

# Teaching code as a foreign language

Sophie Koonin

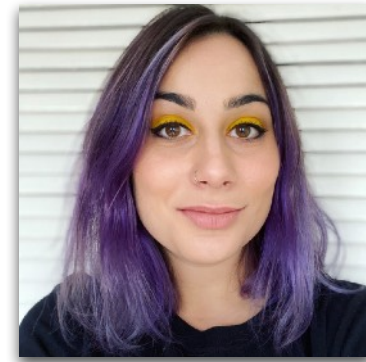
# A little about me

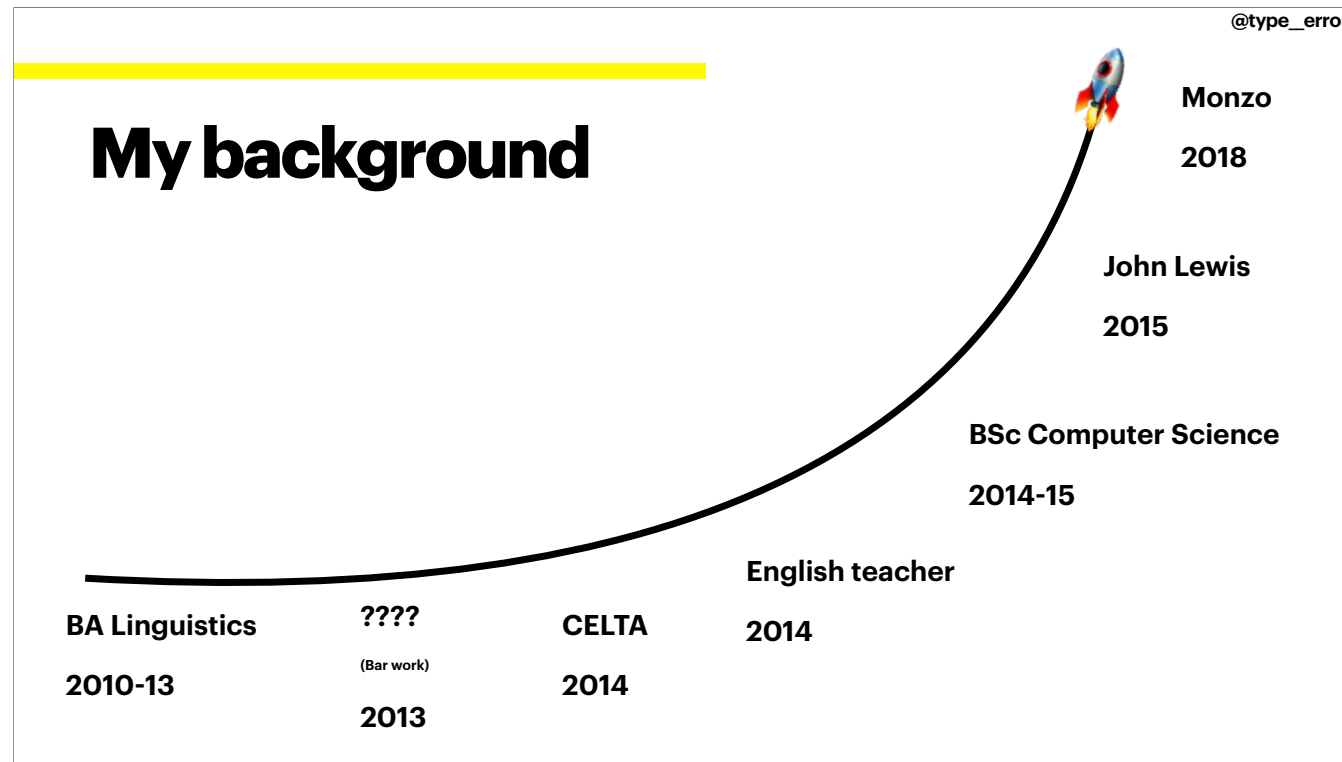
Senior web engineer & Web Discipline lead  monzo

@type\_error

localhost.dev

sophie@localhost.dev





I studied linguistics at university, and graduated in 2013 with no idea what I was going to do with my life. I ended up training as an English teacher and teaching English as a foreign language until I started a masters in Computer Science and became a developer.

# Teaching skills are universal

Once I started a full-time job in tech I was amazed at how transferable my teaching skills were. In the Before Times, when events were a thing, I did a lot of mentoring with kids and adults learning to code, and those skills really came in useful. And further down the line, I'm mentoring junior developers on my team and writing documentation for my organisation, and once again I'm putting those teaching skills to good use.

# Communication skills are essential

And as developers, it's vital we have good communication skills. It's not enough to just put our heads down and write code all day.

We need to be able to deal with stakeholders, clients, team members. We need to be able to get new joiners up to speed, and communicate how the project is going to tech leads and product managers. We need to be able to write a JIRA ticket that actually makes sense. We need to be able to explain why something has gone wrong, and what we're going to do to fix it, to someone who has never touched a line of code.

The skills that I learnt as a teacher set me up to be able to communicate well and command a room. And while the classroom-management side of things helps more when I'm leading a choir, the other skills are indispensable and I'm going to teach you some of those in this talk.

# **human language vs programming language**

I should clarify: I'm not saying that code is the same as human language. There are many similarities, and many differences. You certainly can't hold a conversation in JavaScript (please don't try), and you can't write a program in a sentence of English - although the absolutely cursed invention that is GPT-3 might change that. But human language has things like intonation, pronunciation, irony and socio-political associations that programming languages don't. Believe me, if it was possible to write sarcastic Javascript, I would never do anything else.

But learning a programming language does have its similarities to learning a foreign language.

@type\_error

# Lexicon

she	const
cat	function
read	instanceof
think	yield
my	switch
computer	true

Both human and programming languages have a lexicon - the set of words in a language. Human languages tend to have huge numbers of words that change over time, whereas the lexicon of a programming language is fairly set. But there are still words you need to learn when you're learning JavaScript.

