

# The Essential Guide to Take-home Coding Challenges



by Jane Philipps

## *Introduction*

Hi, I'm Jane. I wrote this guide because I want to help others with non-traditional backgrounds succeed on take-home coding challenges. Please read it, take notes, apply the material, and let me know about your results. You can reach me via email at [jane@fullstackinterviewing.com](mailto:jane@fullstackinterviewing.com).

This guide is intended for anyone who has received a take-home coding challenge as part of the technical interview process and wants to attack it in the best way. This Essential Guide is a distilled version of a longer [Ultimate Guide to Take-home Coding Challenges](#), which goes into much more detail and walks through an example challenge from start to finish.

So, if you've just received a challenge and are anxious to get started, start here, and then check out the [full guide](#) when you want to learn the material more deeply. Good luck!

## Mistakes to avoid making when working on a take-home coding challenge

There are several mistakes you can make with take-home challenges. Some of these are small mistakes that are easily correctable, while others will leave you frustrated and unable to finish your assignment. I want to address these mistakes first, so when you're given a take-home challenge, you know exactly what not to do.

Here are four mistakes you can make:

- 1. Time management and scope creep**
- 2. Trying to learn too many new things at once**
- 3. Making too many assumptions**
- 4. Starting to code right away**

Let's look at each one in detail.

### **1. Time management and scope creep**

Time estimation is one of the hardest problems in programming, and even experienced engineers struggle with it. This plays into take-home challenges in a couple of ways.

First, some challenges come with "estimated time." I usually ignore these, as they are rarely based in reality. Second, some challenges are open-ended. Many people, especially newer developers, will want to add tons of features because they think it will be impressive. Actually, it's more impressive if you keep the scope relatively narrow, but finish everything you set out to do. In this situation, it's better to do one thing really well than to do a million things poorly.

A good question would be: what counts as "going above and beyond" versus what counts as "scope creep?" My rule of thumb would be if your idea accomplishes or improves on the

requirements of the assignment, that is likely a good idea, but if it seems tangentially related or “just cool,” it’s probably scope creep. But, as I describe later, always make it work first.

## **2. Trying to learn too many new things at once**

While a take-home coding challenge can be an excellent opportunity for learning, it is possible to take on **too much** learning. If you’re given a challenge where you must use a specific language or framework, but you’re not familiar with it, don’t add additional complexity by setting out to learn something new on top of that. For example, if you are using a new backend framework for a full stack app, stick to a frontend framework that you’re already comfortable with.

If your challenge is language/framework agnostic, but you’ve been itching to try out some new technology, pick JUST ONE to experiment with. Between reading the docs, getting your challenge properly set up, and getting used to any new syntax, you will have your hands full. Even learning one thing will eat up a lot of your time, so I would highly suggest limiting yourself to one new piece of technology per challenge.

## **3. Making too many assumptions**

As a developer, if you make too many assumptions, you are bound to build an application where the requirements are off, or the user experience is bad. When given a set of requirements for a take-home challenge, ALWAYS take the time to review the requirements and make sure you fully understand them. And, if you have any questions at all, always ask.

First, this shows that you are willing to ask for help when you don’t quite understand something, an important trait for a developer to demonstrate. Second, many companies will intentionally give you product requirements that are vague or not fully fleshed out in order to see how you react in these situations. They are actually testing your ability to make sense of requirements that may have gaps in them.

So, when in doubt, ask questions. Asking questions is also a signal that you are engaged and interested in the challenge.

#### **4. Starting to code right away**

One last mistake you can make is to jump in and start coding right away. I guarantee if you do this, you will regret it. Why? Two reasons:

##### **Without proper planning, your code will suffer**

Without first getting organized and making sure you fully understand ALL of the technical requirements, you may find yourself missing edge cases or rewriting parts of the functionality. I know it seems counter-intuitive, but you will actually SAVE yourself time if you plan ahead.

##### **You will spin your wheels trying to get your app set up properly**

Especially for newer developers, initial app setup can be one of the hardest parts of a take-home coding challenge. It's not something you do every day, so it often takes some research and reading documentation to get reacquainted with the process and ensure you're going about it in the best way.

So, there you have it — a summary of mistakes to avoid making. You'll find that a lot of these are also applicable to your day to day work as a developer. In the next section, we'll dive into further detail on how to get organized before you write a single line of code.

## **Get organized: how to plan before you write a line of code**

Now it's time to get to work! But, it's NOT time to write any code YET.

Why?

Because, as you'll see, a lot of the work actually happens before you write a single line of code. This may seem counterintuitive, but again — the more time you spend up front planning, the less time you will spend writing code.

