Micro-services

Polycode TAD

Simon LUCIDO

Version 1.0, 2023-01-22

Table of Contents

Introducing microservices

Introduction

The web development world has evolved from a simple, monolithic, centralized model to resilient, highly scalable and flexible architectures over the past decades. This means that, getting it right, in what may appears as a convoluted and overly complex way of handling simple requests from users, has become complicated. In this paper, we will explore the key concepts and considerations involved in migrating to a microservices architecture. We will begin by defining microservices and discussing their key characteristics and benefits. We will then delve into specific topics, such as authentication, communication between microservices, tracing and logging requests, and handling security. We will also discuss how to integrate a mobile app with a microservices architecture. We will try to dive into what makes sense and what doesn't for our Polycode application.

For those unfamiliar, Polycode is a platform aimed at developers, offering courses and challenges around development. It mainly focuses on learning languages or coding concepts, and provides interactive development session where the user can mess around with its code, trying to match the requirements of the exercise.

However, the service has evolved quite rapidly, with new features being requested and added. The architecture and the code must reflect this dynamic, fast-paced constraint. Wide-range of features are being developed and will be developed, and this decoupling must be reflected in the approach taken by the project's development team.

The newest feature currently is development, is a new service that opens Polycode to be a assessment platform. Schools will be able to create tests for candidates, and get summed up scores and results.

It is important to note that, although this will all be related to Polycode, not every answers aims at directly solving the problem at the Polycode level. Some suggestions are broader, and can be applied for most microservice architectures. With the same mindset, most suggestions can be applied with no system already in place, and can be used as foundations for building a microservice architecture, with no existing system.

The assumptions

Before continuing, I would like to pause on some assumptions made along this papers. Unless specified, those assumptions will always be taken as true all along this document.

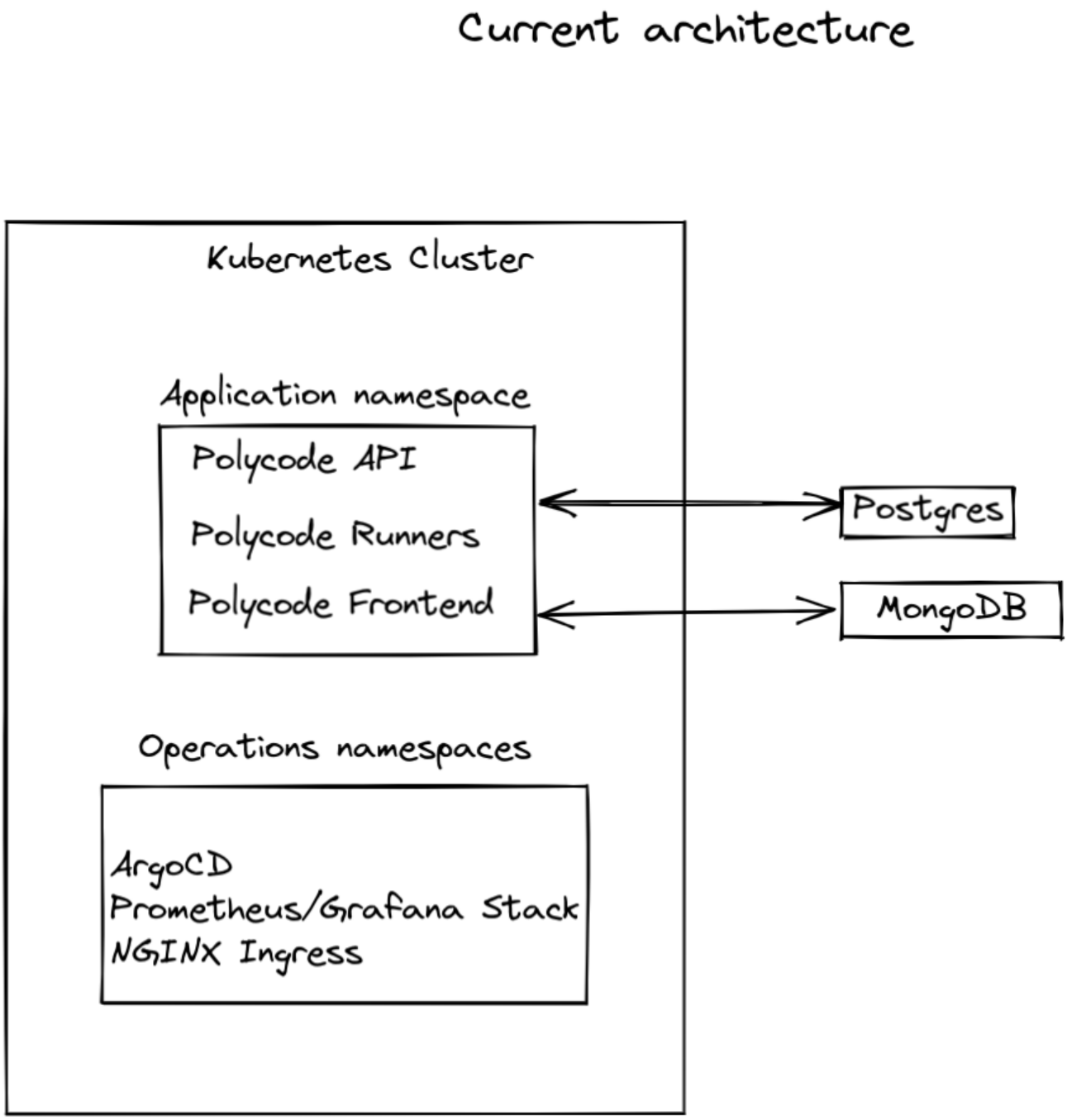
First off, this project is maintained by students, in a scholar environments. This is a tool for learning as its core, and monetary aspects will be dampened in the suggestions. I would suggest that, if it was not, migrating to microservices would be a waste of resources. I will dive more into that later in the paragraph.

I will also take the stance that we have limited time resources, since the team is fixed and can't grow. I will assume that the team will be able to work for a few months specifically on this migration, letting us enough time to implement reasonable solutions. Rewriting a HTTP Proxy from scratch is not reasonable. Setting up a Service Mesh is.

The existing

We first need to understand the current architecture of Polycode. To do so, I think looking at a (very) simplified diagram of the current architecture will best describes what is currently going on :

Current architecture



To give some contexts for those unfamiliar with the project, what I call "Polycode Runners", is the API that allow us to run the code that the user typed in.

Very quick pass on the stack we have :

- Kubernetes as our container manager.
- ArgoCD as our GitOps engine
- Nginx Ingress Controller as our Kubernetes ingress.

- The Prometheus/Grafana stack for our metrics and monitoring.

As you may have noticed, the current implementation is halfway between an old monolithic application, with a heavy native application, running on a bare-metal server. Instead, we are already in a Kubernetes context, which already allows for automatic scaling, if the application is made with the correct architecture decisions.

It is also worth noting that the current implementation mostly follows a service-oriented architecture.

Current limitations

As shown in the diagram above, except for the runner, everything is still mixed up in one big application, the API. This gives us enough flexibility to scale our application if the user count were to increase, but we would spin up a heavy server every time, although probably only a small portion of our application takes all the load.

For example, we currently handles the authentication within the API. The users routes are hit pretty heavily, since every time the user loads the website, they try to fetch data about themselves. This might take over half of the available resources of the container (totally arbitrary, for the example), and the team routes only taking less than 1 percent of the available resources. This can lead to poor performance in high-traffic environments.

Another downside of the current implementation is that the application is getting harder to maintain and update, caused by its growing codebase size and the intricacies and dependencies between each part. It becomes hard to introduce new features, or update existing ones, without running the risk of breaking other parts of the code, although this problem is softened by our code architecture. Developing new features in parallel has also taken a hit, for the same reasons. Deploying it is a heavy process, since the API as a whole needs to be recreated in our cluster.

Microservices

Microservices seek to solve those problems. We will try to understand how, and at what cost. But first, let's define a microservice.

What are microservices

Microservices are a modern architectural approach to building software applications. They involve splitting a large, monolithic application into smaller, independent services that can be developed, deployed, and scaled individually. Each microservice has a specific role and communicates with other microservices through well-defined interfaces, typically using a lightweight messaging protocol.

Why microservices

Scalability

One of the key advantages of microservices is that they allow for greater flexibility and scalability. Indeed, because microservices are typically deployed in containers, they can be easily scaled up or down to meet changing demand. This is perfect for our use case, since we are already running a Kubernetes cluster, whose purpose is to provide tools to automate this task. This resolves the issue mentioned above. We can now spin up 15 authentication microservices, while only having 3 team microservices.

Resiliency and decoupling

Another advantage of microservices is that they are designed to be fault-tolerant and resilient. Because microservices are modular and independent, if one service fails, the others can continue to operate, minimizing the impact of any issues on the overall application. This is not a given feature of microservices, but the architecture sets the groundwork for developers to build systems that are resilient, giving options to exploit the decoupling of microservices to achieve a great resiliency and have fail-over solutions.

Reducing complexity

Because each microservice is a standalone component that communicates with other microservices through well-defined interfaces, developers can easily add new features or capabilities to the application by building and deploying additional microservices. This makes it possible to rapidly iterate and improve the application without having to make changes to the entire codebase.

Finally, microservices can be easier to test and maintain than monolithic applications. Because microservices are modular and self-contained, it is easier to test individual services and ensure that they are functioning correctly. Additionally, because each microservice has a specific role, it is easier to identify and fix issues when they arise, and to make changes to individual microservices without affecting the rest of the application.

The cost of microservices

As you might expect, this doesn't come without downsides. This is not a perfect fit for every situation. Microservices are a great architectural tool, but like every tool, you need to use them wisely.

Cost of infrastructure

The first thing I would like to touch on, is the cost associated with running a microservice architecture. As you might have realized, running microservices come with a big resource overhead. Running multiple containers, each allocating resources for your language runtime (if applicable), running inside a Kubernetes cluster, that, by itself, will reserve some more resources for services, ingresses, internal DNS, will require more resources. For a basic, low traffic service, with no requirements or low requirements on uptime, microservices will add significant cost to your infrastructure. Stick to a well-architected monolith, as you will not benefit from a microservice architecture.

Time and entry barrier

Another aspect to microservices that can be a limiting factor to you, is the added complexity compared to a simple, heavy, monolithic application. You will both need a team architects that have the skill set and the knowledge to actually build a architecture that makes sense (which is not necessarily easy) and a team that can code in a "cloud-native" way, meaning they understand cloud patterns, how to build stateless applications, how to handle failures and how to define stable and sane APIs. You will also need experts to monitor and identify problem with your infrastructure. Developers don't typically now how to handle operations properly, you'll need to hire someone with this knowledge to actually keep an eye on your logs, metrics and traces, giving an helping hand to developers that might need help.

These teams may need to adopt new tools, processes, and ways of working to support the development, deployment, and management of microservices. This can require significant training and organizational changes.

Migrating

The last point I would like to touch on, applies to teams and project which already have an application running, in the form of a monolith. The process of migration is bumpy, and will cause headaches. Decomposing a monolithic application can be a complex and time-consuming process. It requires a deep understanding of the existing application and its dependencies, as well as careful planning to ensure that the resulting microservices are maintainable and scalable. There are tools you can use to ease this migration, such as the strangler pattern, which aims at destructuring your monolith and putting your business logic into microservices step by step, while putting the new features in their own microservice from the beginning. However, this also requires educating your team, as mentioned above.

Does it really makes sense ?

With that being said, we need to take a short time to stop and reconsider if migrating Polycode to microservices is actually worth it. Our current application structure and deployment scheme makes the migration easier than it would be with most of the monolith out there. But as a company, you might see that this project is getting little to no traction, and would probably try to limit expenditure for a project that is not showing signs of growth. You could flip the problem the other way around, and say that you need to invest more to actually have growth, but this is risky, and adding new features is bad, but not too bad as of right now. The load is next to none, scalability is not a problem, and I would argue that we currently have enough flexibility if the project were to gain traction to scale the application enough to have the time to react and rethink our system. I would even argue that this project is already too costly to run for what it is right now, architecture wise (although negligible at this scale) and employee wise. You would need developers that know how to build for the cloud, an operation employee to monitor and maintain your stack. Those are very expensive, and are a huge upfront investment that might yield no return.

Of course, all those considerations are out the window when you take into account that, we are not a company, but a group of students, working for free, with time to spare, longing for new technologies and complex systems. The downsides for us are negligible, and curiosity and the learning experience is worth it every step of the way.

Wrap up

With that being said, we will now dive and explore the microservices world. But before getting all technical, and before talking stacks, implementation, we first need to understand what we are working with, and how we can define our microservices in a sensible and maintainable way.

From domains to microservices

Introduction

It’s important to recognize the way you design you microservices will be reflected on the quality of your application. Poorly divided microservices will result in poor performance, poor maintainability, poor developer experience, and overall a bad user experience. If your data is poorly isolated, it will span multiple microservices and create an integrity and consistency mess. Make microservices too big and you lose the benefits. Make microservices too small and you begin to overload your network infrastructure, increase costs, decrease maintainability and increase the risk of bugs. Whatever your implementation, a poorly designed infrastructure will cause headaches. Let’s see how we can create a sensible microservice architecture, using Polycode as our use case.

The first step in decomposing your application into microservices is to understand the domain that it lives in. To make sure that everybody is on the same page, and to provide a common vocabulary for your engineers, marketing team, executive team, architects and whoever might be working on the project, we need to define the language that is used around your project.

Ubiquitous Language

The ubiquitous language is a common vocabulary everybody involved in the project should define together and use to communicate. The goal is to make sure that everybody refers to the same concept when talking about the project, reducing confusion and misunderstanding, avoiding conflicts and unnecessary discussion.

As the Polycode team, we defined the following vocabulary for the Polycode project. Please take some time to read through it, to make sure your understanding is aligned with mine.

Word	Definition
Practice	A section of Polycode made to train users
Assessment	A section of Polycode made to evaluate user
Admin	A privileged user that have access and permission to manage all resources in Polycode.
User	Someone who has a Polycode account.
Candidate	A person who participate to a test or assessment.
Guest	Someone who doesn’t have an account.
Captain	A user who manages a team.
Assessment Creator	A user who can edit or create assessment, and invite candidates.
Practice Creator / Content creator	A user who can edit or create his content.
Test	It is an ordered group of contents made to evaluate users. It has a grade, which is the total score divided by the total number of points in the set of contents. It is created by an assessment creator.
Campaign	Usage of a test on a group of candidates
Module	A grouping, possibly nested, holding contents and that can give points when fully completed.
Content	A content is a coherent group of components. The contents are organized in a tree structure. The modules are branches and the contents are leaves. In other words, unlike modules, a content can’t contain modules or other contents.
Component	Its a small unit of what will be displayed to the end users. You must use components to build your content, components compose contents. Can be of multiple types, a markdown or an editor for example.

Hint	A hint is an item to help the user to finish a component in exchange for PolyPoints.
Item	Represents a purchasable resource.
Polypoints	A virtual money responsible of the gamification of Polycode. Polypoints can be used to buy items. The user and team leader boards are based on the number of Polypoints. Polypoints can be earned by completing contents or modules.
Runner	A service that executes code in a given language and returns the output (stdout and stderr).
Submittable	A component that can be submitted for validation by a user to the application.
Submission	A user's answer to a submittable. Can be validated through a validator.
Team	Group of users. One of them is the team's captain. The team's Polypoint count is the sum of the Polypoints of its users. Team are ranked according to their Polypoints.
Validator	Validate users submission.
Tag	Keyword associated to a resource used for statistics and filtering.
Purchase	Items that was bought by a user.

User stories

Now that we agree on the vocabulary, and that we clearly defined what are the meaning behind words in the scope of the Polycode application, we now need to define the features that needs to be available to all our users. This can be done using user stories. They define actions that someone should be able to do on the platform, giving context as why they want to do so. Defining the use case ensures we define features that makes sense, and to keep the end goal in sight.

Here are the user stories we came up with, based on our client specifications:

As a	I want to	In order to
Guest	Create my account	Use the application functionalities
User	Update my email	Recover my account if I was to change my email address
User	Reset my password using my email	Recover my account if I was to forget my password
User	Receive a welcome mail when I sign up	Be notified when my registration is completed
User	Show my account details	To review my personal information
User	Show other’s details	To gather information about another user
User	Change my username, my preferred programming language and my bio	Update personal information and settings if there was to change
User	Delete my account	Delete my fingerprint on the Polycode application
User	Connect to my account using my username and my password	Access the application’s functionalities
User	Logout of my account	Avoid non authorized access from a computer I used
User	Create a new team	Gather a user group and participate to teams leader board
Captain	Invite other users in my team	Grow my team
Captain	Delete a member from my team	Remove a problematic member, for whatever reasons
Captain	Give the captain role to another member of my team	Dispose of my role

Captain	Delete my team	Delete this team's fingerprint on Polycode, for whatever reasons
Captain	Change the name and the description of my team	Keeping the team's infos up to date
User	Accept or decline a team invite	Join the team I want to join
User	Leave a team	Not being associated to a certain group of user
User	See my team's points	Follow my team progress
User	See the teams leader board	See my team placement
User	See the internal ranking of my team's members	See who participates the most in the team
User	See the list of available exercises	Choose an exercise
User	See the list of available modules	Choose a module
User	See the sub-modules and exercises of a module	See the steps needed to complete the module
User	See the list of available evaluations	Choose an evaluation to pass
User	See the latest exercises / modules posted online	See the new content
User	See the information of an exercise	Get information on the subject of an exercise
User	See the information of a module	Get information on the subject of the module, the objective
User	See the information of an evaluation	Get information on the subject of the evaluation, the objective
User	See the statement of an exercise	Know the problem to be solved and the questions to answer before validating the concept
User	Propose a solution to the exercise	Earn PolyPoints and progress in the associated module
User	In the case of code to be written, execute an intermediate validator	Check if my code is correct for the validator in question
User	Save the last solution that was the most successful	Take the code from a different device, at another time, to improve it

User	Write (and modify) my code solution in an editor integrated into the exercise page (code exercise case)	Propose a solution to the exercise
User	Add files to the editor	Organize my solution into multiple files
User	Delete files in the editor	Organize the solution into multiple files
User	Buy hints with PolyPoints	Better understand how to solve the exercise
User	Follow my progress in each module	See what I completed, and what's still in progress
User	See the global leader board users	Gain motivation to reach the top
User	Take an evaluation	Get a certification
User	Read the content of a course	Become competent on a subject
Content Creator	Create an exercise, add markdown, a code editor, a QCM	Teach a new concept, check the knowledge of this concept with a question / code to be written
Content Creator	Create a module	Organize exercises by major concept / theme
Assessment Creator	Create an test	check the skill set of a user on something
Content Creator	Add my exercises to a module I have created	Fill the content of a module with a coherent and organized set of elements
Content Creator	Add submodules to a module	Fill the content of a module with a coherent and organized set of elements
Content Creator	Delete a content I created	Fix a mistake or delete the existence of this content
Content Creator	Delete a module I created	Fix a mistake or delete the existence of this module
Assessment Creator	Delete a test I created	Fix a mistake or delete the existence of this test
Assessment Creator	See the results of my test	Mark a candidate
Admin	Promote a user to a creator	Add a new creator
Admin	Promote a user to an admin	Add a new admin
Admin	Fetch data of a user	See the personal information of a user

Admin	Update user’s data	Help recover and fix problems with a user account
Admin	Delete a user	Remove access of a user
Admin	Have a content creator or assessment creator permissions	Edit and manage existing content, if the need arise
Admin	Have a user permissions	Use the site as a normal user
Assessment Creator	Create a campaign	Test a set of candidates
Assessment Creator	Add candidates via a web interface	Add candidates to my campaign
Assessment Creator	Delete candidates via a web interface	Delete candidates to my campaign
Assessment Creator	Add candidates via a web API	Add candidates to my campaign
Assessment Creator	Delete candidates via a web API	Delete candidates to my campaign
Assessment Creator	Add candidates via a CSV file	Add candidates to my campaign
Assessment Creator	See results and stats of a campaign I created	Evaluate the candidates level
Assessment Creator	Add tags to candidates	Group candidates
Assessment Creator	Define a expiration time for my campaign	Close my campaign at a fixed time
Candidate	Come back to a test, without losing my progress	Finish my test if I were to leave the app
Assessment Creator	Define a limited time for each of my questions	Restrain the maximum time a candidate has for a question
Assessment Creator	Define the points a question can give	Score candidates
Candidate	Receive a mail to opt-in the campaign	Join or decline a campaign
Candidate	Accept a campaign	Participate in a campaign
Candidate	Decline a campaign	Not participate in a campaign, and notify the creator
Assessment Creator	Edit my campaign	Fix mistakes or edit its content
Assessment Creator	Define a beginning date for my campaign	Time frame my campaign
Assessment creator	Manually send begin campaign mail to candidates	Customize the campaign access
Candidate	Receive a email when I finished passing my test	Get a confirmation that my answers was taken into account

Assessment Creator	Visualize the results by tag in a graph	Better visualize results
Assessment Creator	Export summed up results as PDF	Save results in a synthetic way
Assessment Creator	Export detailed results as PDS	Save results and have them available locally
Assessment Creator	Compare candidates results via a excel document	Compare candidates
Assessment Creator	Sort candidates by tags or results	Compare candidates using the criteria I want

Domain Driven Design (DDD)

Great! We now have defined a common language around the Polycode domain, and we also have explicitly defined the features and expected user interaction with our app. We now have a solid foundation to start thinking about the architecture of our application. In order to make this goal more achievable, and to actually come up with a sensible approach, we are going to design our architecture based on the functional needs of our applications. Starting high up in the mental model of relations between components of our application, diving deeper and deeper until we come up with an actual microservice architecture. This is called Domain Driven Design.