@type\_error

# Grammar

I code, you code, she codes	<pre>if (thing != null) {     doSomething(); }</pre>
I must have been asleep.	<pre>func someFunction(string str) *Thing {     ... }</pre>
If I were you, I wouldn't do that.	<pre>Person pers1 = new Person();</pre>

And both types of language here also have a grammar. A grammar is a set of rules that defines how the language behaves and how the different words and phrases go together to produce meaning.

In English, we string together words in a particular way to indicate who the actor in a sentence was, to convey something that happened in the past, or to express a hypothetical condition.

In programming languages we string together keywords to define functions, conditional blocks and instantiate new variables.

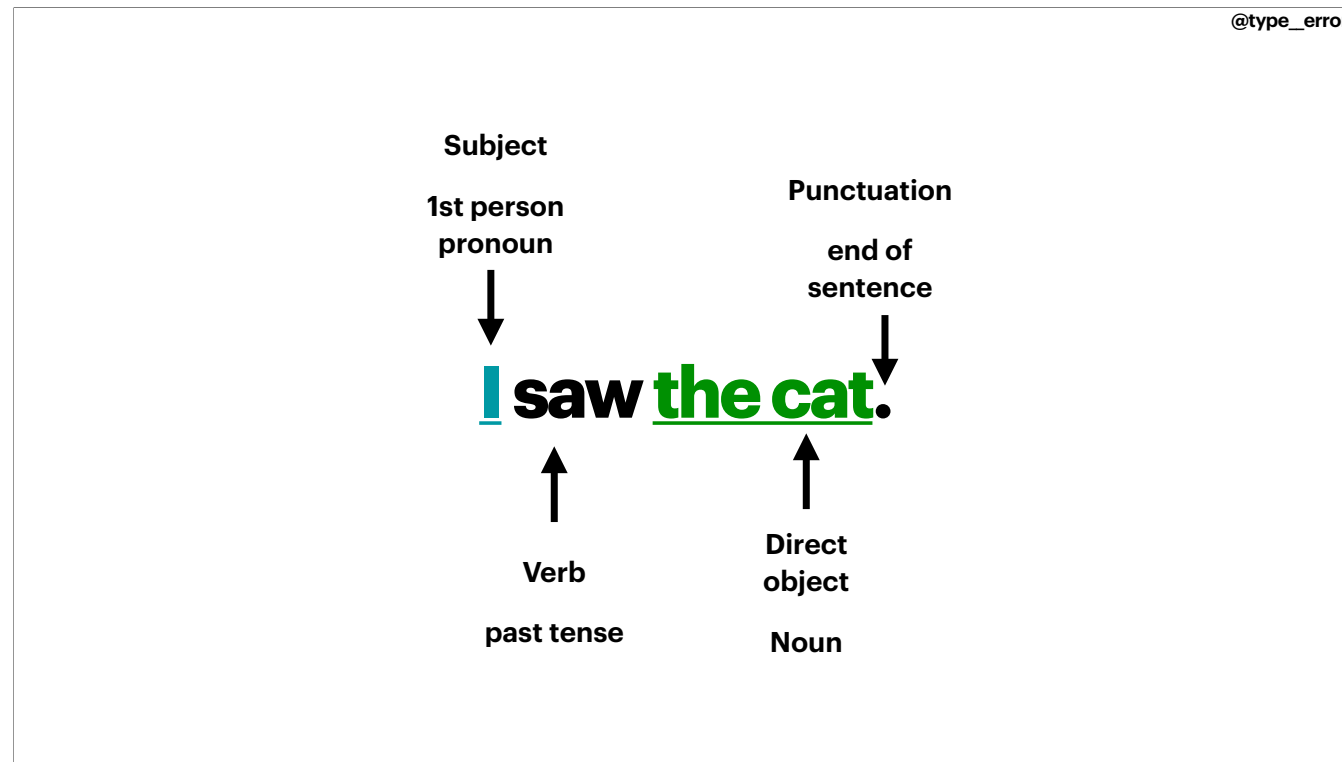
We understand meaning from the order of the words, and so does a compiler.