So, now you have your coding challenge in hand and you are ready to get started with the planning process. Here are my six suggested steps:

- 1. Understand the requirements and ask any questions**
- 2. Identify technical decisions you need to make**
- 3. Technical design & whiteboarding**
- 4. Test plan**
- 5. App setup plan**
- 6. Organize your tasks**

#### **1. Understand the requirements and ask any questions**

First, you need to make sure you completely, absolutely, 100% understand the requirements of the project. If any part of the requirements are unclear, it is up to you to reach out to your contact and ask questions.

Sometimes companies will purposefully make their requirements vague, in order to see how you approach the problem. In these cases, it is always best to ask questions as it shows you are thinking about the problem and not just making assumptions and building an app to a vague spec.

#### **2. Identify technical decisions you need to make**

Your next step will be to identify the technical decisions that you need to make. Making a list of all of your technical decisions up front and thinking about them before you're in the middle of building your app will help you immensely. Not only will it cut down on time figuring things out later, but it will allow you to make big picture decisions up front, as opposed to trying to focus on both the big picture and the small details at the same time.

#### **3. Technical design & whiteboarding**

Now it's time to plan out the rest of your app. For anything that you need to draw out, now is the perfect time to do that. Thinking through these decisions at the start serves two purposes:

- You'll be able to reference these drawings and your original plan while you're building your app. Then if you get stuck at any point, you can always come back to your notes.
- Later, when you are having a discussion with an engineer about your coding challenge, you can use these notes as a reference when they ask you why you made certain design or architecture decisions.

Once you've thought through and answered some of the bigger design and architecture questions for your challenge, the next step is research. If you're planning to use a new technology or something you're a bit rusty with, use this time to search for documentation and other resources.

#### **4. Test plan**

Another very important step to take before writing a line of code is developing a test plan. Although you won't get peer feedback on this test plan, it will help you look at the challenge from a different angle, making sure you're meeting all of the requirements. By thinking through and writing out a test plan before you start coding, you are able to brainstorm possible edge cases that you should account for in your code and you will use this as a basis for testing your app later.

#### **5. App setup plan**

If you're starting an app from scratch, figure out if there are any generators you can use to make your app setup easier and faster. Application setup is one of the hardest parts of take-home coding challenges, because it's something that developers do rather infrequently. Best practices are always changing, so it's easy to forget how to do. Also, when setting up an app with a specific combination of technologies for the first time, it can be challenging to get everything configured and working together properly.

If you are not using a generator, reading documentation and finding working examples are the two most important steps you can take. Being able to play with a working example and compare it to your own app will help you if you get stuck.

## **6. Organize your tasks**

The last step before you start coding is to break down and organize your tasks. Breaking down your tasks is essential because it will help you stay on track as you're working on your challenge, and it will give you a game plan for execution. Note that you shouldn't be a perfectionist here, because there will always be unexpected bumps in the road.

Here is an example task list for a classic Tic Tac Toe app:

- Understand requirements- Choose technologies- Brainstorm test plan- Hello World app setup- Build board with HTML/CSS- Implement Tic Tac Toe gameplay with Javascript- Add reset button- Make board responsive- Add ability to add additional boards- Error handling & tests- Code cleanup- README

Some of these tasks can be broken down even further into smaller steps. For example, in order to implement the Tic Tac Toe gameplay with Javascript, here are some smaller tasks:

- Add a click handler to each square that logs a message- Get click handler to add an X to the square that is clicked- Get clicks to alternate between X and O- Don't allow a square to be clicked more than once- Implement a function to find the winner and end the game- Handle a tie game

## ***3. Writing tests: just do it!***

Testing can be overwhelming, because there are so many different types of tests: acceptance tests, integration tests, and unit tests, not to mention test driven development vs. ad hoc testing.

Why should you include tests in your take-home coding challenge? It's simple: your tests will make your submission shine.

First, adding tests shows that you know or are willing to learn another technology/framework. It also demonstrates that you take ownership of what you're building, because you are taking



responsibility to make sure it works. Testing also shows that you've considered edge cases, which many newer engineers often overlook.

Many companies take tests very seriously. Some will not tell you that they expect tests for your coding challenge, but will automatically reject you if you leave them out. Therefore, my recommendation is to write tests no matter what when given a take-home challenge. Not only will it make you a better developer, but for companies that were not expecting tests, you will stand out even more!

How do you go about writing a tests? First, create a plan. Here's my 80/20 suggestion for how to come up with the right test cases:

### **1. Test the happy path**

For the classic Tic Tac Toe example, the happy path is starting with an empty board and playing a game until X wins.