I will introduce the vocabulary linked to DDD when needed. Here's the important ones that you should now before going any further with DDD:

- Domain: The sphere of knowledge or activity related to a particular subject or business. In the context of software development, the domain is the area of expertise or focus of the system being developed.
- Domain model: A representation of the key concepts, relationships, and rules within a domain. A domain model is used to capture and communicate the understanding of the domain to both domain experts and developers.

The origin

Domain-driven design (DDD) is a software development approach that focuses on the design and development of a system from the perspective of the business domain it operates in. It was first proposed by Eric Evans in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software* which was published in 2003. DDD is based on the idea that by better understanding the business domain and the underlying processes, it is possible to design and develop more effective software systems that can help organizations operate more efficiently and effectively. In his book, Eric Evans defines key component around this approach, as well as identifying and naming common patterns you often encounter in complex, business-logic-heavy domains. This book triggered several engineers and practitioners in this field, to add onto this idea. As you might have realized, the origin of DDD is not rooted within the microservice world. It is a software design approach that can be applied more broadly, when developing software, although it suits particularly well our microservice architecture.

What is the goal

Why should we design our application this way ? Why not just dive right into developing microservices, creating new one every time it *feels* like the right choice ?

Your intuition might be right, but several problems arise :

- Getting it right gets exponentially more difficult the more complex your domain is. Moreover, as mentioned in the introduction, if you have a simple architecture, microservice might not be the best fit for you.
- Even if you're right, you need to justify your choices. Intuition is not a justification.

- You need to document your architecture, and the steps you've taken to get to here. This is important for anyone joining your team, to truly understand the decisions that have been made, why the current state of this architecture is the way it is.
- If you're wrong, it's much more difficult to come back and rethink your design. You will have no base to work with. This won't be a revision of what you've made, it's starting from the ground up once again.

And if it looks too costly to fix your architecture or if you're too stubborn to realize that you've made a mistake, you will have to work around those errors, adding band-aid on top of band-aids, and you'll have a unreliable, under-performing, hard to maintain architecture. It is key to understand that the decisions you're making when designing your architecture will impact every line of code written.

The goal of domain-driven design (DDD) is to help developers create software that accurately reflects the business domain and can be easily understood and maintained by domain experts. By aligning the software with the needs and goals of the business, DDD helps ensuring that the software is able to effectively support the business and its operations.

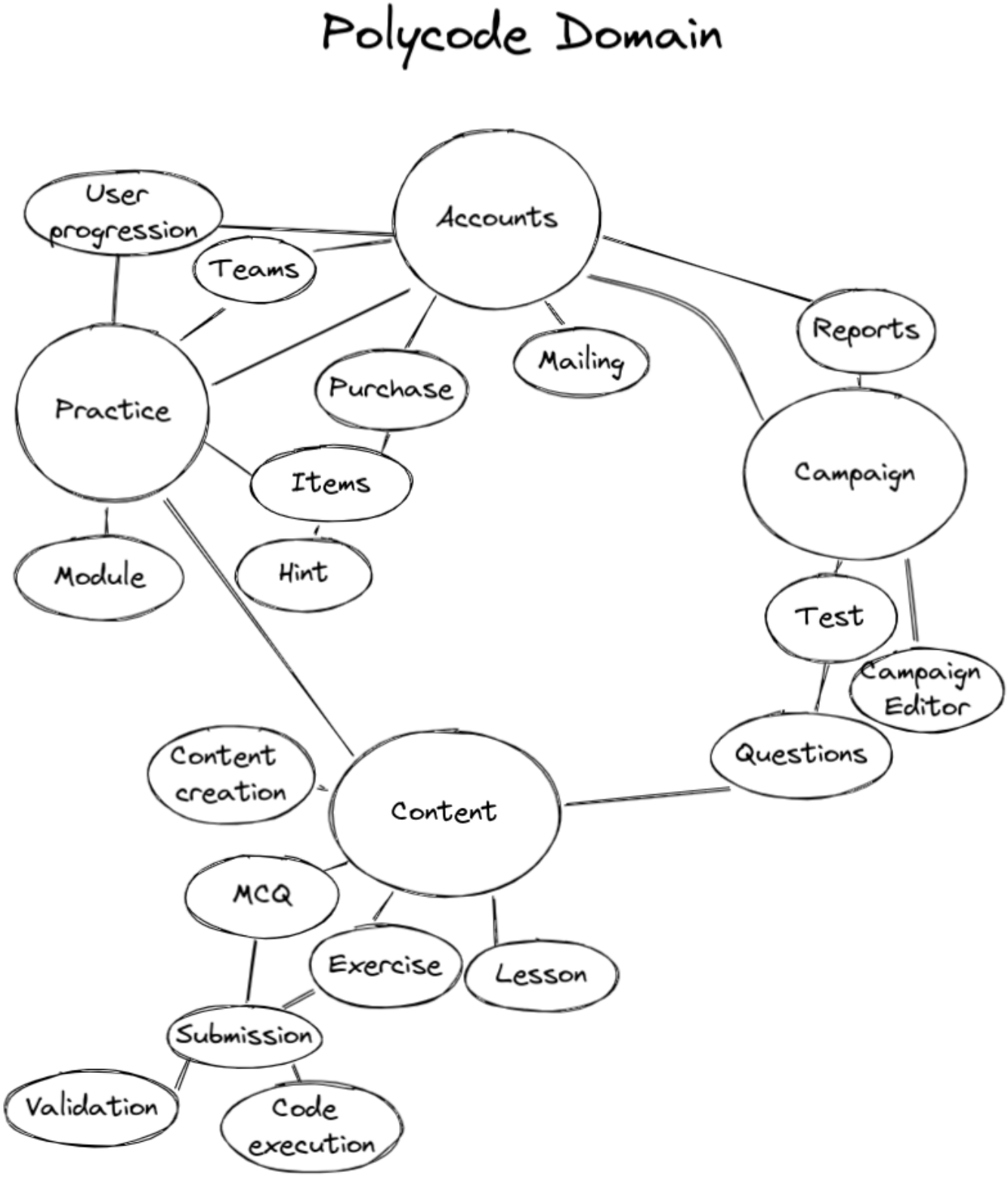
Microsoft

I wanted to highlight the work done by the Microsoft engineers in documenting and creating a microservice architecture. Their architecture documentation is a great resource, and I've learnt most of what I'm applying through it. As a matter of fact, I will apply their process for creating a microservice application using a DDD approach. I **highly** suggest reading their documentation here and here for a better understanding of DDD. We will walk trough each step in this chapter, but having a deeper understanding that the very high level overview that I'm going to give through this paper is worth it. Note that, while going much deeper, the Microsoft documentation is not the bible of DDD, and you might want to read *Domain-Driven Design: Tackling Complexity in the Heart of Software* instead. Microsoft goes much more into details about where using DDD to define your architecture makes sense. There is a lot to say about DDD, and this is misaligned with the core goal of this chapter, which is to explain the thought-process I had when architecting Polycode.

Polycode Domain

Domain Driven Design starts high up, with domain experts, and progressively dive in deeper until we have a clear architecture. But what are domain experts ? What is our domain exactly ? Let's start with the basics. Domain experts are the people that are knowledgeable in their field, and provide the necessary expertise to ensure that the software accurately reflects the real-world domain it is intended to model. But what is our domain ? Good thing for us is that we already did all the hard-work. We defined our ubiquitous language around Polycode, and our user-stories to define the functionalities. Talking with domain experts, we can plot all our vocabulary on a diagram to better understand all the parts around Polycode. Here's what I came up with:

Polycode domains

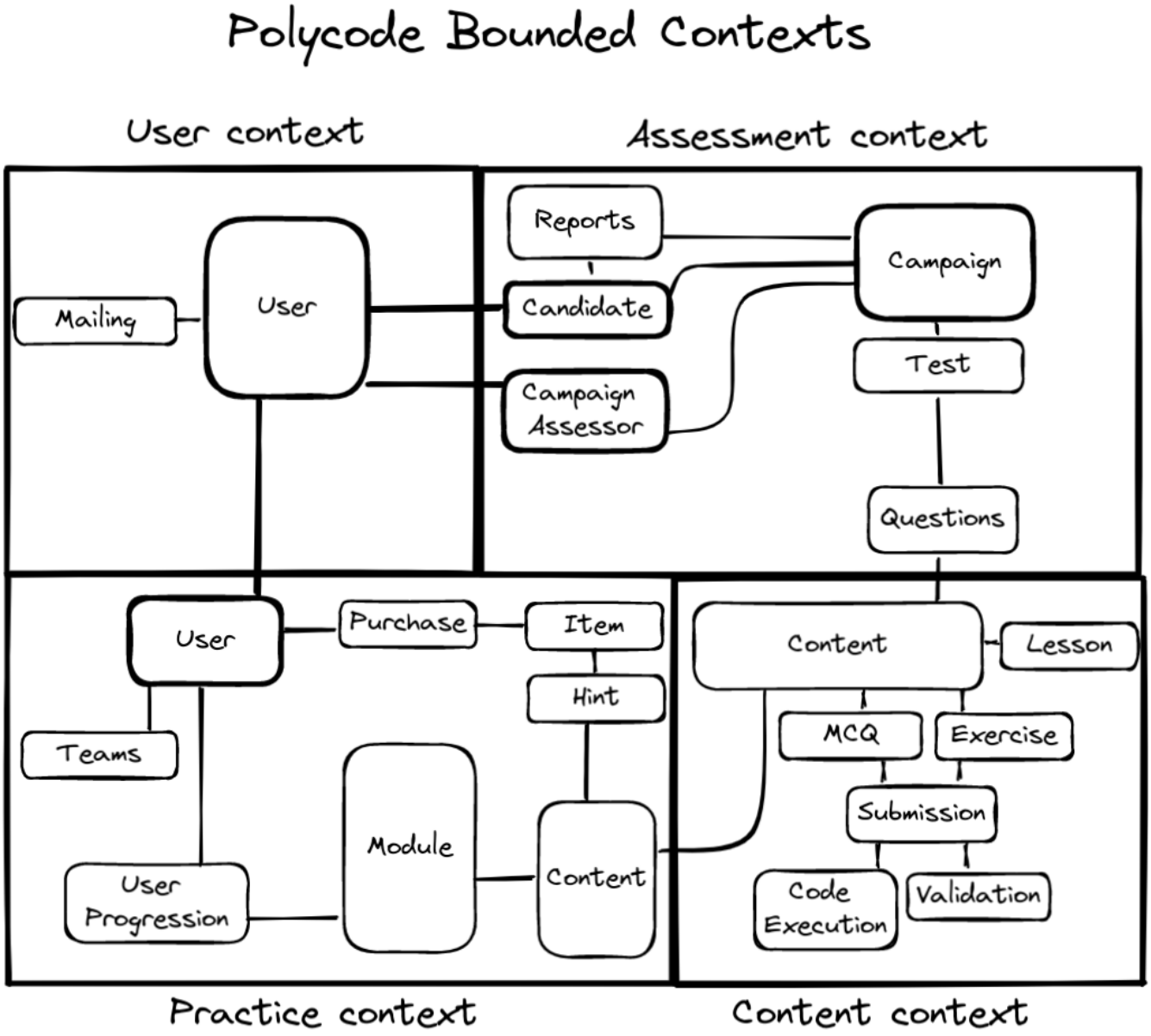


This diagram gives an overview of relations between concepts inside our Polycode domain. This is already quite a lot to work with ! It might look frightening at first, manufacturing a great architecture out of these seemingly abstract vocabulary might be concerning. Don’t be, DDD will guide us.

Bounded Contexts

If you've read the Microsoft documentation or if you've had some experience with DDD, you'll know the next step is to define bounded contexts. A bounded context is a specific way of defining the boundaries and scope of a domain. It is a way of partitioning the domain into smaller, more manageable pieces that can be developed and understood independently, but still fit within the overall context of the domain. The goal of a bounded context is to create a clear separation between different parts of the domain and to explicitly define the ways in which they interact with one another. We are going to divide our Polycode domain in a subset of contexts, that can be worked on independently. I'll give my take on the Polycode bounded contexts, and explain why each of them makes sense:

Polycode bounded contexts



As you can see, I've divided the domain in 4 bounded contexts:

- The user context, which I also might refer as the account context, is the context where we have a deep knowledge about what a user is. This is where we handle his settings, account management, authentication. This is also the context responsible of sending emails, since an

email is intrinsically linked to users.

- The practice context is responsible of handling the practice side of Polycode. This is where we have knowledge about teams and items, where we handle user progression and defines modules that the user will be able to follow. It is important to note that the notion of user in the practice context is not the same as the notion of user in the user context. A user, in the practice context, does not have a password or even an email. It is not needed in this context. This means that there will be translation layer in the communication between contexts, and this is normal. This is the cost we have to pay to make each of our context easy to work with and self-contained.
- The assessment context, in the same regards that the practice context, is responsible of handling the assessment side of Polycode. Separating it from the practice context makes sense to me, since we are handling a very different business logic. We need to grade candidates, with an invitation system on test that have a limit in time and that you can't retry.
- The content context. This is the harder one to justify, and it might look like I've let the technical implementation and details take over my thought process. I don't think this is the case. Wether you are in the practice or assessment context, you're not really concerned about the content itself, but more about the functionalities around it. You're concerned about what is a module, and that it has content within it, but whatever is the content. What is important to you is "Did the user finish this content ?" for example. In the content context, we don't really care where the content actually is. What we care about is inside this content, can we validate it, can we execute it ? How can we create new contents ? Contents exists in the assessment (as a question) and practice context, but they have a very different purpose than in the content context.

The great thing with DDD, is that it can be an iterative process. If you realize that you've made a mistake, you can always come back to the previous step and fix it. And since we're dividing our domain into contexts very early, if we were to come to make a mistake, it should not impact the other bounded contexts.

Now that we have defined our bounded contexts, let's move on to the next step.

Tactical DDD

We have defined our bounded contexts, and successfully divided Polycode into smaller, workable domains. This is great, from now on we can move in parallel, with domain experts for each of our bounded contexts identifying the key elements within each of our contexts. But our bounded contexts is still way too broad, we still can't jump to microservices directly. We need to define our domain model with more precision. To do so, we will be using tactical DDD.

The first step is to identify and categorize all the area of functionalities you have. Tactical DDD defines the following categories :

- Entity: A domain object that is defined by its identity and attributes, rather than its behavior. Entities are typically used to represent things or concepts in the domain that have a long-lived existence and are subject to change over time.
- Value object: A domain object that is defined by its attributes, rather than its identity. Value objects are typically used to represent things or concepts in the domain that are immutable and do not have a long-lived existence.
- Aggregate: A group of related domain objects that are treated as a single unit. An aggregate is used to enforce consistency and to ensure that certain invariants are maintained within the system.
- Domain Service: A domain object that is defined by its behavior and is not tied to a specific entity or value object. Services are used to encapsulate business logic that is not closely related to a particular domain object.
- Domain events: A domain object that conveys information when something happens in the domain. This is useful to synchronize microservice, since they usually don't share data stores.
- Application Service: An object that is needed for the domain to work, but is not directly related to the domain. This is a SMS Message service for 2 Factor authentication for example.

Once again, for a more detailed description of each of them, a highly recommend reading the Microsoft's documentation.

With the terminology set, we now need to analyze each of our bounded contexts, and come up with a well-defined and precise domain model.

The user context

Using the terminology above, I've divided the user bounded context and identified the following entities:

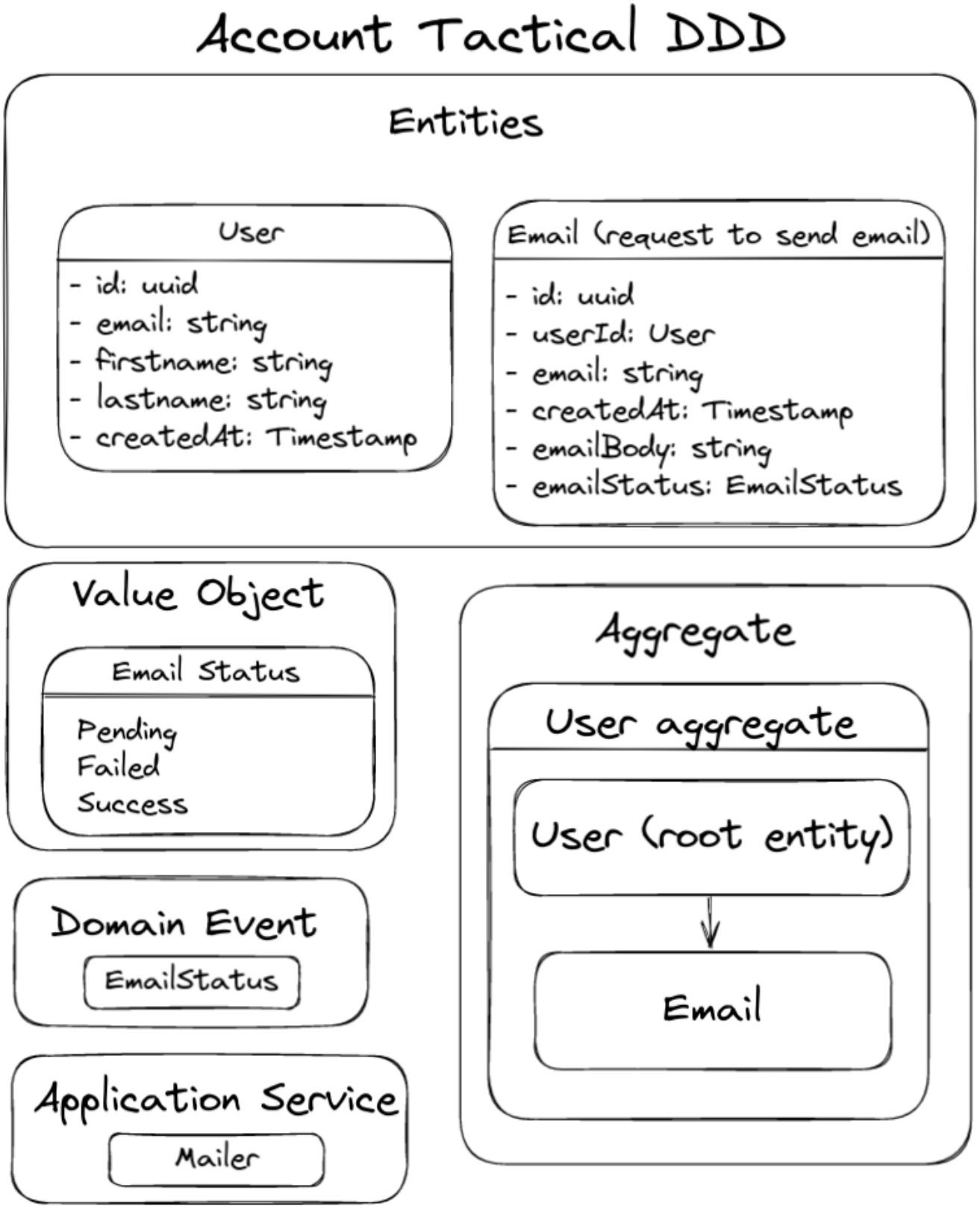


Figure 1. Account entities

This is a pretty simple context, not much to define here. Let's go over what I came up with.