@type\_error

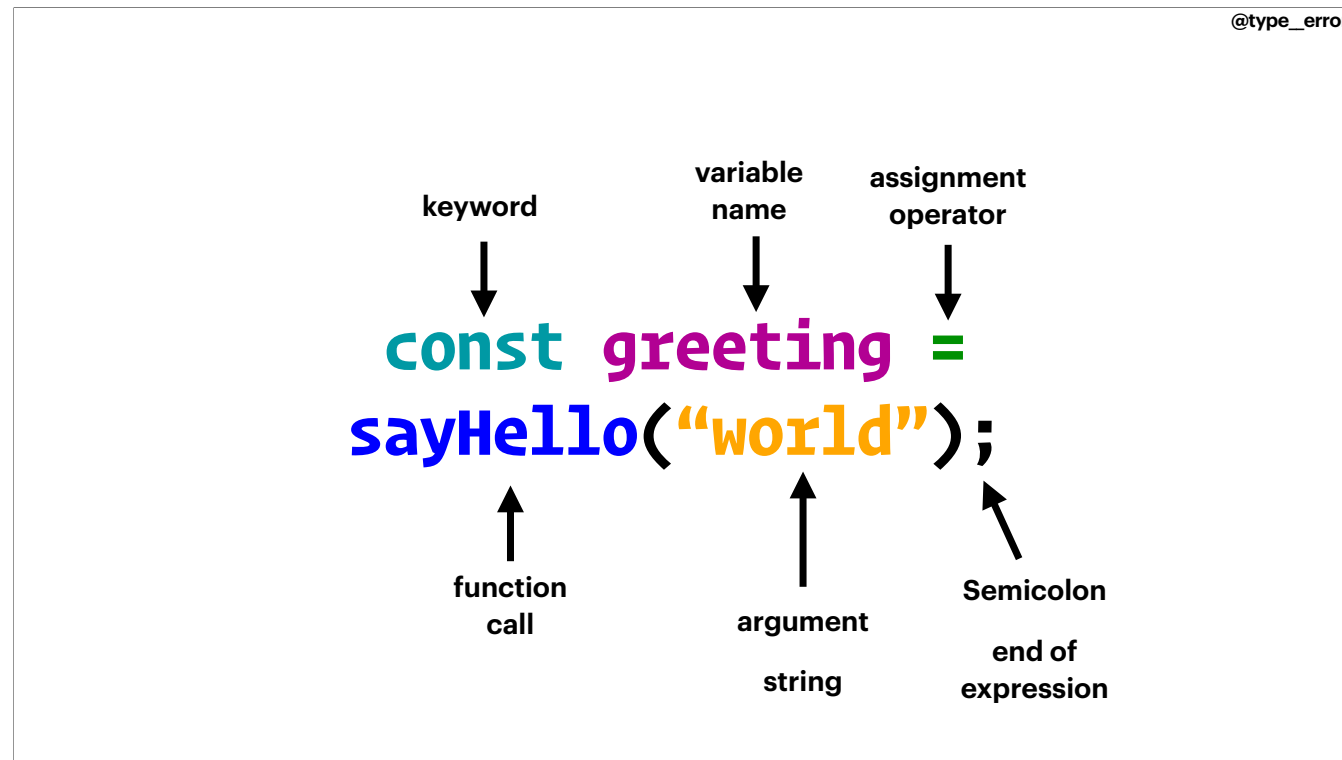
**I saw the cat.**

Take this simple English sentence



We can break up the syntax of this sentence into its various parts. In English, the subject usually comes before the verb, and the object of the sentence - that is, the thing being seen in this case - comes afterwards.

I'm not going to go too deep into this for obvious reasons, I literally took several entire modules on this at university.



But we can do the same to a Javascript expression. We can identify the different parts of the expression, and if we put them in the wrong order, the interpreter will throw a syntax error.

If you were speaking to someone who suddenly started speaking with words in the wrong order, your brain would also probably throw a syntax error as you tried to work out what they were talking about.

## **What can we learn from this?**

So we can see there are definitely some similarities between learning a human language and a programming language. You have to get to grips with a new syntax, and possibly even a new writing system if your native language isn't written in the Latin alphabet and you're not massively familiar with English. You've got to learn all these new words, and what they mean - and if you're learning another programming language you might end up translating in your head to other programming languages to help you make sense of it.

So it makes sense that we can apply some of the same concepts from teaching English to learning to code, and learning technical concepts in general.



# Mentoring others

## Writing documentation

The two areas of software development I'm going to focus on in this talk are mentoring others and writing documentation. Not the hands-on, super technical side of software development, but vital and often overlooked parts of being a well-rounded developer. Often called “soft” skills, which I absolutely hate, because they're really important and the word “soft” unfortunately has a negative connotation or dismissed as a “feminine” skill.

We may have to mentor others as a tech lead, to onboard a new hire, to teach your kid to code or to show your mum how to edit markdown so that she doesn't keep emailing to ask you to make copy changes on her website.

We write documentation when we hand over a project, when we build a tool or piece of software for public consumption or want to educate people about something.

**How teaching skills can help with...**

**Mentoring others**

So let's take a look at how we can learn from teaching when it comes to mentoring other people.

## What constitutes mentoring?

- Formal sessions e.g. CodeFirst: Girls
- Showing a new hire the ropes
- Helping a junior developer level up



I'm using the term mentoring very generally here - it doesn't have to be a formal arrangement, though things like CodeFirst: Girls and Codebar are great examples of that. It could also be onboarding someone new into the team, or taking a more junior developer under your wing to help them learn the skills they need to progress.

And mentoring is challenging. You need to get knowledge out of your brain into someone else's - and it's not necessarily as simple as just telling them stuff.

When you think about what we do, and everything that's involved with building websites and applications, it can be overwhelming for mentors to even know where to start, let alone the people they're teaching. So it's really important for you to let your mentee guide you. It's a two-way relationship, and you need to know why your mentee is there in the first place.

# Learner goals

Why are they learning the language?

Fluency

IELTS

Getting by on holiday



For my English students, there were a few reasons they would be having lessons. Some of them wanted to become fluent, and some needed to take IELTS, an English language qualification in order to get into a university or for work. Some were just looking to improve their conversational English so they could get around on holiday. These goals influence the kind of thing they need to learn, as well as their approach to learning and even their enthusiasm for the course.

There were also some students who had clearly just been sent to the school by their parents, and really didn't want to be there. And they were a lot harder to teach, because they didn't engage. With the exception of maybe the odd kid at a coding workshop, you probably won't encounter this. I hope. I once had a kid in my class who I caught listening to music in the middle of a lesson, and the only time he ever engaged with a lesson was when we taught them swear words.



# Learner goals

Why are you mentoring/helping them?