### **2. Think about variations on the happy path**

A variation on the happy path would be if O wins, or if there is a tie game.

### **3. Think of edge cases**

An edge case would be if a player tries to play a move in the same square more than once.

### **4. Test anything that is complex**

The algorithm to find the winner is the most complex part of this example.

Here's a sample test plan:

- Test that the initial state of the board is correct (i.e. board is visible and empty)
- Test that a move can be played
- Test that moves alternate between X and O
- Test that a move can be played to a square only once
- Test that a winner can be found in a row
- Test that a winner can be found in a column
- Test that a winner can be found in a diagonal
- Test that a draw can be found



So, now it's your turn. Think about your app and, as a baseline, think of 5–10 tests that you can write.

#### *4. Make it work, then make it pretty, then make it fast*

The title of this section sums it up pretty well, but when you're working on building out your challenge, you should follow these 3 steps IN THIS ORDER:

##### **1. Make it work**

##### **2. Make it pretty**

##### **3. Make it fast**

##### **1. Make it work**

**When you're given a take-home coding challenge, no matter what you do, the most crucial part of the challenge is to make it work.** If you submit an app that has a nice UI, that will not matter if your app does not work or meet all of the requirements. Because building features to spec is a key aspect of your future job as a developer, you first and foremost need to focus on the functionality of your app and prioritize that above all else. This is also key if you are low on or run out of time. Coding challenges can be **a lot** of work, especially if you want to go above and beyond to ensure that you make it to the next interview round. But, I can guarantee that you will not make it to the next round if your app doesn't function properly or is missing some key components.

So, if you're building a front-end app, this means focusing on making it work first, and styling/UI last. If you are building a back-end or full-stack app, focus on making it work before trying to refactor your code into the most elegant solution, and only then worry about optimization.

**Even if you end up without any time to go back and refactor your code or style your UI, having a working app to present is more important.** You can always talk to the interviewer about how you would improve your app, and refactoring some of your code might even be part of the next round of interviewing.

##### **2. Make it pretty**

Make it pretty has two interpretations here. One is making the code pretty, and the other is making the UI pretty. Making the code pretty can be done in several ways. First, ensure indentation is consistent and your code is readable. Second, if you got something to work in a quick, hacky way, think about how you can refactor it to be a more elegant solution without overcomplicating it.

If you're doing a front-end or full-stack challenge, you can also make the UI pretty as part of this step. Whether you use a library or write your own custom styles for your app, making the UI look good will show your interviewer that you're taking the user experience into consideration when building a feature.

For some more front-end-focused challenges, you'll be given a specific mockup to match. In these cases, making sure you're detail oriented down to the last pixel is incredibly important. Part of your role may involve translating mockups from designers into user interfaces, so companies want to get a sense of how you approach those types of tasks.

### **3. Make it fast**

Once you've made your app work, made it pretty (in the code, UI, or both), it may be time to make it fast! This is where understanding performance and BigO notation comes in handy.

You should take a look at your code and see if there are any areas where increasing the scale might be an issue. For example, are you using a double for loop somewhere? What if the arrays you're looping over become super long?

If you think about these kinds of edge cases, you can then come up with plan to improve your code. Taking something that would have been running  $O(n)$  and making it  $O(1)$  will show that you're thinking about performance when you're building things.

## How to make your code shine

When given a take-home coding challenge, many people think about how to build an app that works, but stop there. In this section, I'll go over things an engineer reviewing your code will look for, so you can take your challenge to the next level and make your code shine.

When an engineer is reviewing your code, they will look for several different things. They will likely try to run your app to play around with it and see it working. After that, they will delve into the actual code, looking to see how you organized your app architecture and reading code in individual files.

There are several things you can do to make your code stand out. You want your code to be:

- Readable
- Easy to follow
- Well organized
- Clean (properly indented, free of syntax errors and unnecessary whitespace)

These are the basics that don't take much effort outside of mindfulness to get right. Now let's talk about three of the more involved code style considerations:

### **1. How to name things**

### **2. How to use comments effectively**

### **3. How to format your code as you write it**

#### **1. How to name things**

Naming is one of the hardest problems in programming. One of the keys to naming things is to make sure you're naming them in a way that another developer who is unfamiliar with the code can easily jump in and understand.

For functions, think about what exactly the function is doing. Is the function checking whether there is a winner on a row of a Tic

Tac Toe board? Then a great name would be `checkRow`. Is your function handling a click on a square of the Tic Tac Toe board? Then a great name would be `handleClick`.