As entities, we have the User entity and the Email entity. The user defines the basics of what is a user, in the account bounded context. A user should have a name, an email, and a unique identifier. We don't need to define anything else in this bounded context.

I've also defined a Email entity. This entity is here to represent and store emails sent to users (validation emails for example). It is important to note that there is a reference to its corresponding

user, via a userId. I'm also storing the destination email. By doing so, we make sure that we know the destination email at the time the email was sent. A user might change its email, and if we rely on the User entity, we lose this information.

The User is also an aggregate, composed of the user entity as the root, and the email entity as children of user. We want consistency between our user and email, especially in this context, since we are dealing with user data. We want to make sure that when deleting a user, all its related sent email are also deleted. This is important to respect user data privacy, and to comply with GDPR.

I've defined the Email Status value object, which represents the status of an email. The email status of an email should only be modified via the aggregate.

There is a need for an application service. We want to send mail, so we need a service that can actually send mail. This is not tied to the business logic itself, and is here to provide technical functionality.

Finally, I've also defined the Email Status domain event, which will notify the domain whenever a email changes status. This event should be sent when a email is created, when it has successfully been sent, or when it failed.

The practice context

Let's take a look at my take on the domain model for the practice bounded context:

Entities and value objects:

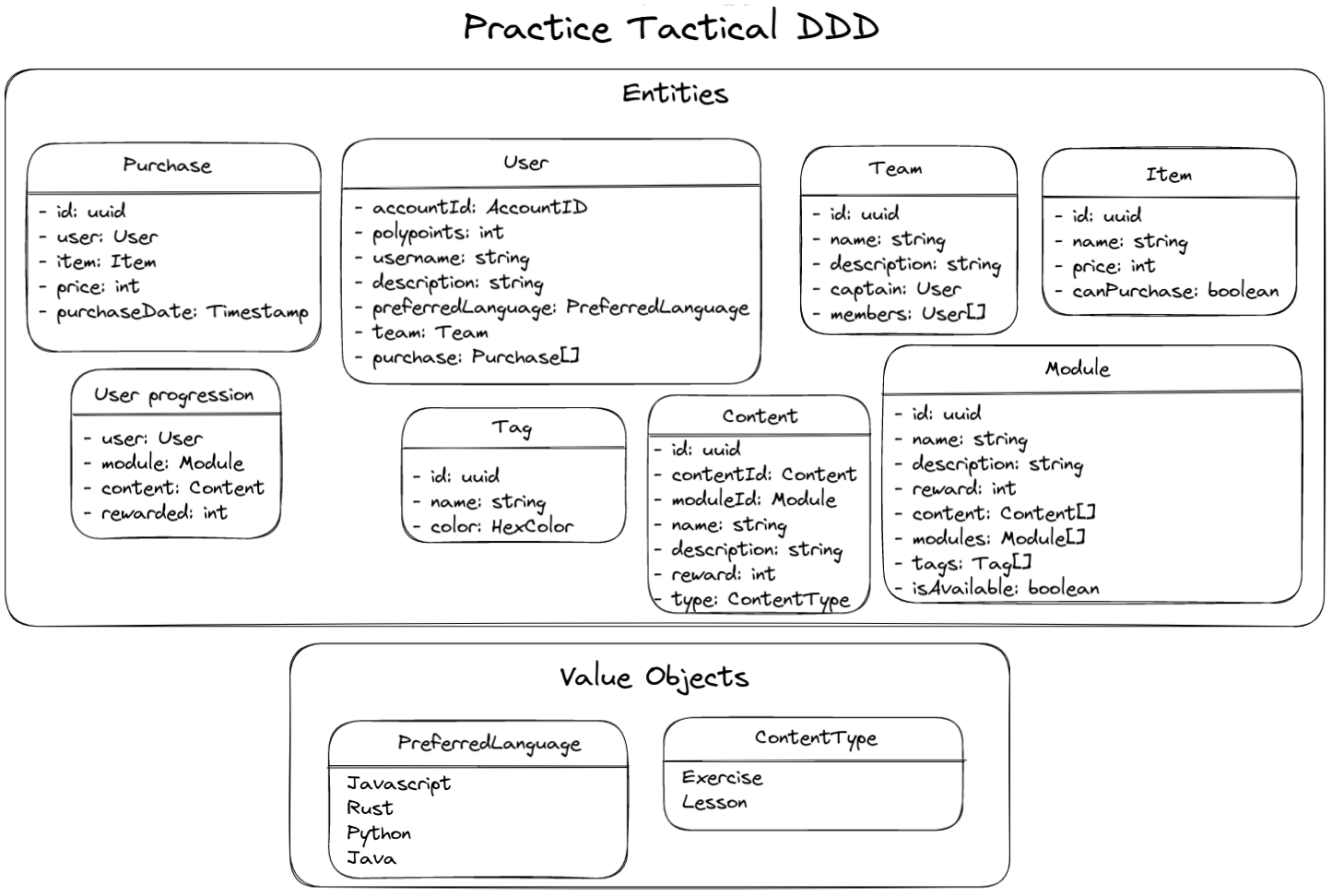


Figure 2. Practice entities and values objects

Aggregates:

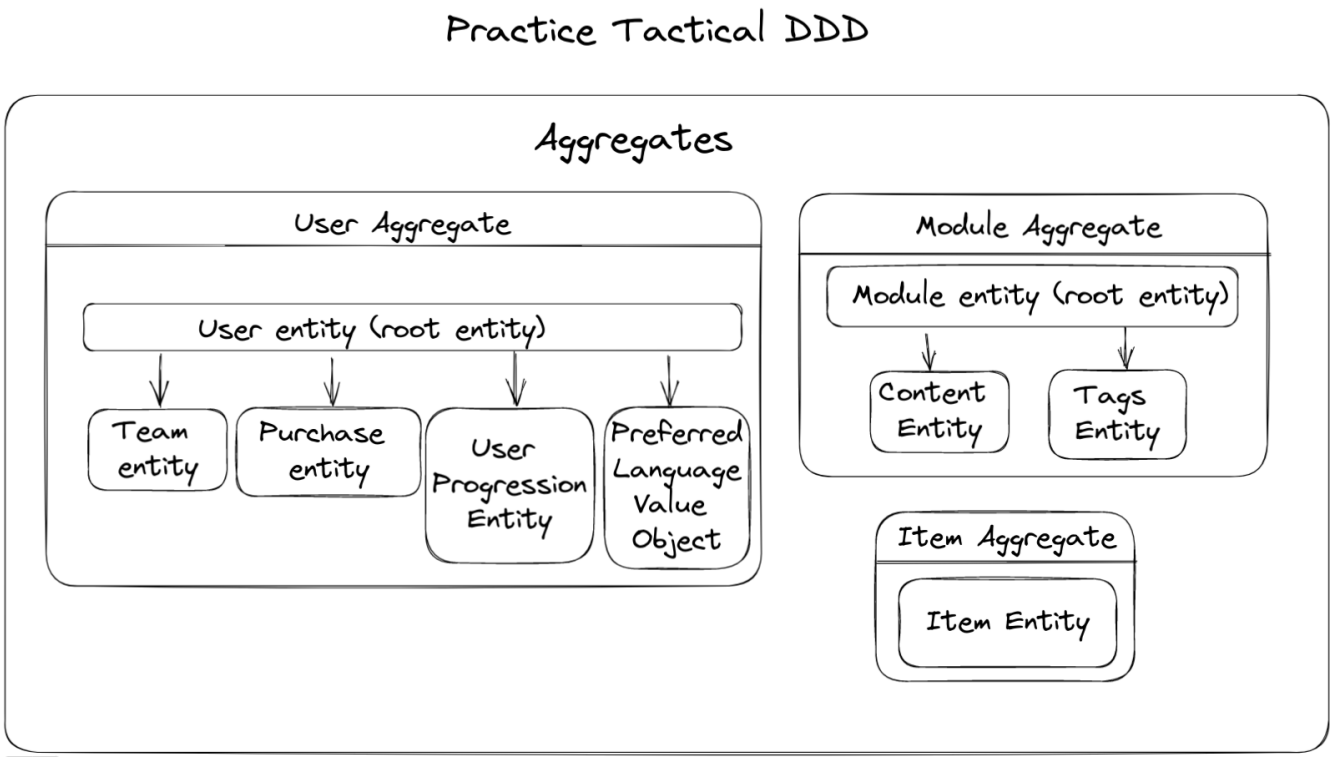


Figure 3. Practice aggregates

As you can see, I've defined the User entity, which is a different entity than the one in the Account context. However, the field `accountId` makes the link between the two. There is additional information that is relevant only within the practice context, and that's why there is a separate entity in it. User is also an aggregate, composed of itself as the root entity. It is linked to UserProgression, Team and Purchase, since I have identified a need for transactional consistency. In the case of Purchase, there is a need for consistency when the user buys an item. We need to make sure that he has enough polypoints, and the transaction should fail if there is an error in either updating its polypoints, or an error when registering the Purchase. There is a similar need for the user progression. Registering user progress and adding the corresponding rewards to its polypoints should be done in one transaction.

This is a great example of what aggregates are for. You don't want to be able to update Purchase without having the correlated side-effects. This is usually taken care of your database in a monolith application, where everything is done within one transaction. However, in a microservice architecture, it's the application layer that has this responsibility.

This is the same case for a Team. When updating or adding a new User to a team, we have consistency to be checked. The user must not be part of another Team for example. And when deleting a user, we have to make sure to stay consistent in the related entities, such as the Team the user might have been a part of. If the user was the captain, we need to elect a new captain for the team.

However, this kind of consistency check is not needed when we are not mutating the data. This is where patterns like CQRS shines, and allows for a better segmentation of your data access, and allow a much needed optimization in the case where your requirements for the command model and the query model are significantly different. We are not allowed to use CQRS in this paper, so I will not dive further into it.

I've defined Item as an entity, which is also an aggregate with only itself as a child. This means, however, that there is no guaranteed consistency when updating or deleting an item and its reference in the Purchase entity. If an item is deleted, purchases will have ghost references. But, as a domain expert, I would argue that an Item can never be deleted. Once you've proposed to your users an item, this item is here to stay. Its price can be updated, it can be purchasable or not, but once it is out, there is no going back. This constraint will need to be respected, because the User aggregate, and more specifically the Purchase entity, will make the assumption that an Item is never deleted.

I would now like to bring your focus to the Content entity. As we will discuss later, a Content is a totally different thing in the content context (content means a lot of different thing here, I'll try to make it clear). But in the practice context, we don't care about what is a content in the content context. We just now that OUR content has a name, description and a reward. An important realization to have, is that the content entities in the two contexts will not even be tied 1 to 1. Indeed, a content in the practice context is unique for each module. This allows us to re-use contents from the content context, while being able to have different names and description, with different rewards, depending on how it is used in the module. This makes the content tightly linked with its parent module, this is why I've defined the module aggregate, with the module entity as the root and the contents as child entities. We need to maintain consistency here, when a module is updated or deleted, we need to make sure to propagate the needed actions to the contents.

You might also have noticed that there is a link between content and modules, and the user progression. They are, however, not in the same aggregate. It's OK if we don't have consistency here, the main purpose of the user progression is to store how he got his polypoints. Ideally, we do not delete any content or module, and just set the unused or outdated modules as not available for the end user to see.

I've decided that tags should be an entity on its own, and not a value object. At first glance, it looks like a value object, it has no identity and is valuable only thanks to the value it owns. But in reality, tags will be used to categorize modules and to apply filters. This means that tags should be unique, and by extension identifiable, else we will have cohesion problems and redundancy in our databases. This is more of a technical decisions, but it is emanating from functional realities.

The assessment context

I've identified the following entities, services, value objects and aggregates for the assessment context:

Entities, value objects and domain services:

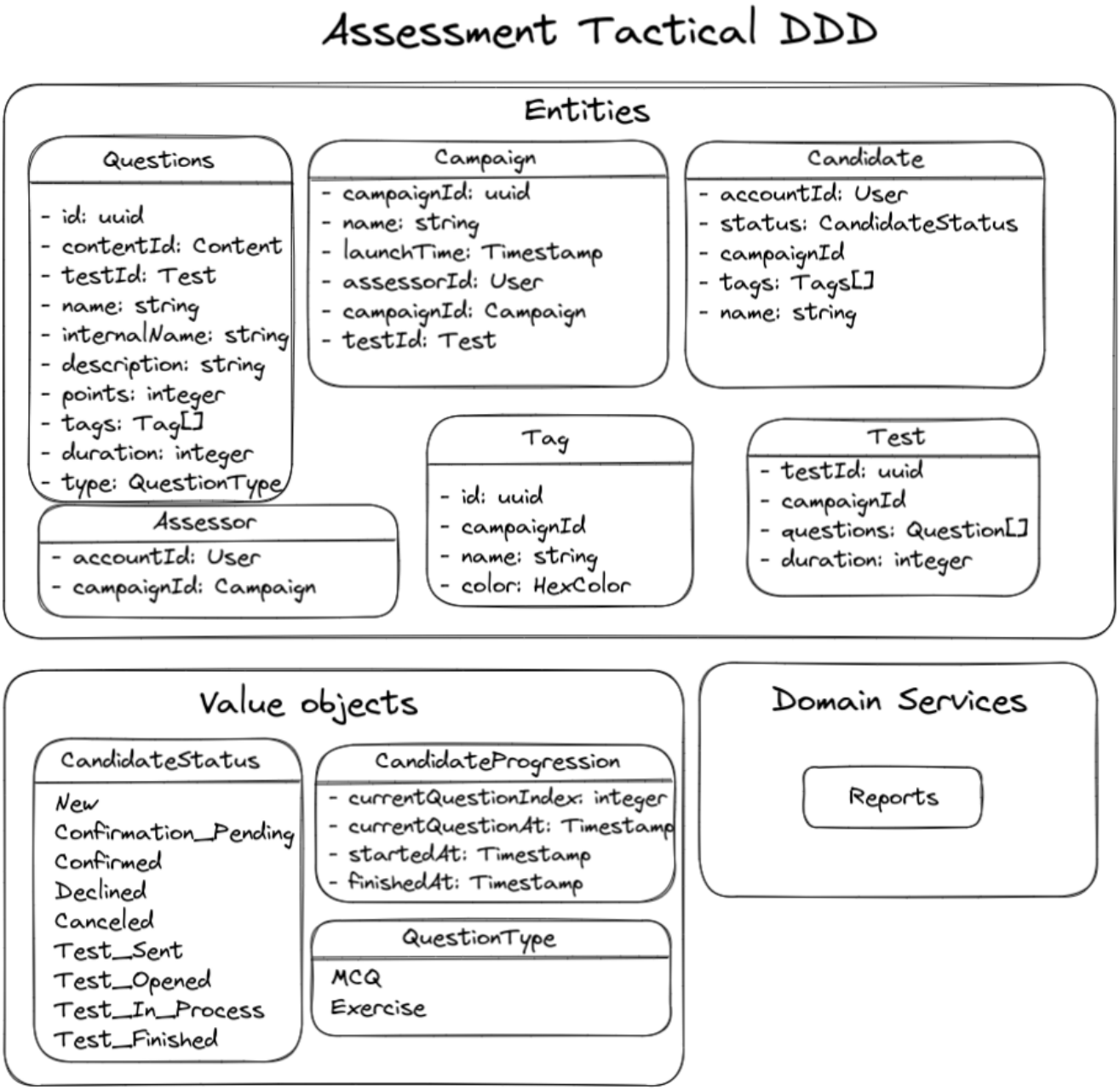


Figure 4. Assessment entities

There is quite a lot to unpack here. Let's take it step by step, beginning with entities. We have two types of users in our assessment context: assessors and candidates. Both are tied to a campaign, meaning that a candidate is not a candidate anymore if the campaign doesn't exist. Same thing with an assessor. This is quite logical, you're not an employee if your company doesn't exist anymore, this is the same logic. This also means there is a different dynamic of dependencies and transactional boundaries than in the practice context. This is a hint meaning that we probably divided these two bounded contexts in a sensible manner. Candidates have a status, which is an enumeration of value it can have. This has no identity, this is why CandidateStatus is a value object. As seen in the user stories, we need to send quite a lot of mails to our candidates. Even if this is a domain functionality, we delegated this part to our account context, which is why you don't see it appearing here. This will be an area of inter-domain communication.

Each campaign has a single test, which is an array of questions with an overall maximum duration. Since each test is tied to only one campaign, this looks like a 1 to 1 relationship, which is commonly

simplified by putting the fields of Test inside Campaign. I've decided against it, to better encapsulate behaviors, and to give room for expansion of functionalities in the future of the application. A question, is a specialized type of content. This works just like the content from the practice context, with the main difference being the fields describing it. We no longer have rewards but points, we have both a name and an internal name, we have a maximum duration, etc.. Just like in the practice context, we don't really care about what actually is inside our content, how it is going to be validated and displayed. We care about the points it gives. However, unlike specified, we don't have a way do have a fine-grained points distribution. We will look at how we can solve this problem in the content context, where the validation is actually done.

We also have a set of tags associated to the campaign. The assessor can manage this tags and set them to candidates and questions. It is important to have them identified, since we want to apply filters on them, and to find them easily. The business flow would look like this:

- Assessor creates or updates a tag
- Assessor adds a question to a test
- Assessor adds the tag previously created to the question
- Assessor updates the tag of a candidate with the previously created tag

We don't want the assessor to manually type in a tag for each of the questions, as it would introduce inconsistency, under-performing queries and most importantly, a bad user experience.

Just like with the content type withing the practice context, we also have a "helper" field to better identify what type of question it is, in the form of the value object QuestionType. Lastly for the value objects, I've defined Candidate Progression, which holds the current progression of a candidate. It has no identity by itself, and is here to define the current progression of a candidate within the campaign's test.

I have identified one domain service: reports. This service will be responsible for generating the data for each campaign. It might be interesting to create some already computed statistics about a campaign once it is finished, and store it in an entity, but given the most likely low volume of candidates and campaigns, I think it is better for now to just compute the data every time.