Getting to grips with the tech stack

Making a simple webpage

Building interactive applications

Helping out when the team is busy

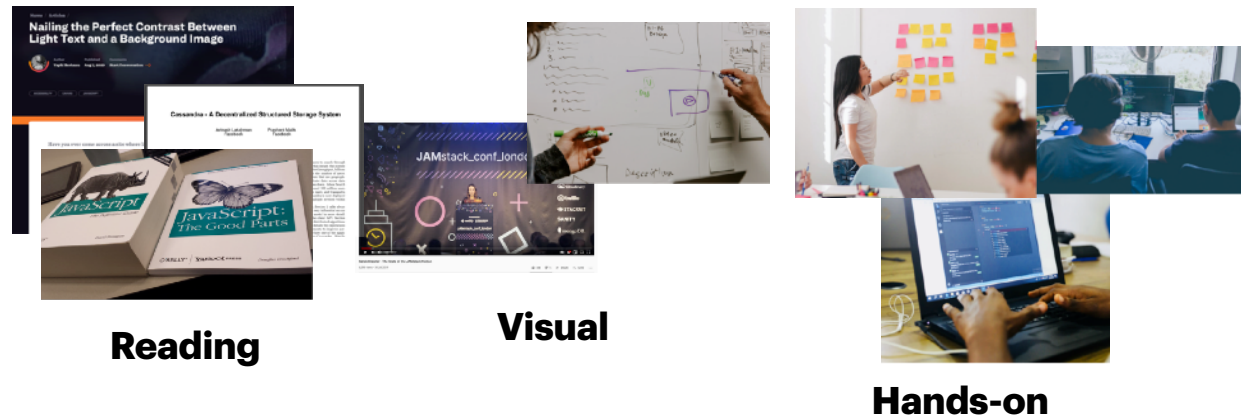
Changing career path

So just like my English students, mentees have goals. Why are they learning to code? Why have you been asked to mentor them? In our world it's not as simple as fluency vs conversational JavaScript, but there could be any number of reasons and it's really important that you know what those are from the outset. It might be obvious and you don't need to explicitly discuss it, for example when you're onboarding someone into the team. In a more formal relationship such as with a more junior developer you should discuss their learning goals and identify the areas they want to improve.

Just as I wouldn't teach someone learning conversational English how to write an academic essay, you don't want to confuse someone who wants to build a simple static webpage by teaching them bubble sort or how to use service workers. Knowing what their goals are will help you narrow down those topics. You don't need to write an entire lesson plan or structure some kind of curriculum, but it helps to make a list of the kind of things you think you should cover.

# Know your learner

## How do they learn best?



As well as knowing what the end goal of the learner is, it also helps to know how they learn best - that is, what techniques are most effective when they're learning something new.

Look at the different resources available to a learner and find out from them which is the most effective - plus, students tend to learn better from a variety of media rather than all the same. Think back to when you last learnt a language - there would have been reading exercises, conversation in pairs, listening exercises and even songs. In the classroom setting you can't cater for individual learning styles necessarily, but you can provide a wide variety of resources so that everyone's needs are met.

When it comes to developers, some will read a lot of books and blog posts, whereas others will prefer video tutorials and diagrams. Others will want to get stuck in, coding and mapping things out. Some people will mix and match these - I know for myself I tend to learn best by doing, so I prefer to get stuck in, and I also read a lot of technical blog posts - but I haven't successfully made it through an entire book about programming and I get bored watching videos.

Ask your learner what their preferred methods are, and maybe you can point them in the direction of some materials that will help them. Maybe they'd benefit from watching a video or conference talk before tackling any code, or maybe you want to give them some smaller JIRA tickets to work on where they can see how they get on and ask you if they get stuck.

And if you're leading a workshop or teaching a group, you'll want to mix and match some of these formats. A lot of your sessions are likely to be very hands-on coding time, but you might punctuate this with mentors pairing with students, or getting students to demonstrate what they've done and explaining how they've done it. Many students will benefit from teaching things themselves, as it can be a great way to consolidate learning.

Provide some resources for further reading on the topics you're teaching for students to read in their own time - granted many of them probably won't, but some will.

Make sure you include videos as well as reading material.

And if you're recording a tutorial, provide a transcript so that people who prefer to read can get the information you're giving in a way that suits them - plus, it's a nice win for accessibility for people who have difficulties watching video or listening to audio.

# Core skills

Speaking & listening

Reading & writing



There are two sets of core skills when learning a language: speaking and listening, and reading and writing. Human languages tend to have spoken and written forms, and the skills are divided into productive skills, speaking and writing, and receptive skills, listening and reading.

Now, the speaking and listening part doesn't apply to programming particularly well. But reading and writing definitely apply.

We think of being a developer in terms of writing code, but part of being a good developer is being able to read code that someone else has written and understand what it's doing. Though it depends how ridiculous the code is in the first place - remember folks, clean code will only get you so far if you're writing code that's so abstract that nobody can understand it. But the same goes for English. I once had to reject a job application from someone who had put so much effort into using language they thought was impressive that I actually couldn't make head or tail of what they were actually trying to say.

In the English classroom, we'd work on these skills separately from learning the fundamentals of grammar and vocab, but it's all linked - these skills help to reinforce everything you've learned by putting it into practice. With programming languages you have a lot less of this to pin down before you start writing code, but the more code you write, the more you'll get to grips with the terminology and constructs within a language.

English, you're not going to sit them down right at the start and ask them to write an essay. You're going to have them do a few things first. You're going to let them get used to writing in front of the

maybe a page once they're confident enough. You don't ask them to get up in front of the class.

# Pairing

- Hands-on exposure
- Idiomatic code
- Learning the “house style”
- A writes test, B makes it pass
- Timed turns



Arguably one of the best ways to build skills as a developer is pair programming. Pairing is a hands-on activity but is much better for sharing context, getting developers used to more idiomatic programming and learning the way you do things in your team or organisation.

I've worked on teams where we paired on everything as a matter of course, and I've worked on teams where we mostly worked solo but paired on things when we're stuck. I personally prefer the latter, but for bringing a new team member or junior developer up to speed, it's invaluable.

You might pair by taking turns to write a test and make it pass, or you might time the turns to make sure everyone gets a fair go.

## Pairing 101: they're driving

- Get them to explain what they're doing
- Ask what they want feedback on
- Resist the temptation to correct every mistake
  - Let them figure out errors
- Save the style comments/nitpicks for afterwards
- Different pairs

To get the most out of pairing, there are some key things to bear in mind. I could apply many of these to 1-on-1 conversation practise with students as well.

When your mentee has control of the keyboard, or driving as it's often referred to, encourage them to explain what they're doing. That's not to say they should narrate everything, but this will help you to understand whether they're on the right track, as well as helping them to cement their own knowledge.

Make sure you know what it is they want feedback on, so that you can pay extra attention to it.

Let them make mistakes. This is really important. Don't correct the code before the error shows up - the compiler or the output will let them know something is wrong. Let them try and debug it themselves - you're there if they need a nudge in the right direction, but try not to tell them exactly what's wrong unless they ask you. And even then, you can try and nudge them in the right direction some more by asking strategic questions rather than telling them outright.

Imagine you were telling a story in a foreign language and your teacher kept interrupting you to correct you - it'd put you off your train of thought.

