything you can remember as soon as possible. It's important not to underestimate the effect of this exercise. It will be so much easier to identify your weak spots and be much more prepared for the next job interview.

Try to recreate the interview step by step. What questions did they ask and what did you answer? Try to reflect on your mood as well. When were you nervous, when confident?

I remember a particular interview with a large tech company. An external recruiter had arranged this interview for me. As soon as I set foot outside the office building he called me. He was very interested in all the juicy details.

I told him that it went really well. At least that's what I believed at that time.

Then I started talking him through the interview. Step by step.

The first question was [some question]… Very easy. My answer was…

The second question was [another question]… Piece of cake…

The third question was [a simple question I didn't know the answer to]… Oh actually I didn't know the answer. Kind of embarrassing to be honest. I should look this up immediately.

It wasn't my day. I didn't get the job. But by stepping through the interview question by question I at least understood why.

Initially I had only remembered the good stuff. But talking the recruiter through every detail uncovered some questions I couldn't answer. I remembered situations where I was very uncomfortable and must have appeared very insecure.

Since you mostly won't get detailed feedback when being rejected this is the best way to be able to improve your interview skills.

Summary

This chapter about the interviews was split into three parts.

Before the interview you should prepare yourself by researching the company, preparing a storyline, and practicing interviews.

Types of interviews: An interview with HR, the hiring manager, and the technical interview.

Types of questions: technical questions, system design questions, and soft skill questions related to teamwork, workflows.

Types of technical questions: specific knowledge questions, open-ended questions, and algorithm/ data structure questions.

After the interview it's your turn to ask questions. Since you might be exhausted at this point it's best to prepare them upfront.

Finally, you learned about my personal technique: **the interview review**. You can use it to uncover strengths and weaknesses in your interviewing skills.

Now it's time to talk about coding challenges.

Chapter 6

The Coding Challenges

In the last chapters we already covered most parts of the hiring process: The resume, portfolio projects, and the interviews. There is only one piece missing to have a complete overview of the hiring process.

The coding challenges.

Coding challenges are a good way for companies to check the computer science knowledge and programming skills of their candidates. After all, a person can seem great on paper or during the interviews but still lack practical programming skills. How would one know only from a conversation?

Of course, coding challenges can't tell an employer everything about your skills. You might be nervous or may have a knowledge gap just at the wrong time. But they usually give good hints about your skill level.

The most popular kinds are algorithms and data structures tests, live coding sessions and take-home challenges. Let's talk about each of them in more detail.

1. Algorithms and data structures tests

These assessments seem to be most common when you apply for bigger tech companies like Facebook, Twitter or Uber. They are often one of the initial steps in the hiring process. Only if you pass these tests you will be invited to interviews.

So how do these assessments work? You usually receive an invitation for an online test that can be hosted at websites like HackerRank or LeetCode. Examples for such questions are "Remove duplicates from a sorted list" or "Find the longest substring without repeating characters".

Sounds scary?

Good news: Depending on where and what type of company you apply to you might never encounter such tests. At least I never did. So they may not be as common as suggested in many online communities.

In case you anyways want to practice for such a test just create an account at LeetCode or HackerRank and start grinding.

Just don't be discouraged if you cannot answer even easy tasks. If you can't find the solution look it up. No shame there. Being able to pass these tests is a LOT about practice. Especially self-taught developers may be able to build complex applications despite failing them.

2. Live coding sessions

Live coding sessions can be nerve-wracking. You're given a task and are asked to complete it in a relatively short time. Like an hour or so.

As a frontend developer, for example, you're often asked to implement a small application that fetches data from a given endpoint and renders it in some way.

But the worst thing is: in most cases, people are watching you.

Either you have to share your screen during a Skype call or you're coding in the company's office next to your future team members. In some lucky cases, you might be given some time to work on the task alone and present it directly afterward.

For a lot of people, this feels very intimidating. Especially, when you don't have a lot of professional experience. After all, we tend to code alone most of the time.

Luckily it's not all about speed and finishing all tasks. And, by the way, you're allowed to use Google.

Here are some of my war stories:

For a job at eBay, I was asked to fetch a list of cars from an endpoint. The cars should be rendered in a responsive grid with certain requirements for the styling. Each car should show a modal with some additional information upon being clicked. They gave me an hour to work on the project.

This time I was lucky. I was allowed to work by myself on my machine. Still, I was in their office. Working in a place unknown to you surrounded by strangers can be stressful enough.

I wasn't able to finish all the tasks in one hour. The grid wasn't very responsive and the modal didn't work completely either. Anyways the following interview went well. I was asked about my technical decisions and what I would have done differently with enough time. The app served as a guideline for the interview. It was totally OK not to finish everything.

Another time I had to implement a small app while sharing my screen during a Skype interview. I was caught off-guard since I didn’t expect a coding challenge at all.

At work I hadn't touched anything related to data fetching for a while. But I knew this could be done with React hooks. Hooks were new at that time and I had basically no experience working with them. Still I decided that it was a smart move to fetch the data with a useEffect. “It’ll look cool and impressive” I thought.