Aggregates:

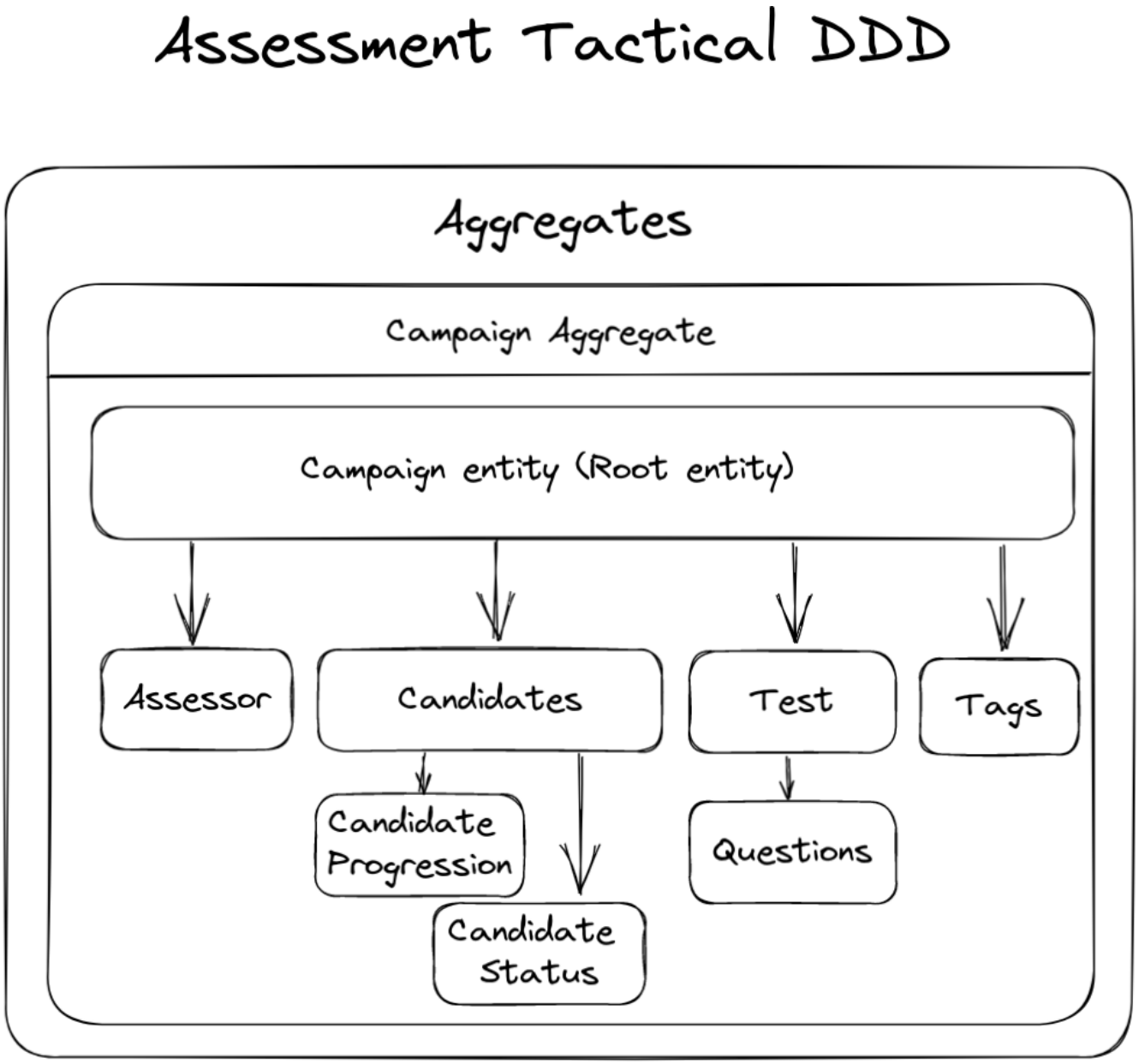


Figure 5. Assessment aggregates

I've defined only one aggregate, the campaign aggregate. As discuss previously, everything in the assessment bounded context is tied to a campaign, with a need for transactional consistency between all of them. The campaign entity will act like the gateway for all the children entities, making sure to take appropriate actions for each modifications and rolling back failed actions. This is a big aggregate and we will see how to deal with it later on. This is not a problem though, you have to accept that some transactional boundaries spans around a lot in your context. You might want to identify which of your dependencies forces you into this situation, and if there is something to be rethought, but for our assessment context, it's fine, not that big, and I don't think I misidentified links between my entities.

The content context

Finally, our final context. We have taken a look at all our bounded contexts but one: the content context. Let's dive right into it.

Entities, values objects and domain services:

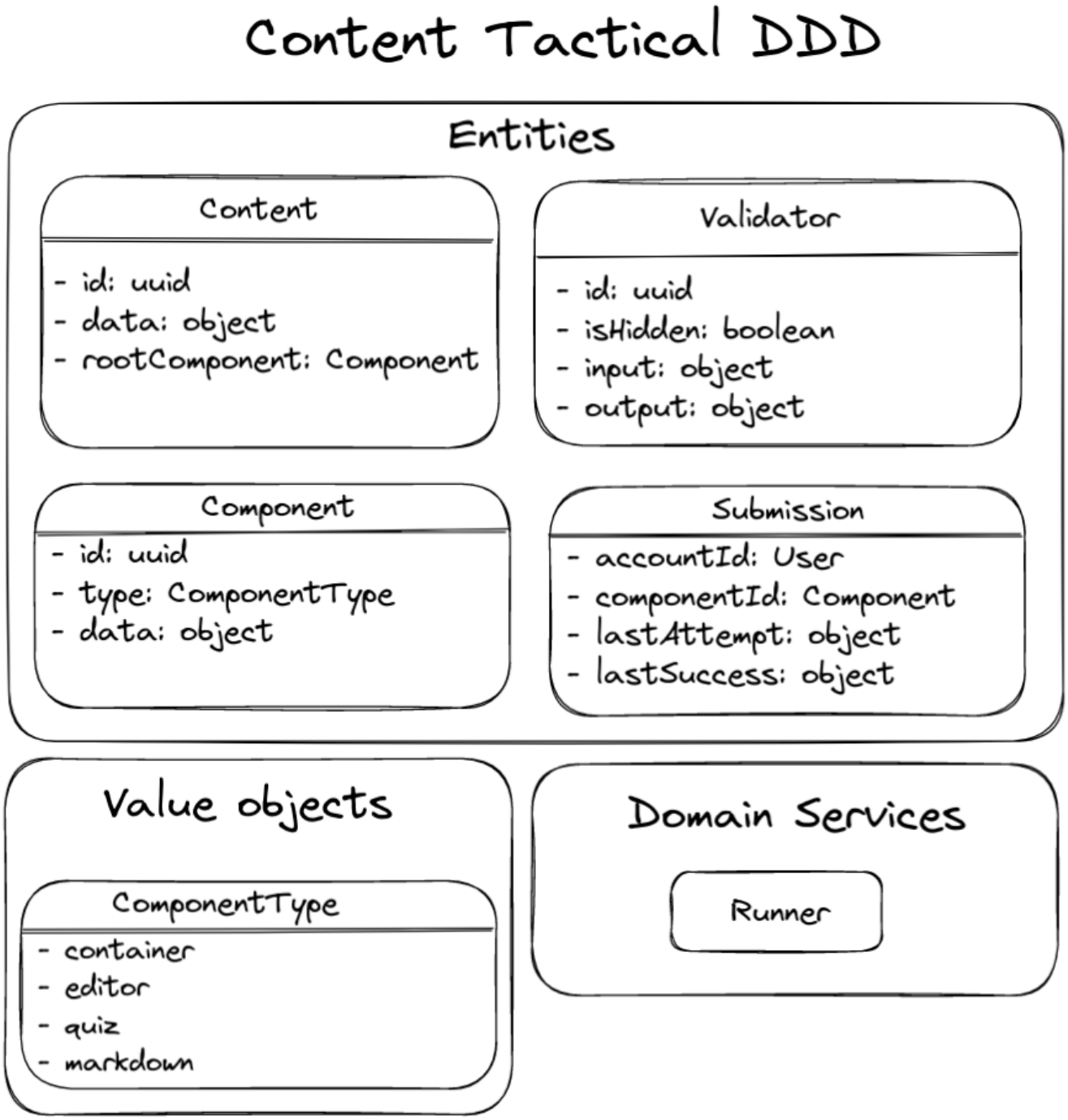


Figure 6. Content entities

I've defined 4 different entities. Let's start by talking about content.

Content is an entity we found in both the practice context (with its own content entity) and in the assessment context (the question entity). However, we've seen a sharp difference between those two already. In this context, content does not mean the same thing as in the previous two contexts. Here, we care about how it works, how it is going to be displayed, and we don't care about the name it or the type it is given in other contexts.

As you can see, a content is actually made of a root component, which itself might be composed of other component, or some text, or an editor. There is always a lot of possibilities, and these possibilities are not fixed in time. We need to have a very flexible way of modeling things, even if we lose in rigidness and gain complexity. This is why most of the entities have fields of value `object`. This means that we don't know the actual data structure it may holds. This can be inferred via the

type for example. Or it might be some edge cases where data that is totally unrelated to our context that other context might want to save, if they can't store it in their own context.

This is the same thing with validator. We don't know how exactly we are going to validate each components, it depends of its type and the data structure might evolve over time. We find the same thing with submission. Submission holds a reference to an account, but this is not an entity by itself. A user is just a uuid to somewhere else for us, we have no business with its email or its name. We just need to have a unique identifier for a unique user, so we are going to use the GUID of our users in our systems, the one emanating from the account context. Note that we are not even going to communicate with the account context, since in practice, this will be given to us at the moment of a submission.

Having all those objects within our contexts is fine, since all the business logic related to it is kept inside our bounded contexts. Our "external API" should not change, and that's what the other bounded contexts really care about. As discussed previously, they don't need neither want to know how our content will be displayed.

I've defined a domain service: runner. It will be responsible for taking executable code, and running it, returning the results. It does not do any validation, it just executes code. Since this is at the core of the Polycode application, and a topic on its own, I will not dive into details about its implementation and inner-working here.

Let's now look at the aggregates I identified:

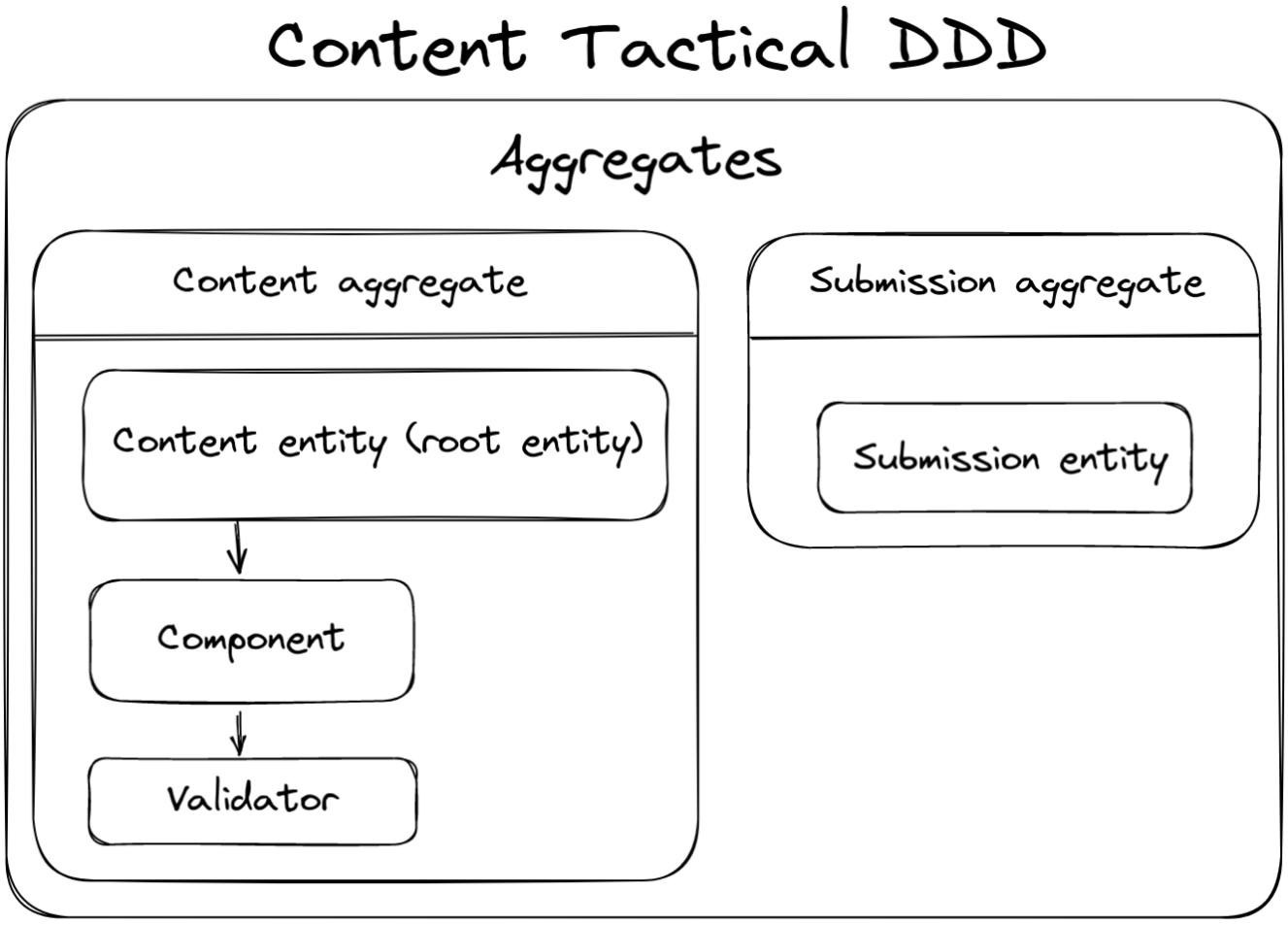


Figure 7. Content aggregates

I’ve defined two aggregates:

- The submission aggregate, which has the submission entity at its root. It is linked to a component, but it is not in the same aggregate as components. This is because, as seen before, we don’t need consistency here, and we don’t even want it. If the component were to be deleted, we want to keep track of all submissions that were linked to it, since users actually submitted something. Deleting a component doesn’t delete what the user did. However, this means that we have no guarantee of the existence of the component when parsing submissions, neither do we know if the component was updated since the submission.
- The content aggregate. Here, we need consistency. Indeed, when modifying or deleting a component, we want to update all contents and components accordingly. We don’t want to have contents with ghost components, and we must ensure this is not the case. Same things we components and validators. Validators are inherently linked to a component, and we need strong consistency, and check for business logic. An exercise must always have a validator linked to it, but a lesson should not. This consistency needs to be respected.

Microservices

This is the last stretch ! We have now defined a domain model for each of our bounded contexts, and have a clear understanding of where our boundaries are. We now need to translate these more or less abstract concepts into microservices.

From domain model to microservices

This section is heavily inspired of Microsoft's documentation on the subject.

The last step is to transition from our domain model to our application designs. During this process, don't be afraid to go back and revisit the previous steps, as you might see some problems emerge, that can be caused by a faulty domain model or bounded contexts.

The main criteria your microservices should respect are:

- Each microservice has a single responsibility
- Microservices should not be too chatty
- A microservice should not be too big for a small team
- No hard dependencies on one another. A microservice does not rely on another to be deployed and to work effectively.
- They can evolve independently
- You respect data consistency when there is a need for one

A microservice should not spread over your bounded contexts. If it does, you probably have a problem within your bounded contexts. As you might have realized, some of those criteria are closed to what an aggregate is trying to solve. This means that aggregates are often good starting points as microservices. When in doubt, you should always start with a bigger service, since it is easier to trim it down than to glue up multiple microservices together.

Polycode

Let's apply the previous criteria to our Polycode application. I will be starting by going over each bounded contexts, and we will talk about inter-contexts communication later.

Account microservices

Let's discuss how I mapped out my account microservices:

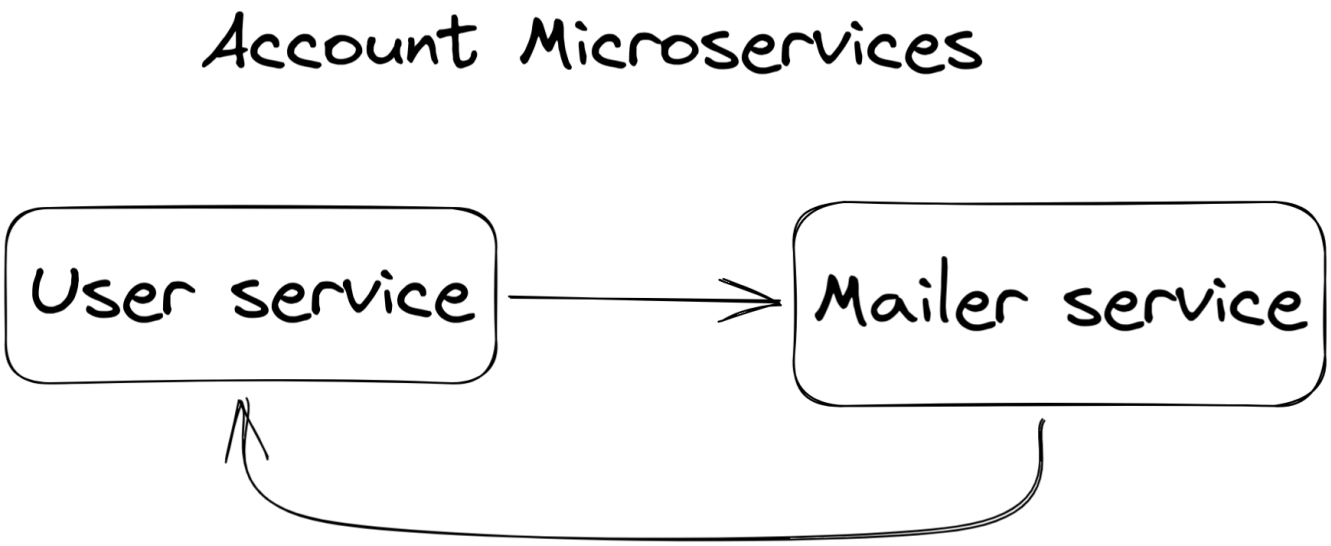


Figure 8. Account microservices

It is very simple and straightforward as you can see, since the domain model is very simple too. There are still some decisions that are not that obvious that were taken:

- The communication between the User service and the Mail service is synchronous, and so is the broadcasting of the EmailStatus Event. This is due to the constraint of not being able to use message queues, but this, to me, would be an obvious fit for a message queue, since we just want our mails to be eventually sent, and this can be done in a totally asynchronous manner.
- I've not divided the email entity within its own service. I think the cost of managing consistency between the two services overwhelms the benefits of splitting it in to microservices. The user microservice still has a single responsibility, which is to manage the users. Emails are part of this responsibility. The microservice is still sufficiently small to be handled by a small team.

As discussed previously, it is better to start with bigger microservices and trim them down later on, rather than refactoring them into one bigger microservice. If the needs arise, we should be able to create a new microservice for handling emails pretty easily.

Practice microservices

We identified 3 aggregates within our practice microservices: the user aggregate, the module aggregate and the item aggregate. I started by mapping each of them to one microservice, since this is where our transactional boundaries were identified. For the module and item microservices, I've estimated that they are small enough, and that they doesn't need to be divided. However, for our user microservice, which handled the user aggregates, encompassing user progression, teams and purchases entities, I think it is wiser to delegate some of the functionalities to some other microservices. Indeed, most of those entities are actually quite independent from each other, but also conveys a lot of business logic and intricacies with them.

Let's think about what are the trade-offs we make if we divides each of the sub-entities of the user aggregate in their own domain-driven microservices:

- Purchase entity:
    - Business logic closely related to the user entity, creating a new microservice for it might

carry a burden of maintenance and communication across multiple teams.

    - Technically simple (more or less just a CRUD). This makes it hard to justify spinning up a new runtime and new containers for this little of code. This might make more sense if we are edging towards serverless computing, but this is not our case neither within the scope of this paper.
    - Needs very strong consistency with the user entity when updated.
    - Not chatty at all with the user entity. Making a purchase is a rare event. The purchase microservice would not rely on the user microservice for any of its operations. The user microservice would rely on the purchase microservice only when mutating purchases of an user.

- Team entity:
    - Business logic related but not entirely tied to the user entity.
    - Technically pretty straight-forward, but has a significant amount of business logic to implement, with a growing list of features.
    - Not too chatty with the user entity. Mutating a team is a relatively rare event.
- User-progression entity
    - Business logic closely related to the user entity, creating a new microservice for it might carry a burden of maintenance and communication across multiple teams.
    - Technically simple (more or less just a CRUD)
    - Needs strong consistency. We don't want the user to complete something without crediting the correct number of polypoints to its account.
    - Not too chatty. Only chats when an user finished a content, which is not a rare event by itself, but I would argue it is still not on an order of magnitude high enough to become worrisome.

I didn't choose the purchase entity because I think it's too small and its logic integrates well enough with the user's business logic to go into the same microservice. The same thing goes for the user-progression entity.

However, I think creating a microservice responsible for handling team's business logic makes much more sense: it is quite unrelated from the user entity from a business logic point of view, while not being too chatty. Both of the microservices will grow independently and can be worked on by different teams. This will also shrink down the size of the user's microservice by a significant amount, making it easily manageable by a small team, while still integrating the features of the user-progression and the purchase entity.