We all have house styles and preferred ways of doing things, but don't sit there and nitpick their code because it will be very offputting. If what they're doing works, that's fine - you can go through it together afterwards and look at ways of improving it and refactoring it.

And if you can, get them to pair with other developers on the team to give them a different point of view. They'll learn how others do it, and you might even learn something from them.

# Pairing 101: you're driving

- Act (fairly) natural
- Explain what you're doing
- Share the tricky stuff

When it's your turn to drive: act natural.

When teaching English students, I couldn't chat away at my normal pace to the beginner and intermediate students, as they wouldn't understand what I was saying. And the same goes for me - I speak passable Spanish, but I really struggle to understand normal spoken Spanish as it seems really fast to me. When you've got the keyboard, take care not to zoom ahead too fast unless it's clear the other person can keep up. If you're a super speedy programmer whose brain can't keep up with your fingers, you'll need to pace yourself a bit.

But conversely, you shouldn't go too slowly. We were taught in teacher training not to speak suuuuper slowly like.. this... because it's not natural and doesn't help students to understand. Coding really slowly on purpose will be frustrating for the person you're pairing with, and it can interrupt the flow of work. Plus, if you go *too* slowly, it'll just seem patronising. Asking for feedback about how they're finding it will help you to pace yourself effectively.

When you're writing code, try to explain what you're doing. Sitting in silence as a pair writes code isn't particularly interesting, but also it will help your pair understand why you're doing what you're doing.

And don't keep all the hard problems for yourself - even if it seems like you're doing them a favour. If you end up with a tricky bit of syntax or a hard problem to solve, let them drive, and work through it together.



## Teaching technique: **Eliciting**

**“What do we call someone who writes books?”**

**“What is the person doing in this picture?”**

**“What did the first person order in the restaurant?”**

Eliciting is a great technique that we used a lot in the classroom to get students interacting in the lesson. Well, technically it's actually a range of techniques, but it involves anything that gets the learner to provide information rather than the teacher telling them.

It wasn't just to make sure they were still listening, although it helped. Eliciting is a way of activating the knowledge the students already have, as well as helping them discover things themselves which makes them more likely to remember it. It's also really helpful for the teacher to find out what the students know, and what they need more help with..

In the classroom setting, we might elicit sentences from students by drawing pictures on the board, or asking them questions and asking for examples of things we're talking about, for example after learning new vocabulary or doing a listening exercise. With younger groups it's often better to nominate students to answer the questions, rather than asking 'does anyone know...?' Because often that results in complete silence, or the same people answering over and over. Adults are generally a bit more willing to share, and you can just get people to shout out the answer - though this depends on the group.

## Eliciting when mentoring

“How can we find where the error is coming from?”

“What should you do with the error here?”

“How can we make this purple?”

“Which rule was it that controlled the number of columns in the grid?”

We can use this technique when mentoring developers as well.

For example, when pairing, ask those strategic questions to help your pair debug a particularly tricky error or to find the syntax they can't remember.

I found this technique worked especially well when mentoring kids - for example, in sessions where they're building a web page according to instructions on a worksheet, it engaged them a lot more when they had the opportunity to think creatively about what they could do. Point to the heading and say “how could we make this a different colour?” - they could look for the colour in the CSS and, with a little help from you, change it to a different one. It really personalised the experience for them, plus they got super excited when they realised they could put ANY image on their website.

## Getting the most out of eliciting

- Right audience
- Use sparingly
- Ask open questions if you can
- Be encouraging

Eliciting works best in more formal mentoring contexts, such as coding workshops or pairing with a more junior developer, because you can imagine in a 1:1 meeting your mentee might start thinking you're being a bit odd if you start asking them a ton of questions. And I wouldn't necessarily recommend using this technique with an experienced developer you're onboarding onto a team because it might come across a bit patronising. But if the person you're talking to is there to learn something new, and you're trying to establish what they know and help them commit stuff to long-term memory, it can be a great help.

Use it sparingly: don't ask questions constantly. It's unnatural and can throw off the mentee, or make them frustrated if they just want to learn something. If they don't know the answer, move on, don't keep asking questions - either ask someone else or tell them the answer. In workshop settings, ask a question now and then to assess the attendees' prior knowledge, or to consolidate something you've just taught. When you're pairing too many questions from you can disrupt your pair's concentration on the code, so keep it light.

It's also important to choose your questions wisely, and be careful how you ask them. It helps to ask questions that are a little open to interpretation, or don't have a yes/no answer - students are more likely to answer if they feel like they're not going to be wrong. That's obviously easier with English, where there are about 5 ways of saying the same thing, but you could arguably say the same thing for JavaScript these days.

And if they do get the answer wrong, rather than saying "no, that's not right", say something like "not quite" or "close" - be encouraging. And if necessary, give a refresher if people really don't know the answer.

**How teaching skills can help with...**

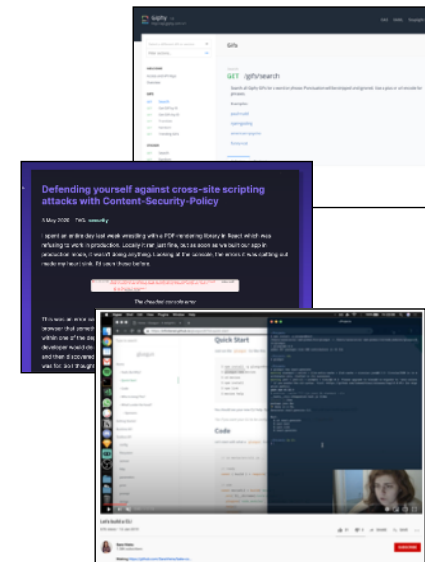
# **Documentation**

The teaching skills I learnt have also helped me massively with the documentation I've written in my career. This has ranged from readmes to full tutorials and even lengthy Slack posts announcing things.

Documentation is often seen as a chore, and can even be skipped entirely, but the power of good documentation shouldn't be underestimated. Good documentation can be the difference between someone getting their work done efficiently and having to stop all the time to find someone to answer questions.