One quick tip: if you find yourself losing your flow because you keep stopping to think of the perfect name, split your process into two steps. First, write working code with any names (like `foo`, `bar`, and `baz`). Then take a second pass through to improve them.

## **2. How to use comments effectively**

Adding comments can be a great way to capture what you were thinking at the time you wrote a specific piece of code. This can be useful to you, or anyone else who comes across your code in the future and needs to understand it, tweak it, or rewrite it.

Think of comments as adding clarity to your code. But, pay attention, because there is such a thing as too many comments.

Here is where you most likely do not need comments:

- When you declare a variable
- When you declare a function

Don't do this:

The variable or function name should be enough to explain exactly what it does. If you need a comment to explain it, then you need to give it a better name!

Here are some examples of where comments can be useful:

- HTML
- CSS
- Technically tricky lines of code

First, let's talk about HTML. Markup seems pretty self-explanatory, right? So, why would you need comments? Let's say you have a really long HTML file with A LOT of `<div>`s. Comments can be a good way to signal which `</div>` tags close which sections.

In CSS, comments are a good way to divide up your styles if you have a lot of styles in one file. This way, when you come back to the code later and want to make a change, it's easier to find the styles for that one section you need to update.

Comments in CSS are also very useful whenever you are hard-coding any math or adding an arbitrary number of pixels as margin, padding, and so on. Comments can be useful to explain things like this that are specific to your application.

One of the best uses for comments is when you've written code that is technically difficult or just not intuitive. You should always strive for simple, understandable code as much as possible. However, sometimes you will have confusing code — maybe you've chained a bunch of methods together or are using a complex regular expression — and it would help to explain what is happening in a comment.

You are almost done learning how to make your code shine! Just one more step.

### **3. How to format your code as you write it**

I'm a STICKLER about formatting when it comes to code. And, it's not just me. You'll find that the best engineers also care about well-formatted, clean code. Why? First, it's much easier to read! Coding can be really challenging, so when code is easier to read, it makes our jobs as developers that much easier. Also, writing clean code sends a message to your interviewers that you take pride in the craft of writing code, and for many teams, this is a big deal.

So, how do you make sure the code style sticklers will approve of your code? There are a few simple tricks you can use as you're working through your coding challenge to ensure the end result comes out clean and you don't have to spend time at the end reformatting everything.

- Choose tabs or spaces and be consistent across your entire application (i.e. no 2 spaces in some files, 4 spaces in others)

- Indent your code properly as you go so that it stays readable and isn't all over the place
- Get rid of trailing whitespace! Whitespace can sometimes wreck havoc, so it's best to just get rid of it as you write your code.
- Keep your syntax consistent throughout your entire app. If you're using a linter, this will be easier, but requires setting one up. If you don't have time to set one up, pay attention. Don't use ES5 in some places in your app and ES6 in others. Pick one and stick with it!
- Remove unnecessary logging and debug statements when you're done using them! Unless logging is part of your application, you'll want to remove any temporary statements you were using while building your app.
- Always leave a newline at the end of every file

That's it! It's pretty simple, and once you're in the habit of doing this, not only will your code be easier for you to read, but it will also be easier for others to read and maintain. Many new developers haven't been exposed to very much code maintenance, but trust me, when you have to clean up code someone else has written, you will be more thankful if it was neatly organized to start. Pay it forward!

Here's an example of badly formatted code:

Here's an example of the same code, but cleanly formatted and MUCH more readable:

## How to take your challenge to the next level

Here are 3 ideas for how you can take your coding challenge to the next level:

- 1. Bonuses**
  - 2. UI/UX design (for front-end or full-stack challenges)**
  - 3. Data validation and error handling**
- 1. Bonuses**

Not all coding challenges come with bonuses, but if yours does and your goal is to get a job offer, do them! Why? It's pretty simple. If you go above and beyond in your coding challenge, it will show that you will go above and beyond once you're hired at this company. Completing bonus requirements is a high competence trigger for the interviewer.

## **2. UI/UX design (for front-end or full-stack challenges)**

Some front-end or full-stack challenges will mention UI/UX design as a bonus, but if they don't, putting in some effort to make the UI look nice and be easy to use will go a long way. You can either go the route of adding your own custom CSS or plugging in a library or two to help make your styling even more painless. If you use a library, just make sure that you understand how it works enough to explain how you've used it.

## **3. Data validation and error handling**

Data validation and error handling are key components in production apps. Adding either one of these (or both!) to your challenge will help make it stand out. Many developers who are new to coding and haven't worked in a production codebase before don't have a ton of exposure to either of these, so if you add error handling for edge cases it will show that you thought through a lot of different situations.