Here's the final diagram for our practice context:

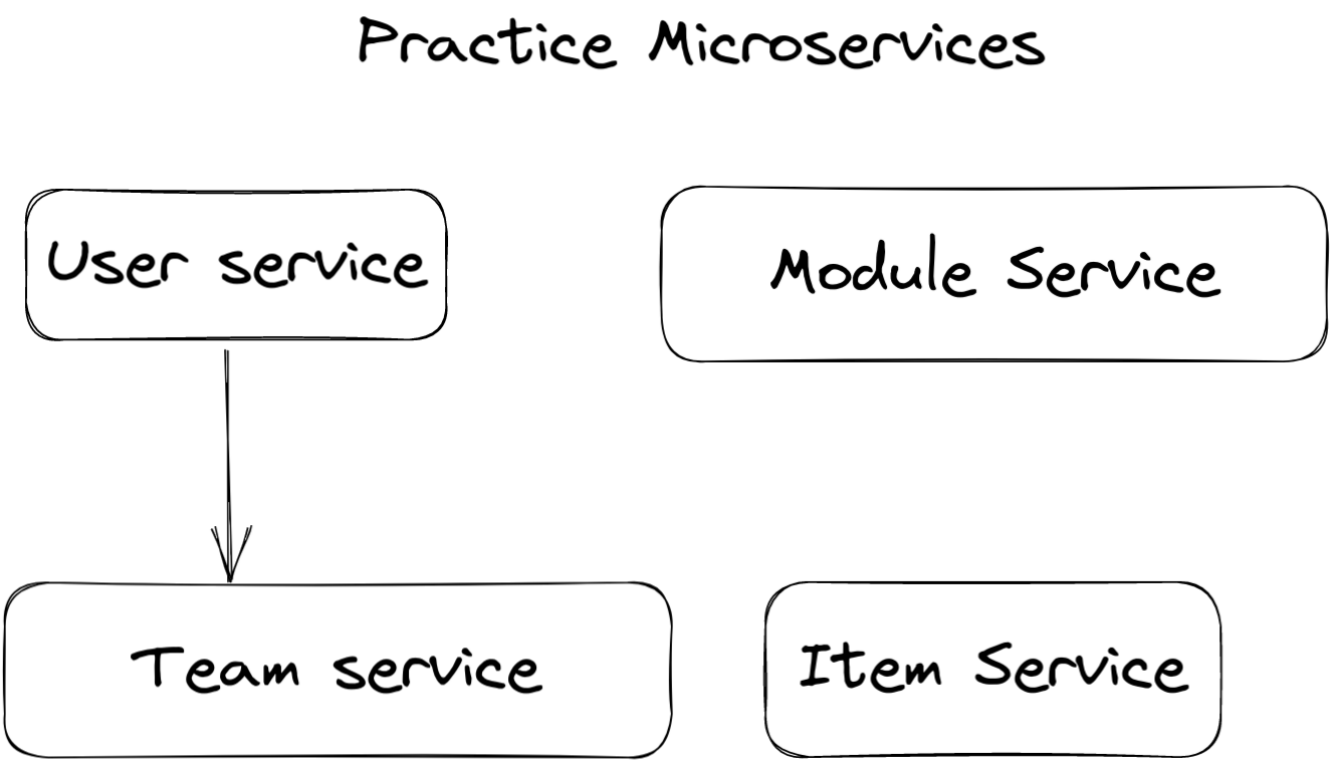


Figure 9. Practice microservices

Assessment microservices

As always, let's start with the aggregates identified in our domain model. In the assessment context, we found only one aggregate, the campaign aggregate, which encapsulates all our entities within our domain. As you might have realized, this is way to big for a small team to handle, and we need to identify where we can divide our microservice.

I'm going to keep the assessor and the tags entity within the campaign microservice, simply because they are way too small to be separated in their own service. They are both very simple in nature, with close to no business logic. They are not valid candidates for a microservice on their own. I'm applying the same line of thoughts as for the practice microservices.

We are left with the candidates and the test entities. Both are great candidates, since they handle logic that are quite decoupled from the campaign itself. This is why I've decided to create a microservice for both of them. Here's the diagram:

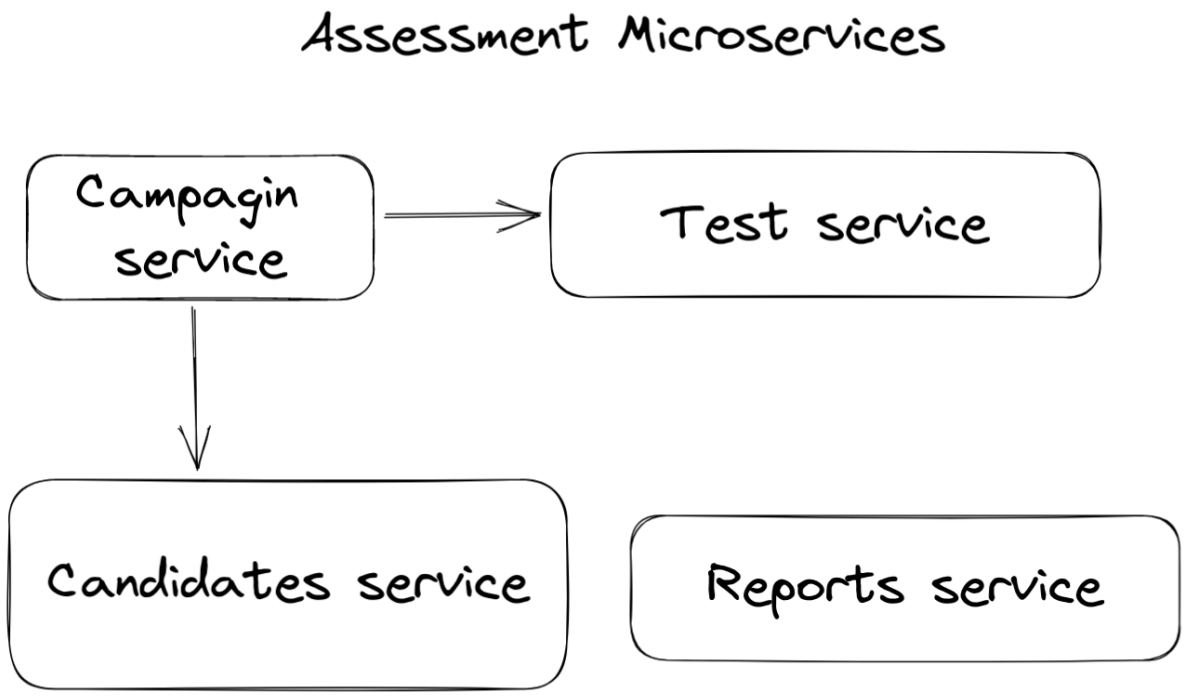


Figure 10. Assessment microservices

We also added our reports domain service as a microservice. It will be responsible for generating reports linked to a campaign. We are left with microservices that are small enough in size, but not too small. They are not too chatty, are not tightly coupled and can evolve independently.

Content microservices

Following the same logic as with the previous domains, we start from aggregates and we trim down each one of them, following some common sense and technical and business requirements:

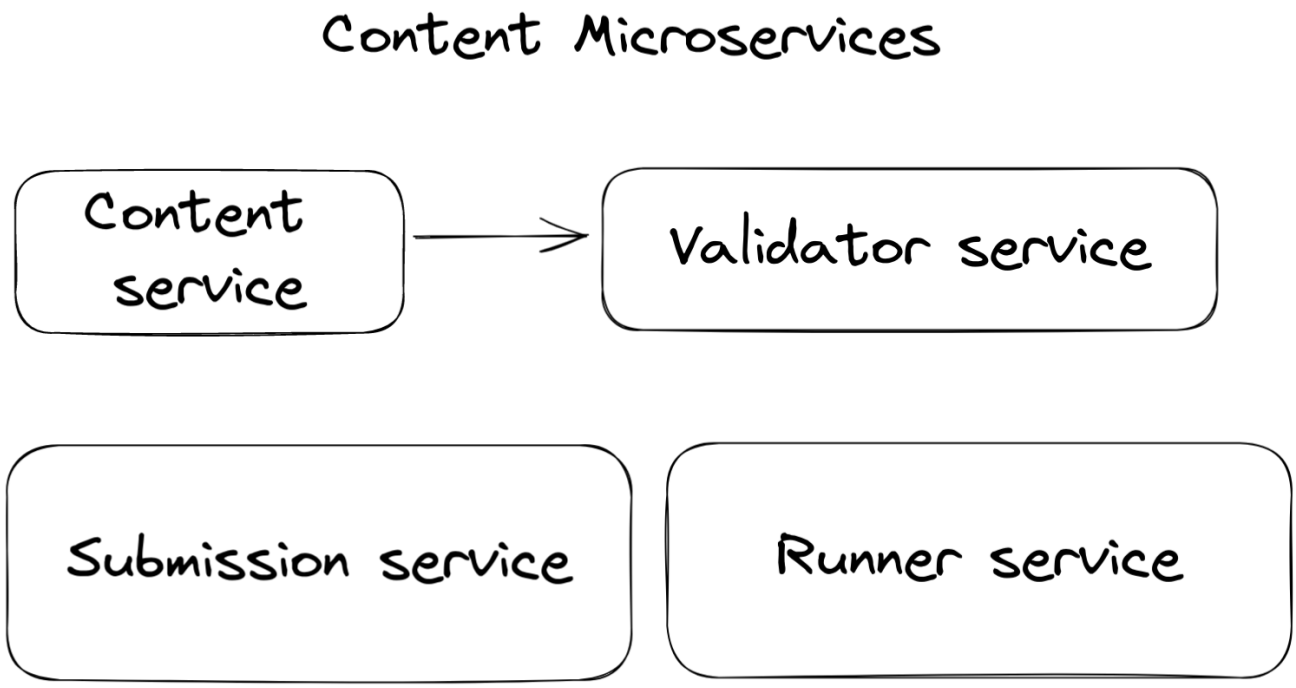


Figure 11. Content microservices

We can not divide our submission aggregate since it only has itself as entity. However, the content aggregate has component as a child entity, which itself has the validator entity as a child. I've decided to keep the content and component entity together, since they are very closely related, and content is nothing more than a composition of component, at least in this domain. They are very chatty, since every mutation and reads on a content will most likely trigger a similar event for one or multiple components. Keeping them together makes for a microservice of a reasonable size, that can be worked on by a small team, within the same logic. We can, and should, create a separate microservice for the validators. Validators might look like a simple CRUD at first glance, but actually is way more complicated than that, since it can take multiple shapes depending of the type of component being validated. I would argue that most of the complexity in this aggregate is within this entity. It is also separated enough for another team to work on, although a good communication between the teams responsible of creating the content microservice and validator microservice will be required.

Just like before, we must not forget our runner domain service, which will become a microservice by itself. I'm not going into any details here, since the runner part of the system will have its own separate section.

Putting it all together

Now that we have defined all our microservices within each of our bounded contexts, let's take a look at the greater picture of our architecture:

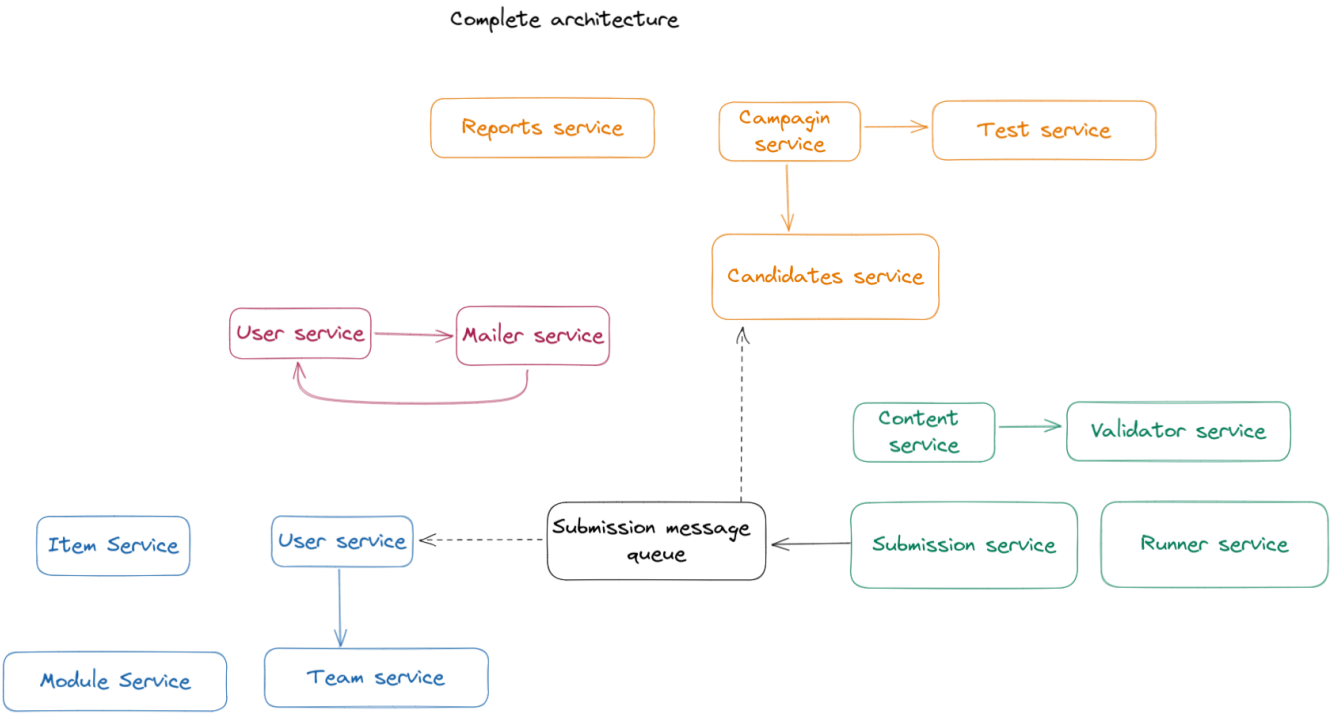


Figure 12. Complete architecture

I would argue that a system-wide event related to major user mutation emerging from the account domain, such as a user being deleted for example, is needed for the other domains to act within their boundaries to take the actions it deems needed. However, since we can't use message queues, I will not dive further into this idea, since it is the most sensible way to implement this idea.

We also need communication between the submission microservice and the candidates and user microservices, since submission are linked to the user and candidate progression. To do so, I would

also propose the use of message queues, where the submission service would publish messages whenever a new submission has been mutated, as it allows for a total decoupling of domains, and improve reliability since messages can be published even when they are no consumer able to handle the messages. It is the only way to do so without breaking boundaries. Any synchronous communication protocol would not fit this use case, so I will implement a message queue here, even if we are not allowed to do so. It makes too much sense to not use it. We will discuss this matter later in this paper.

There are multiple oversight, such as the runner service which is not as simple as it might look on the diagram, or how to handle user's authentication and authorization. Now that we have defined our contexts, domains and our microservices, let's tackle one of the problem we just mentioned: authentication and authorization.

How to manage authentication using an OIDC

Introduction

Let's break down this problem by first understanding important concepts that surrounds authentication, OIDC - and by extension, authorization and the Keycloak vocabulary.

What is authentication?

Most of today's systems defines personalized behavior based on who is using the application. Think of a bank application, you want to see your accounts and transactions, but you don't want to see someone else's ! To be able to do this, the application needs to know who you are. This is what authentication is. Authentication is the process of verifying the identity of a user.

This process can be done in a myriad of ways, but the most common one the end user is going to identify themselves with is by providing a username and password. Or by using a third party service, like Facebook or Google, to identify themselves.

Those third party service, like Facebook or Google, are called identity providers (IdP). There are multiples types of standards, the most common being OpenID Connect (OIDC) and SAML.

This section is about OIDC, so I'm not going to go into details about SAML. OIDC is gaining in popularity, thanks to its use of JSON which make it much more easier to work with with nowadays solutions, while the older SAML protocol is XML based. SAML is still widely used, but mostly in company environments.

What is authorization?

Let's take our bank example a step further. When logging in, you want to see your accounts and transactions, but you don't want anybody else to see them ! This is what authorization is. Authorization is the process of verifying what a user is allowed to do.

To do so, whenever a user is taking an action which requires a level of authorization, the application is going to check whether the user actually have the right to do so.

OpenID Connect

OpenID Connect (OIDC) is a identity layer build on top of OAuth2. It allows to authenticate users using an Authorization Server (AS) and to get information about the user, using a REST-like manner.

The main participants are :

- The end user, which is the entity that is going to authenticate. They are the equivalent of the resource owner in OAuth2, and one of the resource they own is their identity.
- The Relying Party (RP), or client, which is the application that is going to use the identity of the end user. They rely on the OpenID Provider (OP) to authenticate the end user.
- The OpenID Provider (OP), also referred as IdP (Identity Provider), which is the entity that is going to authenticate the end user. They are an authorization server in OAuth2. They provide

the identity of the end user to the relying party. The identity may contains information about the user, like their name, email, etc..

- The Resource Server, which owns the resource that the end user or client is going to access. They trusts the Authorization Server (the OIDC Provider), and will check the token provided by the OP to verify the client was given access to the resource. A Resource Server may be in the same system as the Authorization Server.

This data is exchanged using JSON Web Tokens (JWT). JWT are a way to encode data in a compact way, using JSON. They are usually signed using an asymmetric algorithm, and can be verified using a public key.

The main token used in the OIDC protocol is the ID Token. It contains information about the user, and is signed by the OP. It is the ID Card of the OIDC protocol.

Another type of token used in the OAuth2 protocol (which is used in OIDC since it's a superset), is the access token. It is used by the client application to access the resource owner's (the end user) resources. This token is usually short lived, and a new one can be obtain using a refresh token in most authentication flows. An access token is tied to a client, and defines the scopes and resources the client is allowed to access.

In most flows, the user is also given a refresh token. Since access token are short lived, refresh token provides a way to refresh them, as its name indicate.

More information can be found in the official specifications

Keycloak

Keycloak is an open-source identity and access management solution that provides a centralized platform for managing authentication and authorization for applications and services. It offers a range of features and capabilities, including support for a wide range of authentication mechanisms and protocols, user management and provisioning, and integration with third-party systems.

Keycloak is built on top of the WildFly application server and uses the OAuth 2.0 and OpenID Connect (OIDC) standards for authentication and authorization. This allows applications to verify the identity of users through a third-party identity provider (IdP), such as Keycloak, and securely access resources on behalf of users without requiring them to share their credentials.

It provides a web-based administration console that allows administrators to manage users, roles, and applications, as well as configure authentication and authorization settings. It also offers a range of customization options, such as custom themes and branding, to tailor the user experience for specific applications or organizations.

Keycloak is designed to be highly scalable and can handle millions of users and thousands of applications. It is easy to deploy and can be integrated with a wide range of applications and services, including web, mobile, and single-page applications. Overall, Keycloak provides a comprehensive solution for managing user authentication and authorization in modern applications and services.

Keycloak terminology

Keycloak has its own vocabulary of terms and concepts that are important to understand in order to effectively use and configure the platform. Some key terms and concepts in Keycloak include:

- **Realm**: A realm is a logical partition in Keycloak that represents a separate security domain. Each realm has its own set of users, roles, and applications, and can have its own authentication and authorization policies.
- **User**: A user is an individual who is registered in Keycloak and can authenticate to access applications and services. Users are associated with one or more realms and can be assigned different roles within each realm.
- **Role**: A role is a collection of permissions that can be assigned to users. Roles are defined at the realm level and can be used to control access to different resources and applications within a realm.
- **Client**: A client is an application or service that is registered in Keycloak and can authenticate users. Clients can be public or confidential, depending on whether they can securely store and manage their own secrets.
- **Protocol**: Keycloak supports a range of authentication protocols, such as OAuth 2.0, SAML, and OpenID Connect (OIDC), which define the messages and flows used to authenticate users and access resources.
- **Identity provider (IdP)**: An identity provider is a service that provides authentication and identity information for users. In Keycloak, the Keycloak server itself acts as an IdP, allowing applications to verify the identity of users through the Keycloak IdP.
- **Access token**: An access token is a JSON Web Token (JWT) that is issued by the Keycloak server and used to authenticate and authorize access to protected resources. Access tokens contain information about the user and the permissions they have been granted.
- **Scope**: A scope is a collection of permissions that can be associated with an access token. Scopes define the specific resources and actions that a user is allowed to access and perform.
- **Federation**: Federation is the process of connecting Keycloak to other identity providers, such as social media or enterprise directories, to allow users to authenticate using their existing credentials. This allows users to log in to applications using a single set of credentials, without the need for each application to manage its own authentication processes.