# What constitutes documentation?

- Software documentation e.g. API docs
- Tutorials, blog posts
- Conference talks



So as I mentioned, documentation can take many forms. There's the readmes and docs for the software we build, which could be in the repo alongside the code, actual code comments, or hosted separately.

We might write tutorials and blog posts to share knowledge with others.

And of course there's conference talks just like this one - even though it's not a written medium necessarily, it's still a form of documentation, and many of the same rules apply.

# Web 101

Introduction to Web at Monzo

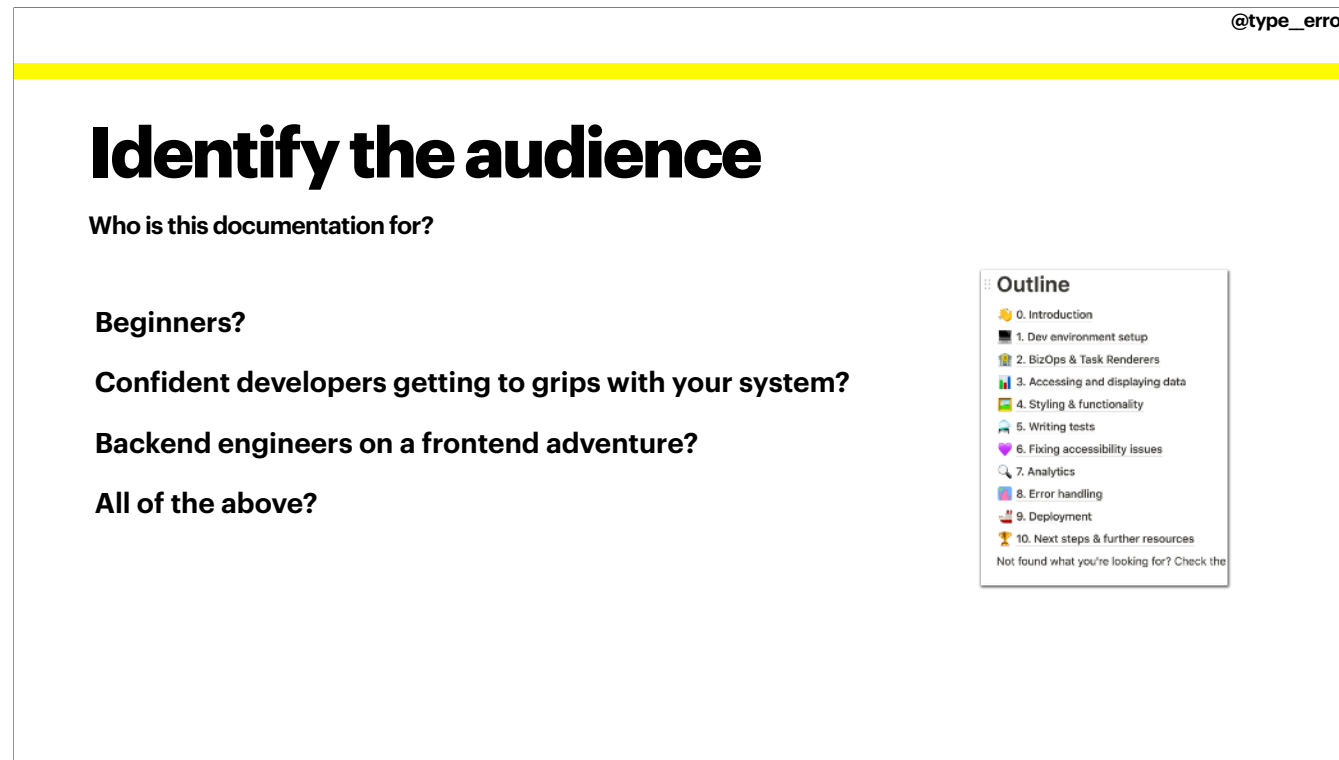


## Web 101

### Outline

- 0. Introduction
  - 1. Dev environment setup
  - 2. BizOps & Task Renderers
  - 3. Accessing and displaying data
  - 4. Styling & functionality
  - 5. Writing tests
  - 6. Fixing accessibility issues
  - 7. Analytics
  - 8. Error handling
  - 9. Deployment
  - 10. Next steps & further resources
- Not found what you're looking for? Check the [Web Engineering Handbook](#).

I'm going to illustrate the points in this section with examples from Web 101, a self-guided tutorial I put together at Monzo. As we're only a small team of web engineers, much of the web work in internal tooling is done by backend engineers in sponsoring teams, and not all of them are super familiar with how we do web at Monzo, so we needed something to teach them best practices and the basics of things like accessibility and testing. I also wanted to create an onboarding document for new starters in web. So Web 101 was born, to cover both of those bases. It's been really well received, and a lot of that is down to the way it was written.



Just like with mentoring, our first step is to identify who the learner is - in this case, the consumer of our documentation. Are they complete beginners to programming? Are they experienced developers looking for the query parameters for an API call? Like with Web 101, are they backend engineers who need to do some web work?

Sometimes you'll have a real mix, so the challenge is writing documentation that suits everyone's needs.

In the world of teaching, teachers will assign work for the level of the majority of the class, and have stretch exercises for the more advanced students once they've finished the work. We can apply this to the technical documentation we write as well. Less experienced engineers will be able to learn at a level that suits them, and more experienced engineers can quite happily skim over the stuff they already know.

With Web 101, we broke everything up into sections so that a developer starting from the beginning could work all the way through, but someone with a bit more context could pick and choose the section that was most applicable to them. The advantage of this was that it's also easy to link to particular sections from other documentation, so for example we have the section on developer environment setup linked from various READMEs.

# Different information for different skill levels

I'm familiar with GraphQL, take me to the bit where I can write stuff

GraphQL is a **query language** and corresponding architecture that allows us to define both the **shape** of our data, as well as the **ways we can access and modify** that data. It was created by Facebook as a way for their mobile apps to request **only the data they needed**, instead of the full response object that you'd normally receive from a REST based request architecture. It also removed the need for clients to perform multiple requests and then stitch the result together - both of these are examples of *overfetching*.