As a result, I wasn't even able to finish the data fetching part properly. Embarrassing!

But surprisingly, I got the job. Why?

This is the great advantage of live coding challenges: You can talk. And you should.

Live coding challenges are all about leading the interviewers through your thought process. Verbalize all the thoughts you have. Explain what you're intending to do. Tell them what the code you're writing at the moment is doing.

Naturally, you'll be much slower writing code. But there's nothing worse than starting to code quickly and lose the interviewers on the way. Might be impressive if you're doing everything right and implement the tasks in record time. But usually, you run into a problem at some point. Next, you get nervous because people are watching. And since they got lost on the way nobody will be able to assist you. Great opportunity to panic!

Being able to explain what you're doing has another advantage: **You show your communication skills. And as you know from job postings, everyone wants great communicators.**

So how can you practice live coding?

There are a couple of common tasks in most challenges: Setting up a project, styling, fetching data from an API, and implement state management.

Practice these tasks: Find a public API, initialize a project (for example with create-react-app), fetch some data and render it. Add some CSS.

Repeat until you're comfortable.

There are lots of public APIs, like weather APIs. Spotify, Etsy, and many others have APIs. You can find an extensive list to get creative here.

And don't forget: **talk out loud while implementing these practice projects**. Might feel stupid, but it will help you a lot to be prepared for a real live coding challenge.

3. Take-home challenges

Take-home challenges have one big advantage: you can work by yourself from a comfortable location, your home. They come closest to a real day at work.

The disadvantage is that these tasks are similar but more complex than the ones in live coding sessions. You're typically asked to spend at least two hours on a challenge. Sometimes even more. And often you'll end up spending more time than requested just to finish the last details and polish the code a bit.

This adds up. Especially, if you have multiple job applications ongoing.

Another disadvantage is that you cannot explain and defend your code in the same way you can in a live coding session. Thus you need to pay more attention to the details. Things to watch out for are:

- Make sure the app works
- Add clear instructions about how to install and run the app
- Take care of clean code formatting

If you want to stand out the best way, in my opinion, is to add tests. Most Junior developers don't know how to write tests. And almost no developers implement tests for code challenges.

Unfortunately, take-home challenges are a double-edged sword. Many developers really dislike them because they require an asymmetrical time investment. Companies can send out one challenge to dozens of candidates even before an interview. The candidate on the other hand has to spend hours on writing code without any promises.

That's why you can find many aggravating stories: developers spend hours on a challenge but never hear back from the company. Some coding challenges look suspiciously like new features for the companies. Sometimes the description of a challenge is so bad that it can only be misunderstood.

On the other hand, from the perspective of an employer it makes sense to test candidates for their coding skills. If the company is open to hiring Junior developers it may even be a requirement. After all, hiring inexperienced developers is like taking a bet on them. An employer has to invest a lot of resources into training new devs but don't know how they'll develop in the future.

So be careful and don't expect too much. At the same time, don't be too stingy with your time. If the potential job is worth it, you should consider investing a couple of hours. Especially if the challenge is reasonable.

Sometimes coding challenges can also be fun. You can learn something or use it as a project on your GitHub portfolio later.

Wrapping it up

In this lesson, we talked about coding challenges. The most important types are tests about algorithms and data structures, live coding sessions and take-home challenges.

Although there is a lot of talk about algorithms they might not be as common as you think.

Live coding sessions can be terrifying but have a lot of advantages.

Take-home challenges, in contrast, are much more relaxed but carry a higher risk of ending up a waste of your time.

Chapter 7

Changing Your Approach

In the last chapters, you already learned a lot about how to stand out during your job hunt by improving your resume and portfolio projects. Hopefully, everything goes well and you get some interviews and coding challenges soon.

If not it might be time to change your approach.

Many new developers search for opportunities on job boards like Glassdoor or Indeed. The problem is: everyone does this. That means that these job postings are often flooded with applicants.

Think of it as the front door to a club that is used by the party crowds. You need to wait in line and once you get to the door the bouncer may not even let you in.

Let's see how you can be smarter and get access via the backdoor.

1. Use your network

This sounds obvious. But make sure to use your network to its full extend. If you can get a referral for a job you'll have a great advantage. Not necessarily at getting the job but it'll be much easier to get invited to an interview.

So ask yourself: Do you know people who work in the software industry? If yes, contact them. Tell them you're looking for a job as a software dev. Maybe you're lucky and you get some insider information.

But what if you don't know anyone in tech? That was my situation a while back. Maybe it's yours now.

Think of it this way: you probably know some people, right? That means you have a network.

Ask your friends if they know a software developer or someone working in a tech company. Ask your mom, your uncle, your neighbor, There's a good chance that someone has a friend or family member who can help you.

Maybe you ask yourself: Why should somebody you barely know help you get a job?

A lot of people are just nice. At the same time, **many companies offer an employee referral bonus**. So this might be a win-win situation: a job for you, money for them.

2. Connect with real people after applying

Why not use the front and the backdoor at the same time? Once you sent out an application via a job board try to connect with a real person. You can use LinkedIn to find engineers working at the company you applied to. People are often very open to connecting with fellow software developers on LinkedIn. You can also find professional email addresses via hunter.io or with the Gmail plugin Clearbit Connect.