Understanding these terms and concepts is essential for using and configuring Keycloak effectively.

Keycloak and Polycode

As of the right now, Polycode defines and runs it's own implementation of a basic OAuth2 server, implementing only the direct grant flow. It is running on the same application that also is the resource server.

It's important to realize that, as our API becomes a Resource Server, accessing it requires authentication, even inside the Polycode application.

The goal is to replace this implementation with one using Keycloak, and its OIDC capabilities. We will slightly touch on some migration steps, but the main goal is to have an high level overview of the architecture.

Architecture Diagram

We will discuss how we can integrate OIDC in Polycode. As of right now, Polycode still follows a monolithic approach. However, as discussed in the previous chapter, there is a clear path to split the application in multiple microservices. Although this is a significant change in the architecture, I think we can simplify the relationship between the different components of the OAuth2 flow. Indeed, in practice, an API Gateway or a shared library between our microservices will handle the authentication. One will run only in one application, and the other will run on multiples services, but since the code that will handle the authentication will be the same, we can for simplicity sake, consider that the authentication will be handled by only one application. I will refer to this application as the "Polycode API" throughout this chapter, whatever the implementation.

I think this approach is justifiable, since I want to focus on the core concepts behind OIDC (and AuthN / AuthZ in general). What I want to show throughout this chapter is the flow of data between the different actors, and the different responsibilities of each actor. The Polycode API, will be a Resource Server and an Client application, whatever how the application is break down.

With that out of the way, let's have a look at the architecture schema:

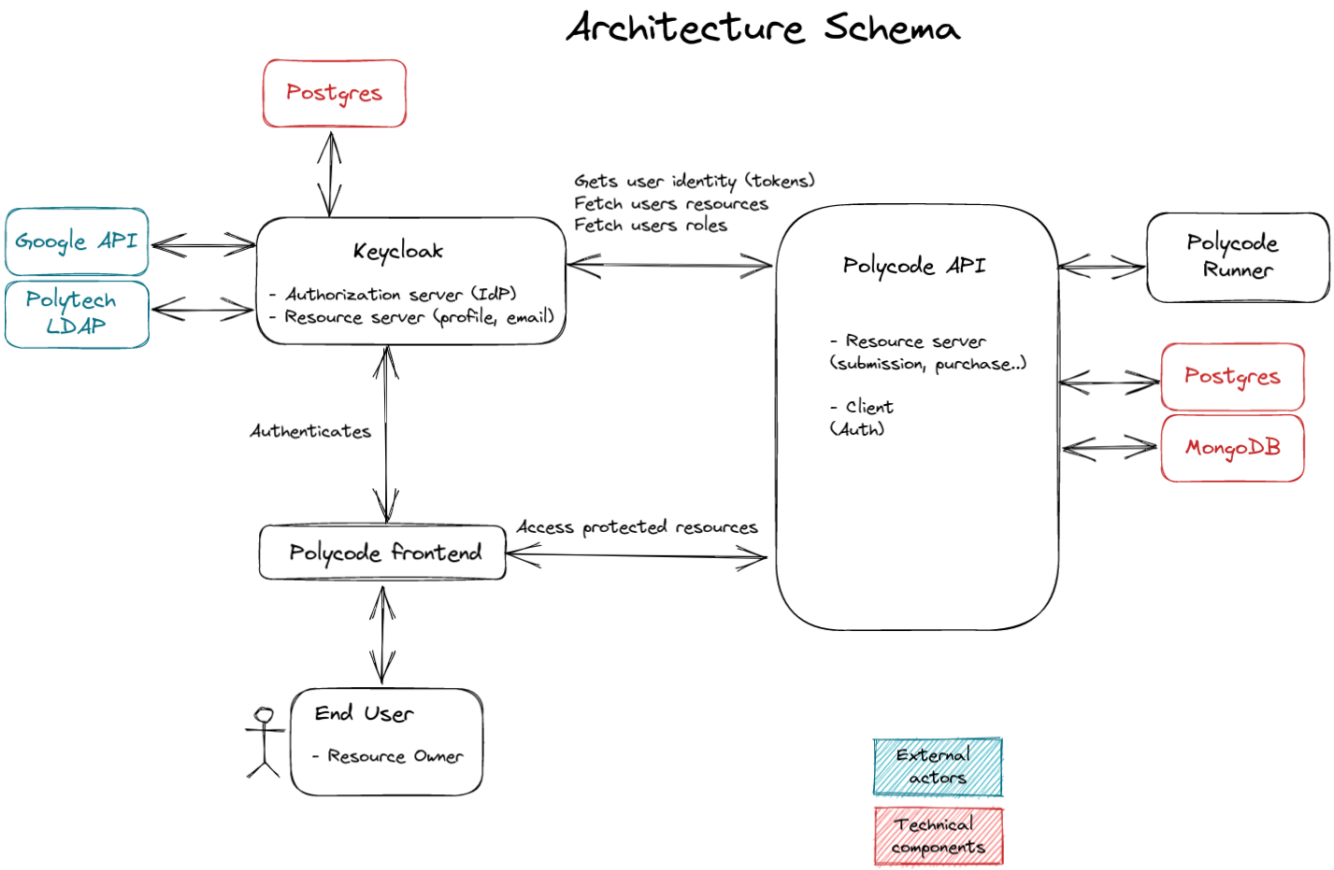


Figure 13. Architecture Schema

Deployment Diagram

We have seen all the elements of our architecture, but we still need to know how they are deployed. This is an important step in the design of our architecture, since it will impact performance, security, scalability and resilience.

First of all, we need to understand that our IdP (Keycloak) is not directly linked to Polycode. Although the API is the only client of the IdP, this might change in the future. We can imagine using this Keycloak for all Polytech related projects for example. Keycloak becomes a dependency of our API, but it is not part of our API.

We that in mind, I think it's a good idea to deploy Keycloak in a separate system. This will allow us to scale it independently from the API, and to have a clear separation of concerns, which will allow us to take down the API without impacting other future users of our IdP. We can see in the following diagram how this is realized. The Keycloak is deployed in its own server, with its own database. It is an independent system that doesn't rely on the API deployment infrastructure.

Just like the current implementation of Polycode, all the services of Polycode are deployed in a single Kubernetes cluster, composed of multiple nodes (only one node is shown in the diagram for simplicity). I've made the decision to use SaaS services for the databases, to off-load the time and risks associated with managing its own database. We could use self-hosted database, but this would require more maintenance and would be less resilient. This, however, has performance and cost implications.

Performance will be greatly impacted, since the database will be hosted in a different location the the API, which is currently hosted at Polytech. By doing this, we are increasing delays by orders of magnitude. This is a trade-off I'm willing to make, and migrating back and forth to a self-hosted database is not a huge task (but still needs careful thinking), just like migrating the current infrastructure to the cloud.

As shown in the diagram, the Keycloak will be accessible from the internet, just like the API and the Frontend. It is necessary but it is important to recognize that this is a security risk. We need to make sure that the Keycloak is properly secured, and that we don't expose any sensitive information.

Just like like in the previous diagram, the API is a simplification of what actually runs. This is still justifiable, since all micro-services and gateways will be running in the same cluster, and in the same namespace.

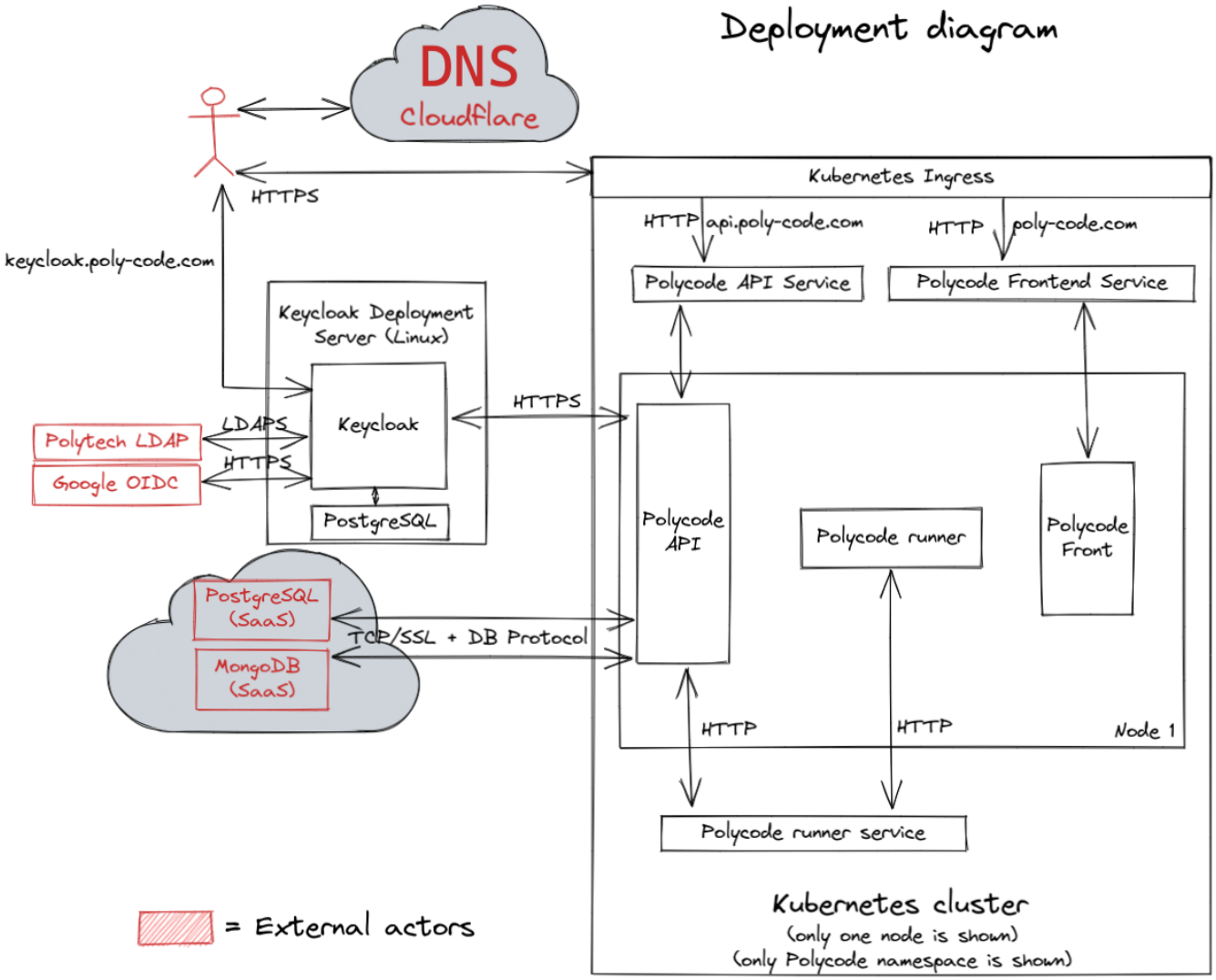


Figure 14. Deployment Schema

Sequence Diagrams

General authentication flow: the code flow

This diagram sequence presents the authorization code flow, a way of granting access to a web application using an authorization code. This flow is often used in the context of OIDC. In the authorization code flow, the user is first redirected to the IdP to authenticate. The IdP then redirects the user to the provided redirect URI, with an authorization code. This code can then be sent to the API that will exchange the code for an access and ID token. This access token can be used to access protected resources, by the API, on behalf of the user.

The authorization code flow provides several benefits over other authentication flows. It allows the user to authenticate without sharing their credentials with the API, and it allows the API to verify the user's identity without having to store the user's credentials. Additionally, the use of an authorization code adds an extra layer of security, as the code can only be exchanged for an access token by the intended recipient.

The authorization code flow is commonly used in modern web and mobile applications to securely access user data from a third-party service. In the case of Keycloak, the IdP is responsible for authenticating the user and managing their identity information, while the web application and API can focus on providing a great user experience and accessing protected resources on behalf of the user. The future diagrams in this sequence will describe different ways to connect to the application using this flow, such as connecting with a Google, Polytech or vanilla account.

Once the API has exchanged the code for the tokens, it should create a mapping between its Session ID and the tokens. This Session ID will be used by the web Application to authenticate its requests to the API.

Implementing this can be done in multiple ways. The API can use a shared, fast database such a Redis or Memcached, since the data stored should be indexed by a simple key, with no need for relationship. TTL would also be useful, since each entry could be set to expire at the time the tokens expire. By doing so, we keep our API stateless, and scalability is not impacted.

We can also think about creating a gateway that would be responsible for this mapping. In a microservice world, it is out of the scope of each microservice to map cookies with tokens, and should be off-loaded to a gateway. This gateway could also be used to implement rate limiting, observability, and other features that are not directly related to the business logic of the API.

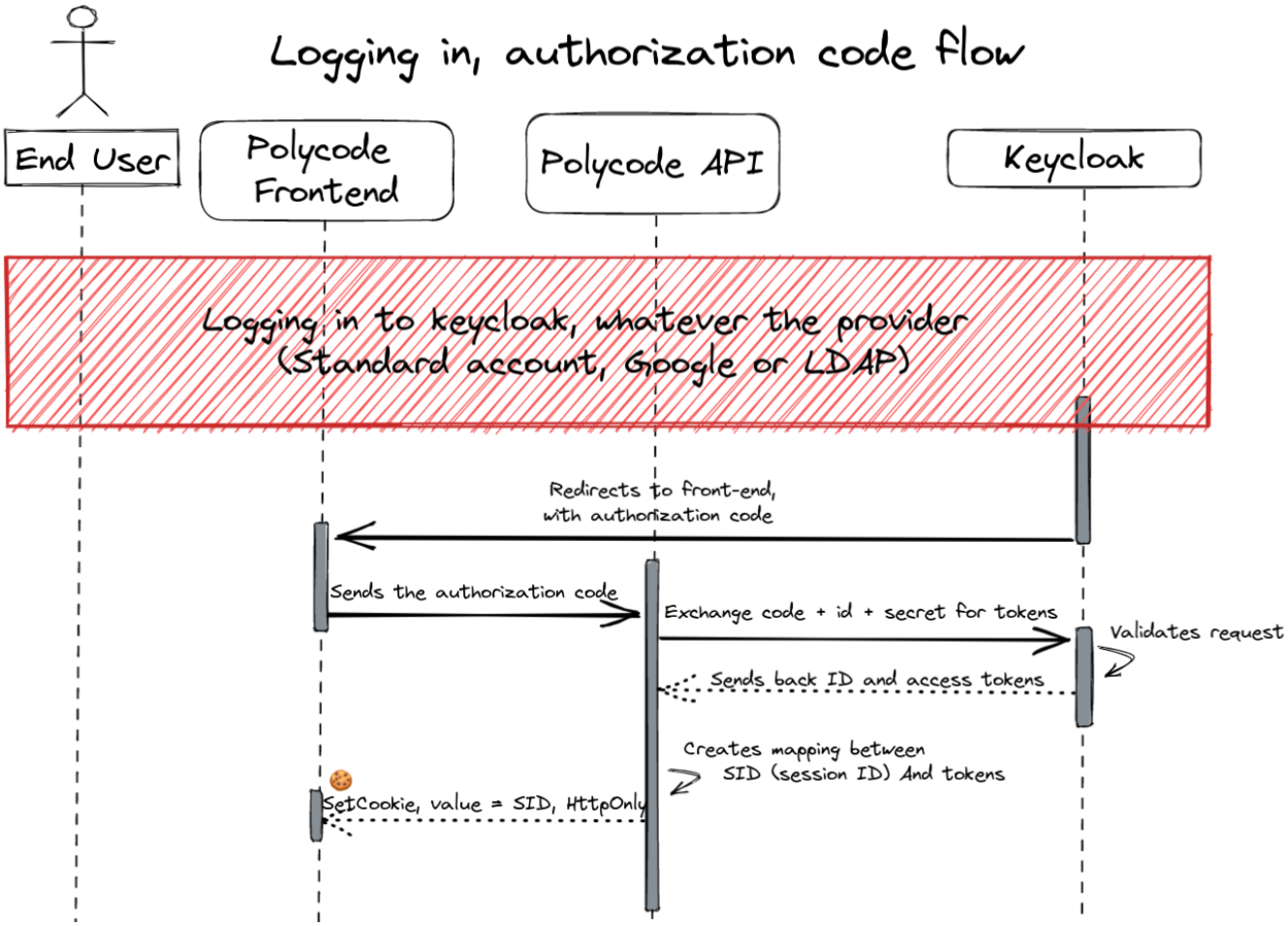


Figure 15. Authorization code sequence diagram

User creates an account, vanilla

The following diagram shows the process of creating a vanilla account on Polycode. We can see that the user is redirected to Keycloak to create an account and provide their email address and password. The IDP will then check for email duplication, validity, and password strength to ensure that the user's account is secure.

Once the user has created their account, the IDP will send a verification email to the user, which the user must click on to confirm their email address. Once the user's email address has been verified, they can use their vanilla account to log in to the web application and access protected resources on the web application on behalf of the user.

The next diagram will describe how the user can use their vanilla account to log in to the web application and access protected resources.

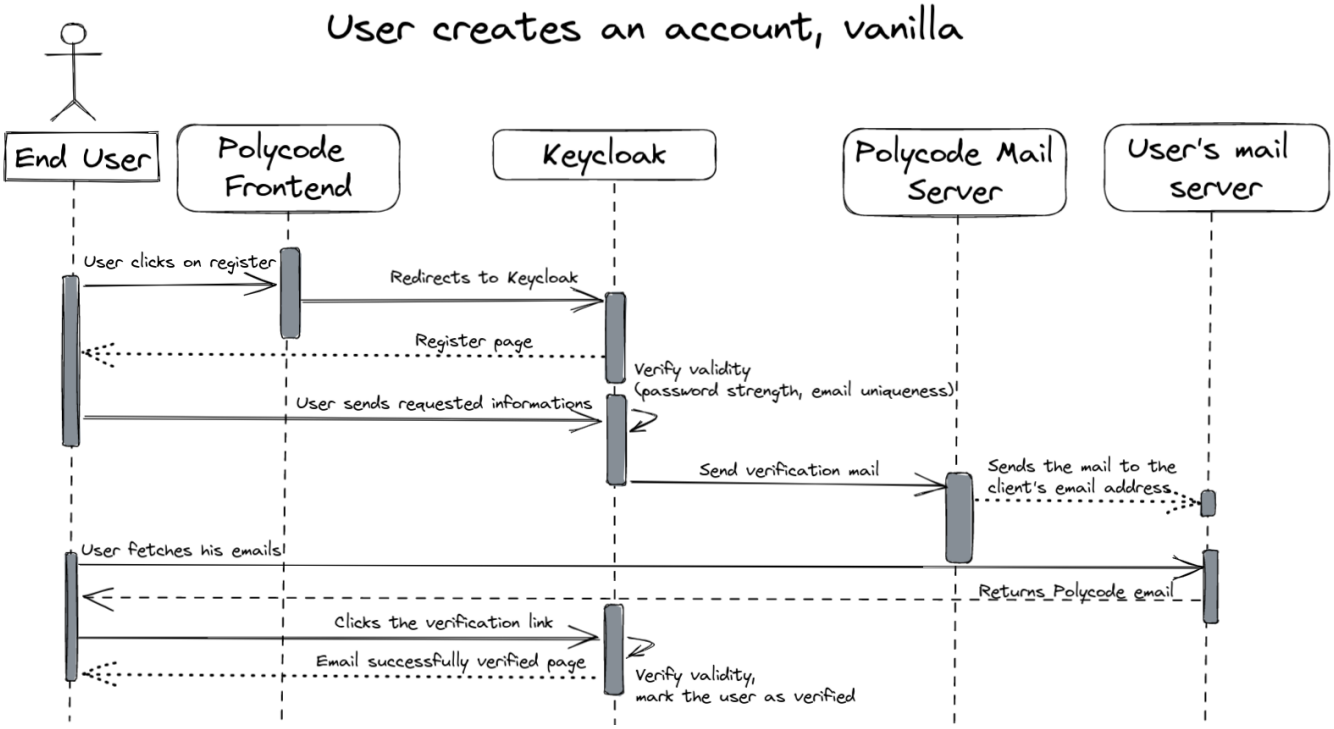


Figure 16. Vanilla account creation sequence diagram

User logs in, vanilla

This diagram sequence demonstrates how the user can log into Polycode using a vanilla account. The user is redirected to the IDP, where they can enter their email address and password. The IDP will then verify the user's credentials and redirect the user to the web application, with an authorization code. The authorization code flow is described in a previous diagram.