A GraphQL server is made up of three distinct pieces which work together: the schema, operations and resolvers.

**The schema**

Provide opportunities for users to skip sections they don't need - here, in a section on the basics of GraphQL, we have a link to skip to the actual tutorial work, if they don't need the primer on what GraphQL actually is.



# The importance of register



We use the language our audience uses, and make technical stuff as clear as we can

Swap formal words for normal ones

We're friendly people, and we don't want to come across like a cold, faceless organisation. So use the kind of language you'd use if you were talking with the person you're writing to, and avoid business-speak.

The best test for this is to read what you've written out loud. Does it sound like the kind of thing you'd actually say? If not, some of the words below might be the culprit.

Would you say...	Oh...
Assistance	Help
Commence	Start
Enable	Let
Ensure	Make sure
Further	More
However	But

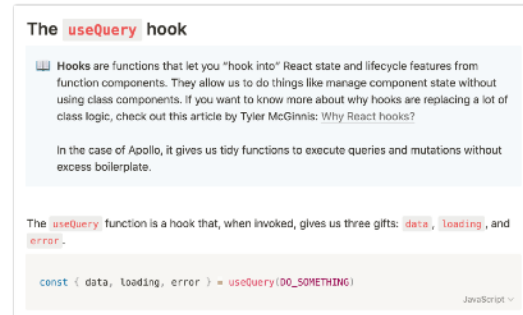
<https://monzo.com/tone-of-voice/>

Register isn't a term we hear very often in terms of language - it means a variety of language used for a particular purpose or situation. For example, I would use a more simplified version of English when talking to my class, whereas when I'm giving a conference talk I use words like "whereas". Sometimes it's less conscious, for example you might use more slang with your friends or peers than with your manager without even thinking about it.

When it comes to written language, especially in a business context, we tend to talk about tone of voice. This is almost like the register of your writing - how you say what you're saying. At Monzo we have a very distinctive friendly tone of voice which is aimed at making sure all our customers can understand things like terms and conditions and weird banking jargon, which there is loads of. Our tone of voice was drafted by our amazing head of writing, Harry Ashbridge, and the article has some great tips for keeping your writing simple and clear.

The world of programming is enormous, and people may be reading your documentation from all over the world, with different levels of English. Writing in a clear, concise and informal manner will mean that *everyone* can understand what you're saying. It's also a win from an accessibility point of view as well, as people with reading and comprehension difficulties will find it much more digestible.

# Don't assume prior knowledge



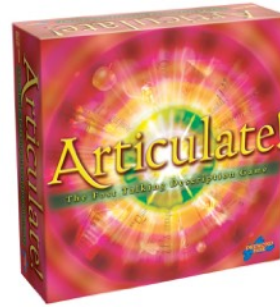
When teaching English students, we had to be careful not to assume they already knew stuff - especially things that are tied to local culture. Slang words that we use without even thinking, which might seem obvious to use but actually might make zero sense to the learner. This can end up with the student feeling disheartened. So we taught them slang as a formal lesson because they'd need it in everyday language, but used more common terms when communicating in lessons.

With documentation, assuming too much prior knowledge can have the same effect. You need to make sure your reader has sufficient context and understanding for your documentation to be useful. If you don't want to spend the time explaining the background knowledge required, put in a section at the beginning outlining what the required background is and add some links to find out more.

In Web 101 we take engineers through creating a new view on our internal customer service system, BizOps. We made sure to add a background on what BizOps actually is and how it's used before getting into the code. We've also added explanations for various concepts for whoever needs them, but as callouts rather than in the body text of the tutorial so that it's easy to skip over them if you don't need it. For example, when we're guiding the reader through adding a GraphQL query, we explain what hooks are in a callout before going into the detail of Apollo's `useQuery` hook.

I'd recommend getting a less experienced developer than you to read through your documentation to identify anything that they don't have context on. Then you can expand on things where needed.

# Explain technical concepts clearly



A great game we used to play with our English students was Articulate - where you are given a word, and you have to explain what it means to your team without using the actual word. It was a fantastic way for students to practise both their vocabulary, and explaining things - description skills are useful in foreign languages when you can't remember the word you want to say! I once bought tissues in a pharmacy in Spain by asking for "small paper for my nose". It's something we had to do as teachers too, because if a student asks what a word means, we can't use that word to explain it! And we have to explain its meaning in words that the student can understand.

Being able to describe technical concepts in simple terms is such a useful skill - I really can't state that enough. A clear explanation can be the difference between someone sitting and banging their head against the desk and feeling stupid, and a lightbulb going off and everything becoming clear. Some concepts are easier to explain than others, but a good explanation can really transform your documentation.

If you want to practise your skills explaining technical concepts, why not put together a list of words and have a game of Programming Articulate with your colleagues? It may sound a bit silly, but it's actually really good practice for being better communicators. The non-technical folk in your team will be glad for it when you can explain exactly what you're working on in non-technical terms.

### The schema

The schema is the **single source of truth** for our data. It consists of **types** and **fields** that define what data we can access, as well as what types we expect to get back from the operations we can perform. Let's take a look at a simplified version of the schema for our doughnut service:

```
type DoughnutPurchase {  
  purchaseId: ID! //fieldname:type  
  userId: ID!  
  date: Timestamp!  
  price: Money!  
}  
  
type Query {  
  doughnutPurchase(purchaseId: ID!): DoughnutPurchase  
  doughnutPurchasesByUserId(userId: ID!): [DoughnutPurchase]!  
}  
  
type Mutation {  
  buyDoughnuts: DoughnutPurchase!  
}
```

GraphQL ↗

First, we've defined a **DoughnutPurchase** type which has four fields; each field also has a **type**.