Tell them who you are and what you're doing. Mention that you applied for a certain position. If you email them you can attach your resume.

You can start by asking them about the application process and get valuable insight. Finally, you ask them if they or somebody in their team would be open to chatting about the position.

If they don't answer don't annoy them. Send at most one follow-up. If they answer, you can ask them if they would be willing to connect you with a hiring manager.

If they are nice you could even invite them for a coffee? Nothing better than connecting on a personal level.

3. Attend Meetups

Meetups are a great way to build a network. Just don't make the mistake of coming along as needy. If you're at a meetup for the first time don't ask around for jobs actively. Try to talk and engage with people.

If someone asks you what you're doing you can, for example, tell them that you're learning React and preparing a career transition into web development. Then you can ask if they have tips for you. But don't ask if they know about open positions at their workplace immediately.

If this feels uncomfortable remember: Many companies offer referral bonuses. Other developers might be happy to help you out.

The best thing is to visit a meetup regularly. After some time you will find out who other regulars are, who is involved in the community. People will start talking to you automatically.

This might sound terrifying, but consider speaking at a meetup. Even if you're not super-experienced. Talk about something you learned or some library you used. You could call it "X for beginners".

Reach out to the organizers before you prepare the talk. They can tell you if it's interesting for their audience. Often meetup organizers are desperate to find speakers for their events. So the chances are good that they'll be more than happy to accept your talk.

If you did a talk on a meetup mention it on your resume. You could link to your slides or a recorded video. If you're a regular on the meetup you can mention it as well. It will show dedication and eagerness to learn.

By the way, meetups are not only a great opportunity to find a job. They are also a chance to accelerate your learning.

Before I started attending meetups I was used to finding information on my own. Listening to developers who talk about technologies, libraries, or patterns they use at work was very helpful to get up to speed with professional tools. I rarely understood anything during the talks, but I wrote down keywords and looked them up at home.

4. Participate in discussions on Twitter

A lot of developers hang out on Twitter, you might have realized this. That makes Twitter a great place to get in contact with people that can help you get hired.

Again, don't participate in discussions with the purpose of asking for jobs. Ask genuine questions. Share blog posts that you liked and tag their authors. Give feedback. Follow interesting people. And include people that live and work in the area that you're interested in.

It's a long game but every person you had contact to might remember you. Sometimes developers share things they did at their company. This is always a good point to send them a comment or direct message.

Tell them their tweet was very interesting. Maybe remind them of an occasion where you had a conversation on Twitter. Start a conversation. At some point you can ask them politely if there might be an open position in their company. Again, remember the referral bonus.

Getting in touch with developers on Twitter might be easier than you think. Try to find accounts who don't have many followers yet but are active. Many experienced developers try to build an audience these days. It's much easier to get in touch with them compared to the popular people with 10k followers or more. Engage with them, comment and share their tweets and blog posts. You'll get noticed soon.

Since you're at it already, feel free to connect with me as well.

5. Attend Hackathons

Hackathons are events where a group of software developers, designers, and so on come together and work on a project for a couple of days, often a weekend. Hackathons mostly have a certain topic, are often hosted by a company, and the best groups might win a prize.

Hackathons can be a great way to gain experience working in a team on a real product. You can build a network and in some cases expose yourself to interesting companies and recruiters.

You can find hackathons for example on Devpost. If you participate in a hackathon your team can also share the project you worked on there. Again, mention the hackathon and the project on your resume pointing out that you worked as a team.

6. Job fairs for software developers

From time to time you can find job fairs for software developers where companies present themselves to attract new talents. These are great opportunities to **establish personal contact with these companies.** At the same time, there are lots of other young developers looking for jobs.

One technique you can use to **stand out** is basically the same as you would prepare for a job interview: **do your research.** Try to see if the companies at the job fair have a tech blog. Find blogs or Twitter accounts of the employees. Try to gather information about the tech stack or patterns they use. It's much greater to start a conversation with "I read this interesting thing on your blog. Can you explain that in a bit more detail?" **You can show your interest and social skills.**

If you're able to start a conversation ask them for their name. Connect with them on Twitter or LinkedIn. **You might have your first advocate on the inside.**

7. Find new sources for job postings

Don't restrict yourself to popular job boards. There are lots of places where you can find job opportunities. You can find startup jobs at AngelList. Examples for the React community are:

- The "Who's Hiring?" posts on the React subreddit
- The Reactiflux job board
- The Spectrum jobs channel

Similar pages should exist for other technologies as well.

Summary

If you still have problems finding a job even after you worked on your resume, your portfolio projects, and your interviewing skills you might want to change your approach.

A lot of the tips in this chapter require some level of socializing. It's not easy for many people to simply start talking to strangers at meetups, send them messages on LinkedIn or Twitter or speak in front of a crowd. But with practice comes mastery. That's also true for social skills. And once you put in the effort you'll be rewarded.

This was the **final chapter of this book**. I hope it helps you getting this daunted first job. I’d be happy if you’d connect with me on Twitter and let me know when you do.