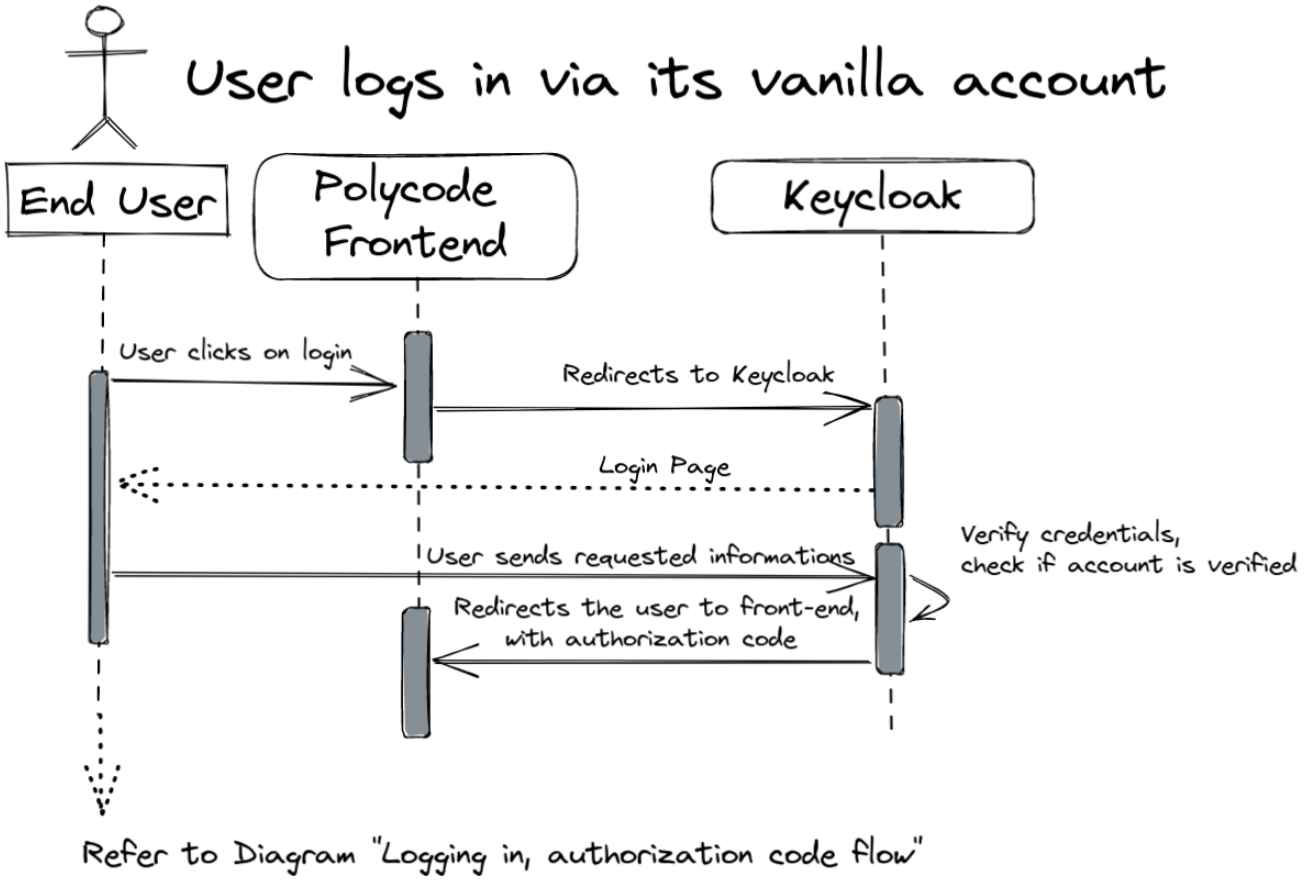


Figure 17. Vanilla account login diagram

User creates an account or logs in, via Polytech LDAP

In the following sequence diagram, the user logs in into Polycode using its Polytech LDAP account. The user is redirected to the IDP, where they can enter their Polytech LDAP credentials. If it's the first time the user connects to Polycode, a new entry will be made in the Keycloak database. Keycloak needs to do this in order to store information that are needed in the OIDC context. Since we are likely to be limited to a read-only access to the LDAP, we also need to store everything the user might edit later on. Storing in the local database also allows for caching, which will speed up all requests that needs to check for a LDAP linked user.

It is worth noting that Keycloak will always validate the user password using the LDAP, and delegates the storage of the password to it. This is done to ensure that the user's password is always up to date, and synchronized with the LDAP.

The IdP will then verify the user's credentials and redirect the user to the web application, with an authorization code. The authorization code flow is described in a previous diagram.

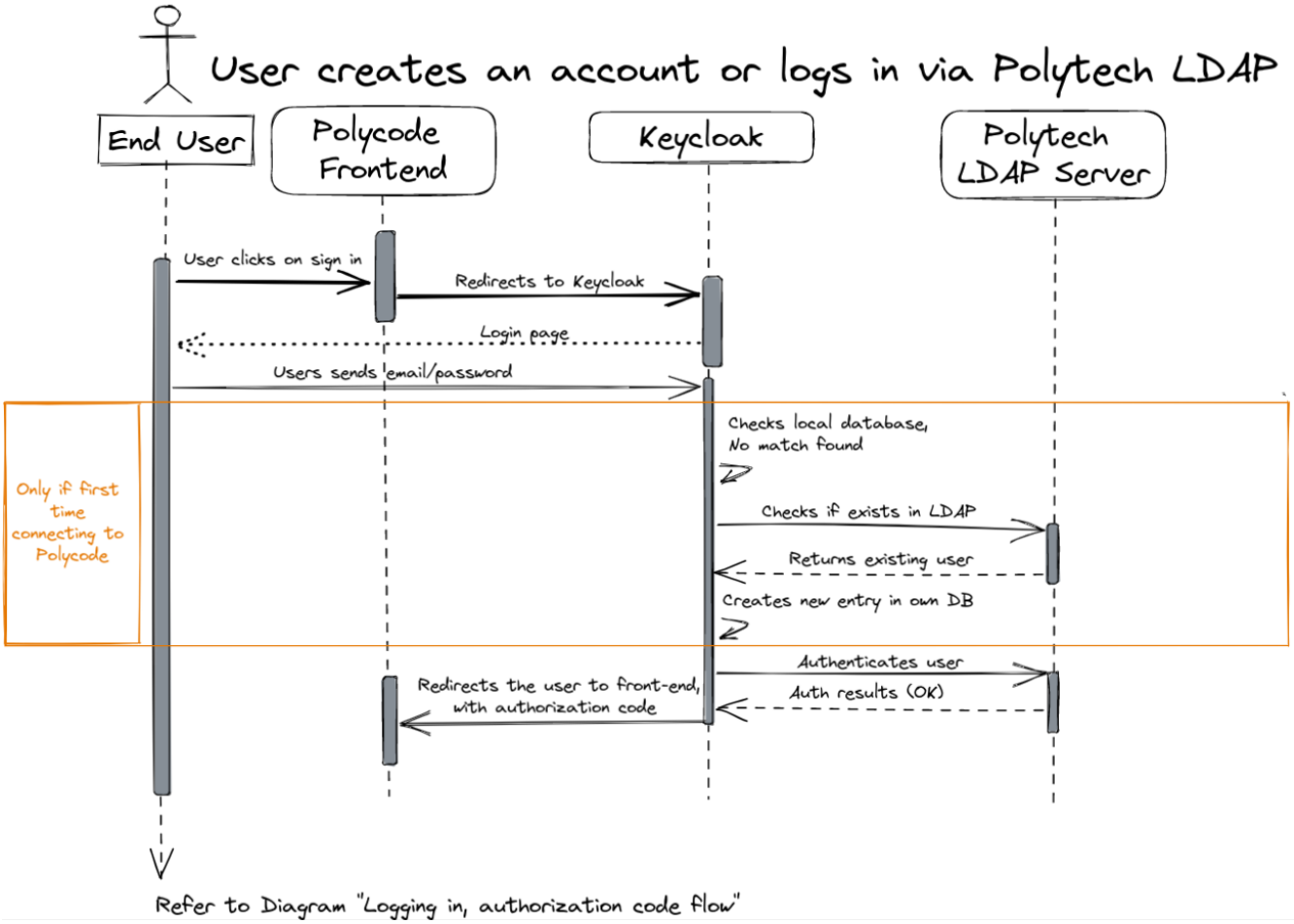


Figure 18. LDAP sequence diagram

User creates an account or logs in, via Google

In this diagram, we will be demonstrating the process of connecting to an application using a Google account. This process is becoming increasingly popular among users as it allows them to easily and securely access multiple applications using a single set of credentials.

Since Google is an external identity provider, the user will be redirected to the Google login page, where they can enter their Google credentials. Google also uses OIDC, so the user will be redirected to an OIDC endpoint, with a authorization code. The authorization code flow will then be used to exchange the authorization code for an access token and ID token. Keycloak automates this process.

If it's the first time the user logs in using Google, a new entry in the Keycloak database will be created. This entry will contain the user's Google ID, which will be used to identify the user in the future.

The user will then be redirected to the web application, with an authorization code. The authorization code flow is described in a previous diagram.

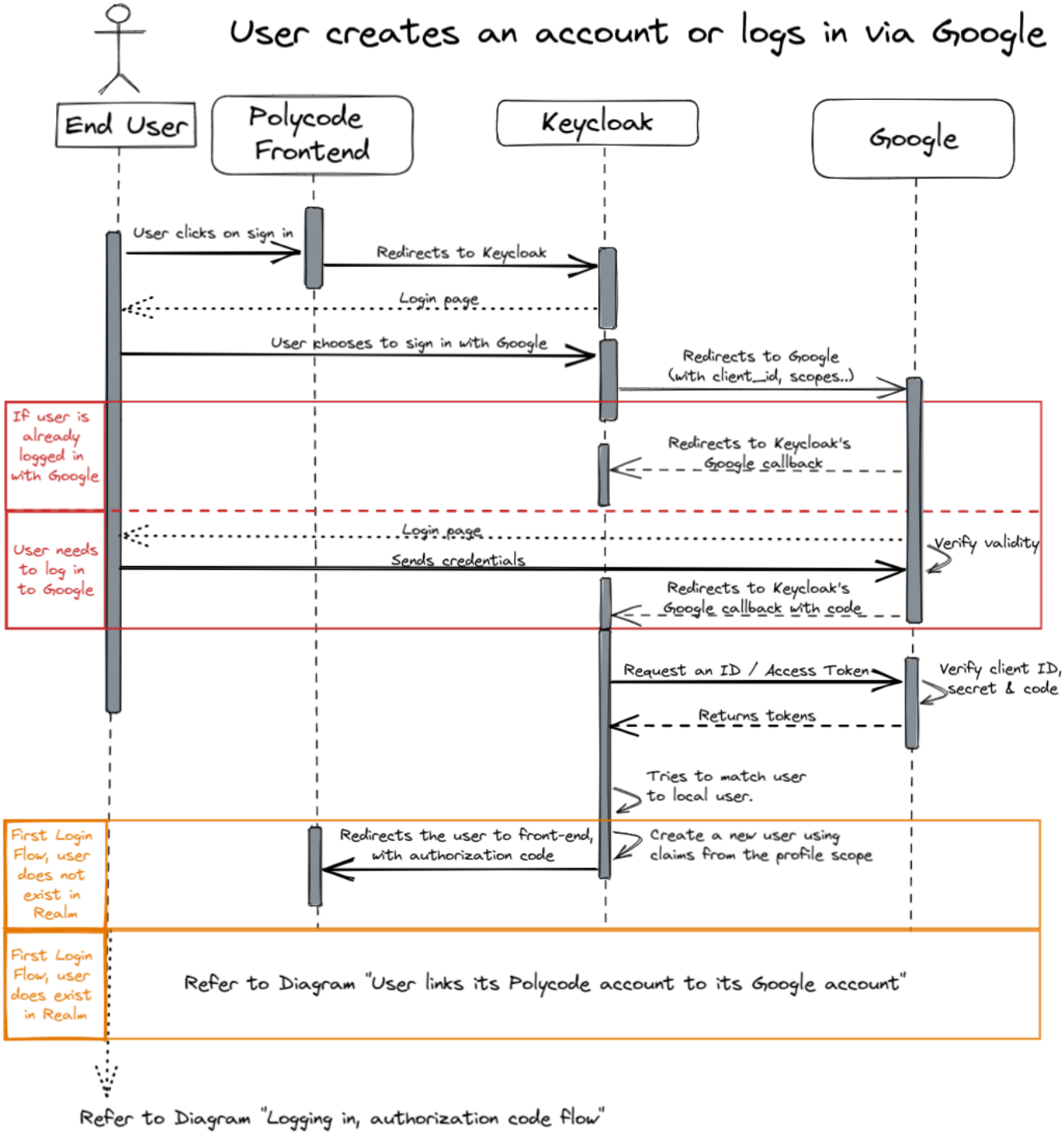


Figure 19. Creating or logging in via Google, sequence diagram

User links its Polycode account to its Google account

A user that created an vanilla account on Polycode might also want to log in via its Google account. This is possible if both uses the same mail address. This is what this sequence diagram describes.

In this case, when logging in, the user will be able to edit its profile before linking its Google account. The user then need to verify that he is the owner of the Google account by clicking on a link sent to his email address. Once the user has verified his email address, the user will be redirected to the web application, with an authorization code.

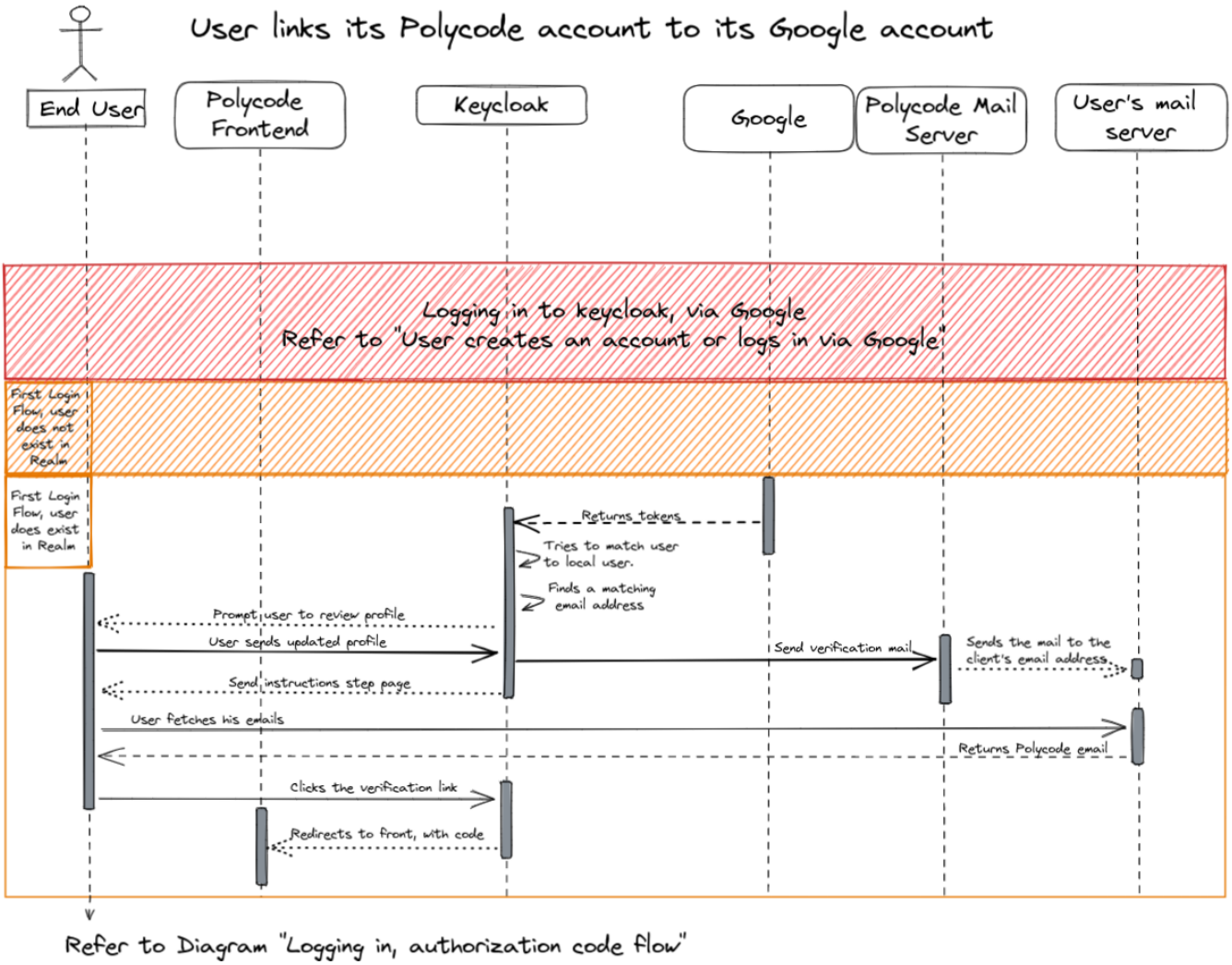


Figure 20. Linking your Google and vanilla accounts sequence diagram

Proof of concept

Now that we have a deeper understanding of how I suggest handling authentication in the Polycode context by using keycloak as our OIDC, I have made a sample proof of concept that implements some of the describes flow shown above.

You can find the repository here, and an already deployed environment here. To sum up what this PoC does in a few lines:

- Our IdP is keycloak
- Our client and resource server is the backend

- The user delegates access to its information to the backend using OAuth2
- The user stays logged in using a cookie session
- Authorization is handled at the keycloak level

They are, however, some lacking features and shortcuts:

- Authorization is not as fined-grained as we have in the existing polycode
- Mapping between cookie and JWTs is not done on a database, meaning we can't have replicas for the API
- Is a bit buggy

This is a PoC and I think it demonstrates nicely that my implementation can be made to work pretty easily.

Conclusion

We discovered authentication, authorization and OIDC. We looked what keycloak was and how we can make use of it for Polycode, by using as our IdP. I would like, as an opener for further thinking to this section, to discuss a bit more about how we could use this new keycloak instance. Indeed, this keycloak would only be used by polycode for now, but nothing is guarding us from using it for something else. Effectively, this keycloak could become our main IdP for all our future projects, federating users from Polytech's LDAP, Github accounts, Google accounts and vanilla accounts. We must not see keycloak as being part of the Polycode system, but like Polycode system being reliant on keycloak to provide authentication and authorization capabilities. This opens up a pretty wide range of possibilities for future projects, or existing one who would want to integrate our user-base.

Inter microservices communication

Introduction

With microservices comes a set of new challenges. One of them is not only to think about the data getting transferred between your services and layer of applications, like with a monolith application, but also to think how this data is transferred. Indeed, it was relatively simple before. Since everything was running in a single application, all you had to do was a function call ! With microservices however, this becomes much more complicated. You will have to make networks calls to fetch your data from other microservices. But a network call is abstract and doesn't provide much more information other that we are probably going to use the internet protocol to talk to each other. This doesn't define the layer on top of them. Are we going to use raw TCP / UDP and serialize and deserialize the data ourselves ? What would that implies ? Do we rely on existing technology, such as HTTP ? Maybe, but how do you define your payloads ? Where exactly resides your data, or business logic ? Maybe doing synchronous calls doesn't really makes sense. Do we really need to wait for an email to be sent ? Doesn't this create a strong coupling, something we're trying to avoid ? All of those questions, and so much more, is why you knowledgeable architect and developers in your team. The decisions and implementations you are going to take will be a set of trade-off, and you need to identify, for each of your microservices, what are the trade-offs you're willing to make. But what are the trade-offs you can make exactly, and what are the available solutions on the market that can help us resolve those problems ?

What are the goals

Let's identify and talk about the main goals and characteristics we look for in a inter microservices communication. These will be concepts, and we are not yet diving into actual protocols and solutions, but we are rather taking a high level overview of every piece of the puzzle we will need to build up a sensible solution for our use cases.

