Guide to Developer / Engineering specs

Agenda

- Introduction
- What is a dev spec and why do I care?
- How do I write effectively write an effective dev spec?
- Effectively presenting your spec

What is a dev spec? Why do I care?

What is a developer / engineering spec?

- A dev spec is a step along the way of the journey towards building the right feature / product experience
  - It is a document that represents the engineer's point of view
- But what is the right feature and product experience?
  - It's what delights customers
  - It aligns to Microsoft's goals
  - It aligns to our division goals (DevDiv, C+AI)
- A dev spec helps us refine these. It does this by helping us:
  - Refine our goals & customer experiences
  - Collect feedback from key partners, stakeholders
  - Communicate our intentions

What is the purpose of a dev spec?

- Ultimately – a dev spec communicates what you are building. Through this communication, we can iterate to challenge and verify:
  - The viability of what we are building
  - The value of what we are building
  - The plan for how we are building it
  - The implications / consequences of how we are building it
- By going through this exercise, you hope that you spend the least amount of time building the most value

Measure of success

- The measure of success then is how much time you “save”
  - Keep in mind though that “time” is not just time to write the initial code; we should also consider support costs, live site issues, and potential re-writes down the line, customer pain points, cost of changes (versioning), and so on
  - In other words, it can potentially take a long time to evaluate the benefit of a spec, and sometimes, there is higher initial cost that “blinds” us to the real value

Process

- Authoring and reviewing dev specs is part of a bigger process
  - Review process
  - Code and design update process
  - Deprecation process

So why should I care?

- Product success
- Career success
  - Networking - especially if your spec covers interactions between teams
  - Mentoring - especially if your spec is a shining example

How do I write one?

What goes into a dev spec?

- Ultimately anything and everything you need to achieve your goal
- In practice we can use a template to help us and we can add/remove from the template as needed
- Since we are in DevDiv, I'm going to focus on spec content and issues that are relevant to "platform" (and maybe less relevant to "applications" and "services")

Who is the audience?

- There is a primary audience – you
  - Writing specs help you
- But there are others to consider:
  - People who will integrate with your feature
  - Your immediate team (other engineers, PMs, etc)
  - Your engineering leadership (managers, architects, etc)
  - Your partners (engineering partners, PM partners, etc)
  - Your friends in documentation / writers
  - Your customers (developers likely will end up using what you are building directly)

How should I proceed with a dev spec?

- My preference for ordering:
  - Table of contents – structure from H1 → H3
  - Introduction - Background / Context information
  - Contracts (APIs – bi-directional, file system, etc)
  - Flows (flows are implementations of scenarios through your contract)
  - Architecture block diagrams
  - Revisit the Introduction

Use the ToC to help structure your document

- Think of your document as a communication device. Most authors / song writers have some sort of structure – for example:
  - Happy state, tension, character building, resolution
  - Intro, v1, chorus, v2, chorus, bridge, v3, chorus, outro
- Build the ToC first:

Contents

Contents

Start with public contract - APIs

- For platforms consider the public contract – things that other developers will be interacting with
- Generally the API is the most important public contract
- There are two APIs to consider, both important, but in priority order:
  - The public API contracts
    - The boundary to your system
  - The internal API contracts (especially important once your project has multiple people working on it)
    - The boundary between blocks composing your system

How to start with APIs?

- Write a contract in code that you don’t mind (should?) throw away
- Interact with the API to test its effectiveness and your design
- I will often choose something simple a quick to get started:
  - For code APIs, I’ll write a simple API contract (in Typescript) and write unit tests
  - For REST APIs, I’ll use Swagger and test it with Postman / Insomnia
- When you have an API, generate docs from it using your favorite doc generator (api-documenter and api-extractor for Typescript)

Discuss other “public contracts”

- APIs are not the only public contracts
- Consider also:
  - Developer-facing events (these are really APIs but sometimes we forget that these are)
  - Logs / events – teams need to build dashboards, monitors, etc on these. These should be documented because changes may break your monitors
  - Configuration
  - Database schemas

Once your public contract is done

- Review it! (This is the iteration process for design reviews)
- Then move on to implementation details:
  - Data and control flow diagrams
  - Block diagrams

Flows – data and control flows

- Flow diagrams help visualize and document the “workflow“ of operations through your design
  - Should include global flows and local flows
- Control flows
  - Swim lane / sequence diagrams are a good choice to visualize control flow
- Data flows
  - Shows how data transforms from “input” to “output”
  - The focus here is to show the transformation of the data and the key processes that affect data
- Why work on flows before block diagrams? Flows can help you formulate the right blocks to create. If you create blocks first, you’ll force the flows to fit the blocks

Architecture

- This section should give a high level view of the architecture – starting from how your feature fits into a larger system, to the way your feature is structured
- Building blocks is appropriate here

Service considerations

- For services there are additional sections that need to be covered
  - SLAs
  - Security
  - Compliance
  - Disaster recovery / Continuity
  - Sharding / Geolocation / Backup

Spec template

- I’ve attached a template which might be a good starting point for you
- It’s focused on tooling – if there is interest I can extend this to services

Presenting your dev spec

Presenting

- Presenting a spec is an important consideration
- Presenting can happen via:
  - A person reading the spec off-line in a doc
  - A meeting where the author speaks about the doc
- The goal of presenting your spec is to:
  - Communicate the context
  - Communicate the design
  - Ask for feedback

Some tips

- Practice! Understand the order that you want to present in:
  - Start with introducing the topic
  - Describe the interface points next
  - Finally, go through the design details
- Take notes! Actually – have someone take notes for you
- Make space for people to give you feedback
  - If it’s going to take you an hour to go through your spec, you will not have time for feedback, which means you lose a major benefit of this process
  - Prepare some questions for your audience – engage them

Wrap up

Some myths

- Specs are easy to write ➔ they take an incredible amount of time, but they will save you even more time
- Specs are too heavy weight ➔ A right-sized spec provides the right balance
- Specs slow us down ➔ Specs help us run faster and smoother, especially in the long run
- Specs are for “waterfall development methods” ➔ Specs are a tool that can be used in any development method
- Specs benefit others ⇔ Specs benefit you
- Specs are only useful for “big” things ➔ Specs can be useful even for the smallest feature

Thank you!

- Location of slides and template
- Take the survey and give me feedback - https://www.surveymonkey.com/r/JXGGKMF