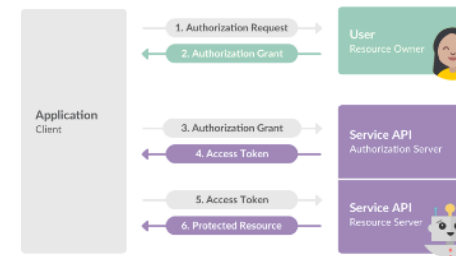
Next, we have two special types, **Query** and **Mutation**: these tell the GraphQL server running this schema what **operations** we can perform. The operations are listed within alongside their **arguments** (in parentheses) and their **return types** (after the colon). Exclamation marks mean the type in question is **non-nullable**:

- Exclamation marks in **arguments** mean that they must be included with a request.
- The exclamation mark next to the **return type** tells us that the request will always return something of that type, never null.

In this case, we can **query** a **doughnutPurchase** by its purchase ID, and we **may** get a **DoughnutPurchase** back, or **null** if none exists.

In Web 101 we could assume a base level of programming knowledge, but things like GraphQL wouldn't be familiar to everyone so we had to explain them. The tutorial centres around doughnuts and writing a frontend view to manage doughnut purchases, so using this to illustrate what a schema looks like, we explain exactly what queries and mutations are, and what the various constructs in the schema do. Significant words are bolded, and explanations are simple, not too wordy.

# Analogies and stories



A great way to explain technical concepts simply is to use real-world analogies. We'd use this in the English classroom too, for English words our students didn't understand - using real-world concepts and stories to model the meanings. For example, you might describe "embarrassed" by describing a series of really embarrassing events as if they happened to you.

A few years back I worked on a team where I was implementing an OAuth-style authentication flow with refresh tokens, and our product manager wanted to know what it involved - she had no background in tech. So I ended up illustrating it using a ridiculous analogy involving a secret clubhouse, where you say a password to get in, but then there's a secret handshake (the access token) you need to speak to anyone. The handshake changes every 5 minutes, and you are given a special badge (the refresh token) that you need to show when you want to learn the new handshake. And when you leave the clubhouse, they take the badge off you, and you get a new one next time you come in. It was silly, but it got the message across.

Another good exercise to do with your friends and colleagues is to try and come up with ridiculous analogies for technical concepts. See if they work on your non-technical friends and family members, as well.

# Explain why



I spoke before about the importance of knowing your learner - understanding their goals and intentions. What you teach them should be relevant for their goals. READMEs and tutorials are a bit easier for this as you know who's going to be reading them, but for a blog post you can imagine who your target audience might be and write for them. But presenting facts to the reader will only get you so far.


I would teach my students all sorts, from letter writing to describing injuries. We even had a lesson on the f-word once, with all the ways it can be used in English. The thing is, if the students don't know WHY they're learning this, it won't be nearly as effective. You learn how to write letters so you can write a cover letter for a job or sort out visa things. You may need to make a doctor's appointment sometime when living abroad, or have a medical emergency on holiday. And living in a foreign country, you'll hear people swearing, and you'll need to understand what they're saying!

With documentation, explaining *why* will help the reader to understand what they're doing. Why do we need these environment variables? Why is it bad to give an onClick attribute to a div?

Giving the additional context will make things seem clearer to the reader, and make it seem less like magic.

# Explain why

The build script checks for Flow type errors, transpiles our nice fancy syntactic-sugared JavaScript and React code into browser-readable JavaScript using Babel, and bundles it using Webpack.

 Wondering why we need to transpile JavaScript? Check out this article while you're waiting for BizOps to build: [JavaScript Transpilers: What They Are & Why We Need Them](#). It's a little old now, but most of the article still stands.

```
<headers>
<main>
<footers>
<section>
<nav> - for a menu or toolbar
```

Using these semantic elements builds a map of landmarks on the page, allowing it to understand what each part of the page does and where the important elements are. This means assistive technology like screen readers can jump to the relevant sections and help the user navigate the page.

In Web 101 we made sure to explain things like *why* we have to run a build script when browsers already understand JavaScript, or *why* we should be using semantic HTML. It just gives that extra bit of clarity, and especially with the semantic HTML it makes the reader more likely to actually do it.

## It's not "simple"

I should have been doing it.

They might have had dinner already

You would have been able to do it.

Imagine learning a new language - whether that's a human or programming language - and trying to get your head around a difficult concept. Look at these sentences - examples of what we call modal verbs - and imagine trying to get your head around this as a non-native speaker.

Some people find grammar a lot easier than others. I have always been more of a grammar person than a vocabulary person. And when learning modal verbs in German, for example, I found them comparatively straight forward, but some of my classmates thought they were really confusing.

In one Spanish class I took, I was struggling with understanding something and said it was difficult - the teacher said "no, it's easy". It was horrible, and I didn't learn anything. And I felt like crap.

Now imagine looking through the docs for a library you're using, and being presented with this sentence:



# It's not "simple"

Simply synthesize the circuit, then just get to the VGA sensor  
through the online HDD address

<http://shinytoylabs.com/jargon/>

It makes literally no sense, and yes that's because it is literally complete nonsense, but for someone reading a technical document about something they're not familiar with, what you've written might seem like this too. What seems simple to you won't be simple to everyone.

We don't use the words "simply" or "just" or anything like that in Web 101. All it does is make people feel bad about their own abilities if it doesn't seem simple to them.

## Web 101: the outcome

***“I just wrote a brand new bit of code following the guidance there and it just worked 🤖 (I find this happens very rarely in engineering, and is testament to the quality of the docs)”***

We got some great feedback on Web 101, largely down to the consistent, clear writing style, the background information and the thorough explanations we wrote.

## **Practising your teaching skills**

I've given you a lot of things to try out in this talk, so if you want to put them to the test, here are some things you could try.

## Mentoring opportunities

Brighton Codebar: [codebar.io](https://codebar.io)

CodeFirst Girls: [codefirstgirls.org.uk](https://codefirstgirls.org.uk)

Find a junior developer in your company

Offer to onboard a new starter

## Documentation

Contribute to open-source projects on Github

Start a blog!

[DEV.to](https://dev.to)

# Thank you!

@type\_error

[sophie@localghost.dev](mailto:sophie@localghost.dev)