## **How to write an awesome README**

You may be done writing code, but you're not done writing yet — it's time to write your README.

### **Why you should include a README**

READMEs are incredibly important, both for professional developers and for job seekers working on take-home challenges. Including a README shows that you care about documentation.

Documentation helps spread knowledge across teams and serves as a supplement to your code. Having documentation for your take-home challenge ensures that anyone else (or future you) can



jump into your code with a clear understanding of what you've built without any guessing games.

Your README is also the KEY to making sure that everyone reviewing your challenge has the most painless experience possible. Finally, your README is a way of proving to your reviewer that you successfully met the requirements of the challenge.

## **How to write your README**

Writing a great README is not hard, and you will stand out a great deal from the other applicants with one. Here are the five sections I'd recommend you include:

- 1. Installation instructions**
- 2. Discussion of technologies used**
- 3. A section demonstrating that you met the requirements**
- 4. If there are bonuses, a section demonstrating that you met them**
- 5. For algorithms and data structures, time and space complexity**

### **1. Installation instructions**

When writing your README, don't make any assumptions. Write out all of the steps to run your app locally and test them yourself. This includes cloning the repo from Github, running installation commands, and starting up a server. Also, make sure to include versions of software that you are using. This will ensure that the developer reviewing your code has a seamless experience setting up and running your app, and if they do happen to run into any trouble due to versioning, they will have all of the information they need right there in the README.

### **2. Discussion of technologies used**

This section is as simple as it sounds — make a list of all of the technologies you used including frameworks and libraries. If you had to find a library for a specific piece of functionality in your

take-home challenge, mention it here and include a link to the docs.

### **3. A section demonstrating that you met the requirements**

Usually your take-home challenge will come with some sort of requirements spec, so make sure to include a section in your README where you describe the requirements and how you met them. In some cases, you can take the product spec you were given and write a short explanation of how you met each requirement in a list. In other cases, you can simply include a short paragraph explaining how you satisfied the requirements. It's totally up to you how you do it, just make sure you include it.

### **4. If there are bonuses, a section demonstrating that you met them**

Similar to the requirements section above, you'll want to highlight any bonuses you completed while working on the take-home challenge. If you attempted a bonus, but couldn't quite get something to work, then the README is also a good place to address that. You can discuss the approach or approaches you tried and what worked or didn't work.

### **5. For algorithms and data structures, time and space complexity**

If you had to write any algorithms or data structures as part of your take-home challenge, it's helpful to include the space-time complexity of your final algorithm. This can be done in Big O notation.

One final word of advice: write your README in markdown so it looks nice! This will demonstrate that you know (or are willing to learn) another language that will come in handy as a full-time developer.

**Here is an example README for a Tic Tac Toe app:**

## Final steps before you hit send

Now that you've written your README, you're almost ready to hit send! Before you do that, take the time to double check all of your work using the following checklist:

- Re-read the take-home challenge instructions to make sure you didn't miss any requirements
- Review your app's code to ensure that it shines
- Run your app's automated tests and make sure they are all passing
- Test your app manually and make sure everything is working properly
- Test your app installation instructions from your README
- Start an email draft and copy your README into it for convenience
- If requested, make sure to attach a zip file of your code
- Write an email to your contact at the company

Your email can be short and sweet — I always like to highlight something I enjoyed about the challenge or something I learned. Here's an example:

Hi <NAME>,

I hope you had a great week! I had fun diving back into React with this challenge. Here is my Github repo and I've included my README below. Please let me know if you have any questions.

Just so you know, I'm interviewing with a few other companies and I just received an offer yesterday – I need to get back to them next week. Of course, I am excited about the opportunity at <COMPANY NAME>, so I'm looking forward to hearing from you!

Thanks, <NAME>

Note that you should only mention interviewing with other companies or offer deadlines if either is actually the case. I feel you should be honest and candid about your situation and maintain leverage for a potential future compensation negotiation at the same time.

Now, finally, hit send!

## Conclusion

I hope this Essential Guide was helpful and you learned something that you can apply to a take-home challenge or in your day-to-day work. If you have any comments, questions, or other feedback, please don't hesitate to reach out. You can reach me at [jane@fullstackinterviewing.com](mailto:jane@fullstackinterviewing.com).


Also, if you enjoyed this guide and want to learn more, feel free to sign up for my email list:

**Want to CRUSH your next take-home challenge?**


**Sign up to receive your FREE PDF of The Ultimate Guide to Take-home Coding Challenges PLUS a special bonus on how to ACE the first part of every interview: the non-technical phone screen.**

\* indicates required


**Email Address \***



**First Name \***



**Last Name \***



**Subscribe to list**