Latency and performance

As mentioned previously, since we moved our services in different applications, we also added overhead for all of our communication between our services. Wether the applications runs in the same machine or not, the transport delay and overhead is now significant, and can't be ignored, especially when requesting something that does not need a lot of processing. When choosing a protocol, you need to think about the overhead it will introduce in term of latency, especially when the two services will be chatty.

You need to identify if your services need synchronous response, meaning that they can't continue to process a request until they have a response from another service. If this is the case, you probably want a transport protocol that is fast, that can keep connection open (so you don't have to handshake every time), and if you're going to send batches of request at the same time, a transport protocol that can support multiplexing over the same connection. You don't want to end up open dozens of connections, as this is a significant overhead, on both of your services. Opening, closing and multiplexing connections are far from the only aspects of performance. On top of your connection, you're going to transfer data. The way you encode this data is also very important for your overall performance. Indeed, more verbose encoding (self-describing one, like JSON) are heavy to transport, and ends up in more traffic on your connections. This might not be a big concern, depending on your deployment architecture, but you still need to take this in consideration.

Another downside of those verbose encoding, is the cost of serialization and deserialization. I've seen applications where 60 to 70 percents of the CPU load was encoding and decoding JSONs. This microservice was poorly designed, its main job was to handle huge batches of requests, all with a well-known request and responses structure. It was running on HTTP/1.1, which does not handle multiplexing (unlike its bigger brother, HTTP/2), with a REST API (using JSON as data structure). The performance were catastrophic, and switching to a protocol that supported multiplexing and that has a close to computer serialization, we were able to speed up significantly all the requests to this microservice, that became a bottleneck, while reducing replicas, clearly highlighting the cost related to choosing the wrong protocol for the job.

However, if you need to process data that might not be well-defined ahead of time or if you don't need to have this kind of performance, because your microservices will not be chatty at all, self-describing encoding can make it way easier to debug, since those tends to be human-readable, and does not cause a big performance hit when the volume is low.

Fault tolerance and resiliency

Fault tolerance and resiliency are important considerations in inter-microservice communication, as they help make sure that the overall system can continue to function properly even in the face of failures or errors. These concepts refer to the ability of a system to continue functioning in the face of failures or errors, and to recover from these failures in a timely and effective manner.

Fault tolerance is the ability of a system to continue functioning normally, or at least in a degraded mode, when one or more of its components fails or experiences an error. This is accomplished through the use of redundant components and failover mechanisms that allow the system to continue operating even if one or more components fail. In the context of inter-microservice communication, fault tolerance may involve the use of load balancers, message queues, and other mechanisms to make sure that messages are delivered to their intended recipients even if one or more microservices are offline or experiencing errors.

On the other hand, resiliency refers to the ability of a system to recover from failures or errors in a timely and effective manner. This could imply the use of techniques such as rolling updates, which allow a system to be updated or repaired without disrupting service, or the use of self-healing mechanisms that automatically detect and correct problems as they occur. With microservices, resiliency usually involves the use of monitoring and alerting systems to detect problems and trigger corrective actions, as well as the use of recovery procedures to restore the system to a healthy state after a failure or error.

Together, fault tolerance and resiliency are essential for ensuring the reliability and uptime of a distributed system, particularly in cases where the system is critical for business operations or customer experience. By designing their inter-microservice communication systems to be fault tolerant and resilient, organizations can reduce the risk of service disruptions and improve their overall system reliability.

One way to achieve fault tolerance in inter-microservice communication is through the use of redundant communication channels. For example, if a microservice communicates with another microservice over a network connection, you could have multiple network connections available in case one fails. This can help prevent a single point of failure in the system. This is mostly applicable on systems that communicates over the internet, and you need to design your architecture with this knowledge in mind. Another way to achieve fault tolerance is through the use of circuit breakers. A circuit breaker is a pattern that allows a microservice to temporarily stop attempting to communicate with another microservice if it detects that the other microservice is experiencing problems. This can help prevent the microservice from becoming overloaded with failed requests, and can allow it to continue functioning properly even if the other microservice is experiencing issues. It has, however, some downsides :

- Increased complexity: Implementing circuit breakers can add complexity to a system, as it requires the development and maintenance of additional code, if implemented in the application.
- False positives: If the circuit breaker is too sensitive, it may trigger unnecessarily and cause a microservice to stop communicating with another microservice even when there is no actual problem. This can lead to disruptions in service and reduce overall system performance.
- Stale data: If a circuit breaker is triggered and a microservice stops communicating with

another microservice, it may continue to use stale data until the circuit breaker is reset. This can lead to outdated or incorrect information being used, which can have negative consequences.

- Limited visibility: Circuit breakers can make it more difficult to diagnose and troubleshoot problems, as they can obscure the root cause of failures.
- Dependency on a single point of failure: While circuit breakers can help prevent a single point of failure from cascading through the system, they themselves can become a single point of failure if they are not implemented and maintained properly.

Overall, while circuit breakers can be a useful tool for improving fault tolerance and resiliency in inter-microservice communication, they should be carefully evaluated and used with caution to make sure that they do not cause more problems than they solve.

Resiliency in inter-microservice communication can be achieved through the use of retries and failover mechanisms. For example, if a microservice is unable to communicate with another microservice, it might retry the request a certain number of times before giving up. Retries refer to the process of repeating a request or operation in the event of a failure or error. They are commonly used in distributed systems to improve reliability and make sure that requests are successful even if there are temporary failures or errors. There are several factors to consider when implementing them in a distributed system. These include the number of retries to be attempted, the interval between retries, and the conditions under which retries should be attempted. It is important to balance the benefits of retries with the potential for increased load on the system and the risk of creating an infinite loop if the failure is not resolved. Overall, retries are a useful tool for improving the reliability and uptime of a distributed system. By implementing retries in a smart and well-considered manner, we can improve the success rate of our requests and reduce the risk of service disruptions.

In addition to these proactive approaches, it is also important to have robust recovery procedures in place in case of more serious failures or errors. This may involve the use of backup systems and data, as well as well-defined recovery processes that can be followed in the event of a system failure.

Decoupling

One of the main benefit of a good microservice architecture, is how you can make your microservices independent from one another, at the development stage but also when running. By decoupling the services, they become more independent and can be developed, tested, and deployed independently of each other. This makes it easier to make changes to one service without affecting the others, as well as allowing for more efficient and flexible development processes. You should aim at making your microservices the least reliant on others as possible. If your microservices can’t function without another one, it should be a sign that those two microservices should be merged into one.

One side effect of decoupling your microservices, is that they can be scale independently. This means, however, that your inter-microservice communication implementation should not rely on a specific instance, or make any assumptions about the availability and state of another service, since others microservices will be scaled up and down based on the current load of your application. You need to be able to adapt to the current state of your services, meaning you have state somewhere in

your system. Since handling state within an application should be avoided, this is usually done at the operation layer. This provides multiple benefits:

- Your applications can be deployed in a variety of environment, and is not tied to your current system.
- Operation-layer solutions usually focuses on this specific problem. This means there are solutions that are robust, reliant and with a large set of features already available to be used.
- Since communication between your microservices should be standardized, adding an implementation in each of your microservice adds a lot of overhead.

This is usually done by using service discovery. In a Kubernetes environment, service discovery refers to the process of locating and communicating with services running in the cluster. There are two main approaches to service discovery in Kubernetes: client-side discovery and server-side discovery.

Client-side discovery involves the use of a client library that knows how to discover and communicate with the various services in the cluster. The client library abstracts the details of service discovery and communication, allowing the application to focus on its business logic. This, however, as discussed above, makes your application architecture-aware, in the sense that it has to handles operation specific implementation. If your microservices are written in different languages, or use a different framework or version from one another, you will have multiple library to maintain, which can become cumbersome.

Server-side discovery involves the use of a central service or component that is responsible for managing the registration and discovery of services in the cluster. The services communicate with this central component to register themselves and locate other services they need to communicate with. This, however, introduce a single point of failure. If your central component were to fail, the entire system may become unavailable, and you need to have a strong failover system to take over when this component eventually fails. This single point of failure is also a single point of communication for all your microservices, meaning it becomes much more easier for your developers to interact with your system.

Security

In a microservice architecture, security in inter-service communication is important because services are decoupled and communicate with each other over networks. This means that there is a higher risk of unauthorized access to data or services and the potential for attacks such as man-in-the-middle or replay attacks. There are several way to secure inter-service communication in a microservice architecture:

- Use secure communication protocols. Services can communicate with each other using secure protocols such as HTTPS (or more broadly TLS). It encrypts the data transmitted between services, making it more difficult for attackers to intercept and read the data.
- Implement authentication and authorization. Services can authenticate each other using techniques such as mutual TLS (mTLS) or JSON Web Tokens (JWT). This ensures that only authorized services can communicate with each other.
- Use a operation layer that sits between services and handles communication between them.

This can be typically handled by a service mesh (more on them later), and can provide features such as mTLS, rate limiting, and request tracing.

- Network segmentation: Divide your network into smaller, isolated segments. This helps preventing unauthorized access to services and data by limiting the ability of attackers to move laterally within your network.
- API Gateways: It's a reverse proxy that sits between clients and services, and acts as a single point of entry for all incoming requests. It usually also implements authentication, authorization, rate limiting, is the entry point for your traces, and some also serve as a service discovery registerer.

System agnosticism

The last goal I would like to talk about is system agnosticism. A good communication protocol should not rely on a specific underlying technology, and should be usable whatever the implementation you make of them. Java's Object Streams is a good example of a bad idea, since they're not agnostic (and have a ton more problems totally out of this scope). What the underlying constraint is, is to choose a protocol that can be serialized and deserialized easily, whatever the stack you use. Most solutions out there nowadays are system agnostic (even if some languages are easier to work with for some), such as JSON or protobufs. Once again, the idea with microservices is that you have multiples, small-sized, independent systems. If you introduce technologies that makes assumptions about what's running on the other end of your connection, it means that for your system to work, the other end actually has to follow those assumptions. You're putting yourself into handcuffs, and are tying your microservices together.

The options, protocol layer

Now that we defined the main goals of an inter-service communication, we'll look into what are the existing solutions, and which makes sense in our Polycode architecture. We'll start by talking about solutions at the protocol layer, meaning we will look at solutions that defines how data is structured inside the messages that are sent, what type of messages can be sent, etc..

GraphQL

I'll start by talking about a solution that doesn't make sense to me. While researching, I've came across solutions that were using GraphQL as a inter-microservice communication protocol. I would argue that GraphQL is made to sit between the frontend and the entry point of your API, most likely with an API Gateway as your GraphQL Server, and your microservices as GraphQL Resolvers. Its front-facing features are great, and you can filter out the fields that should remain internal to your system. However, when working within your microservices, you should not have to strip fields and data. You should not rely on a GraphQL Server to resolve your requests for your, as this introduces a massive single point of failure. All your microservices become dependant on this service, and if it were to go down, all your infrastructure would go done as well. Your microservices should be able to talk to one another through a well-defined API, with contracts that does not change over time. With GraphQL, you have no decoupling, scalability will be limited by the scale of your GraphQL server (which needs to do all the heavy-lifting, not just passing-through requests), you have no option for resiliency and your failover options are limited to implementing another communication protocol, which you should probably do in the first place.

GraphQL is not the right tool for inter-service communication, and I'll strongly suggest avoiding using it as your communication protocol. I would not use GraphQL is any of the inter-microservice communication protocol in Polycode.

Asynchronous communications

While we are not allowed to use them in our solutions, asynchronous communications have become a standard for some types of inter-microservice communication. Message queues solutions such as AMQP or Kafka have demonstrated how well they can handle heavy-load, and how they are useful at distributing events through your domain. Their main use case is to broadcast events that need to be handled eventually, but doesn't need an immediate response for the continuation of the current process. Sending a confirmation email, is a great example. You need it to be done eventually, but it is usually not critical to make sure the email was sent before continuing. Another big advantage of message queues is that they offer a total decoupling of microservices. A service does not need to know who will handle the event. This comes with the side effect of a great resiliency and fault tolerance, since the events are stored, even if the micro service consuming them is down, so when it will eventually get started back up, it can consume the requests it missed and have the operations brought back to normal, during a totally transparent process to all the others microservices.

However, messages queues should not be used for critical and synchronous operations, such as a transactions, as it is often tedious to have feedback when your request has been processed, and you can not afford putting a whole transaction in standby waiting for something to eventually be

processed. You want a fast, synchronous response, that fails if the service that is handling one of your requests fails to respond correctly. If I was allowed to use them in my architecture, I would use them for most of my domain events, such as sending emails for example.

REST API

REST, or Representational State Transfer, is a popular architectural style for building web-based APIs (Application Programming Interfaces). One of the main advantages of using REST for inter-microservice communication is that it is a widely adopted standard, which means that there are many tools and libraries available for building and consuming REST APIs. This can make it easier to integrate with third-party services and to build scalable, reliable systems.

Another advantage of REST is that it is based on the HTTP protocol, which is a well-established and widely supported protocol for communication on the web. This means that REST APIs can be easily accessed from any platform or language that supports HTTP, which is basically everything. Additionally, HTTP has a number of built-in features, such as caching, security, and error handling, which can be leveraged when building REST APIs.

However, there are also some disadvantages to using REST for inter-microservice communication. One potential issue is that REST relies on a stateless request-response model, which means that each request must contain all of the necessary information for the server to understand and fulfill the request. This can make it difficult to maintain context or state between multiple requests, which can be an issue if your microservices need to communicate complex data or maintain a stateful connection. This, however, might be seen as a blessing in disguise. Having a protocol that doesn't allow for stateful connection, forces the hand of developers into creating stateless systems, that are easily scaled up, but also easily scaled down. Both of those problems are difficult to tackle in stateful environments, and this is why you often see a push towards making stateless application.

Another disadvantage of REST is that it can be difficult to ensure that the API is being used correctly, as there are no strict rules governing how the API should be implemented. This can lead to issues with compatibility and maintainability, as different teams or developers may implement the API in different ways. Additionally, REST APIs can be difficult to version and maintain over time, as changes to the API may break existing client implementations.

Pull vs Push model

Another thing to consider, is the flow of your data within your system. This is the push vs pull model:

In the push model, the server pushes data to the client as it becomes available. This is typically achieved using a technique called long polling, in which the client sends a request to the server and the server holds the request open until it has new data to send to the client. The client can then receive the data and send another request to the server to get more data as needed.

The pull model, on the other hand, involves the client pulling data from the server as needed. In this model, the client sends a request to the server to retrieve a specific piece of data, and the server responds with the requested data. The client is responsible for initiating each request and can decide when and how often to request data from the server.

Both the push and pull models have their own advantages and disadvantages. The push model is useful for scenarios where the server needs to send data to the client in real-time, as it allows the server to proactively push data to the client as soon as it becomes available. However, it can also be resource-intensive for the server, as it requires maintaining open connections with multiple clients.

The pull model, on the other hand, is more efficient for the server, as it only needs to respond to requests from the client as they are made. However, it requires the client to actively request data from the server, which can make it less suitable for scenarios where real-time data updates are required.

Ultimately, the choice between the push and pull model will depend on the specific needs of your application and the data exchange patterns between the client and server. However, implementing a push model with a REST API is cumbersome, and long polling is a hacky way of using HTTP requests to reverse the flow of data. If your use case is within this use case, or if you need a duplex channel, using REST might not be the right choice.

Performance

There are several factors to take into consideration when talking about REST and performance. First, let's talk about multiplexing:

Multiplexing in REST is typically achieved using a technique called HTTP/2 multiplexing, which is supported by the HTTP/2 protocol. HTTP/2 multiplexing allows multiple requests and responses to be sent over a single connection in parallel, rather than having to wait for each request to complete before sending the next one. This can help to improve the performance of a REST API by reducing the overhead associated with establishing and tearing down separate connections for each request.

To use multiplexing in a REST API, the client and server must both support HTTP/2 and the client must initiate the connection using the HTTP/2 protocol. The client can then send multiple requests over the same connection, and the server can respond to each request as it is received.

Multiplexing in HTTP/2 comes with several features to improve performance, such as solving Head Of Line Blocking at the HTTP Level, but the problem still persists at the TCP Level. This is why HTTP/3 is taking a whole new approach, but this is out of the scope of this section. I would highly suggest to communicate through HTTP/2 for all of your REST communication, especially when you have high traffic between your microservices.

A second factor to take into account when talking about performance and REST, is that REST API needs to serialize and deserialize payloads in self-describing formats, typically JSON. I will assume the use of JSON for the rest of this paragraph.

Both JSON serialization and deserialization can add overhead to the performance of a REST API, as they require additional processing to convert the data between different formats. This overhead can be particularly noticeable for large payloads or for scenarios where high volumes of data are being transferred.

One way to reduce the overhead of JSON serialization and deserialization in a REST API is to optimize the design of the API to minimize the amount of data that needs to be transferred with each request and response. This can involve using compact data structures and minimizing the number of unnecessary fields or metadata included in the payload.

In comparison, gRPC (Google Remote Procedure Call) uses a binary encoding for data transfer, which can be more efficient than the text-based encoding used by JSON. This can make gRPC more performant than REST, particularly for scenarios where high volumes of data need to be transferred or when low latencies are important. However, it is important to note that the performance of any specific implementation of REST or gRPC will depend on a number of factors, including the hardware and software infrastructure used to host the API, the design of the API itself, and the volume and complexity of the requests being made. We will talk more about gRPC in the next part.

Overall, REST is a powerful and widely used tool for building APIs and facilitating inter-microservice communication, but it is important to carefully consider the potential disadvantages and limitations when deciding whether to use it in your specific use case.

gRPC

In this part, I'm going to dive into what is gRPC, why is it becoming a more and more popular option for inter-microservice communication and what are the pros and cons of using it. gRPC is a high-performance, open-source framework for building remote procedure call (RPC) APIs. It is based on the Protocol Buffers data serialization format, and uses HTTP/2 for transport. gRPC enables efficient communication between microservices, with support for bi-directional streaming, flow control, and flow-limited concurrency. Additionally, gRPC provides built-in support for load balancing, health checking, and error handling. Overall, gRPC is a powerful tool for building scalable and efficient inter-microservice communication systems.

Unlike REST, which operates in a resource-based manner, gRPC takes more of a function call approach. You define services, that exposes functions that you can call in multiple manners. There are four main ways to define a "function":

- Unary RPC: This is the simplest form of RPC, in which the client sends a single request to the server and the server responds with a single response. This type of RPC is typically used for simple, one-time queries or commands.
- Server streaming RPC: In this type of RPC, the client sends a single request to the server, and the server responds with a stream of responses. This is useful for situations where the server needs to send a large amount of data to the client, or fetching them on the fly.
- Client streaming RPC: In this type of RPC, the client sends a stream of requests to the server, and the server responds with a single response. This is useful for situations where the client needs to send a large amount of data to the server.
- Bidirectional streaming RPC: In this type of RPC, both the client and the server can send and receive streams of data. This is useful for situations where real-time communication is required, or when processing can begin as soon as with get our first piece of a payload.

Efficiency

gRPC is known for its high performance and efficiency, which are largely due to the use of Protocol Buffers as the data serialization format and HTTP/2 as the transport protocol.

Protocol Buffers, also known as protobuf, is a binary data serialization format that is smaller and

faster than other formats such as JSON or XML. It is designed to be language and platform neutral, allowing for easy cross-language communication between microservices.

HTTP/2 is a binary protocol that is designed to be faster and more efficient than HTTP/1.1, the previous version of the protocol. It allows for multiplexing multiple requests over a single connection, reducing the overhead of opening and closing connections for each request. HTTP/2 also supports server push, which allows the server to proactively send data to the client without a request, further reducing round-trip time.

Furthermore, gRPC provides more advanced features that helps improving the overall performance of the system, if well-used, such as request cancellation, health checking, error handling or load balancing.

Overall, the use of Protocol Buffers and HTTP/2, along with the features provided by gRPC, makes it a highly efficient and effective tool for building high-performance inter-microservices communication systems.

Multiplexing

Multiplexing is a technique used to send multiple requests and responses over a single connection.

In gRPC, multiplexing is achieved by using a single HTTP/2 connection to send multiple requests and responses. Each request and response is identified by a unique stream ID, which allows the server to distinguish between different requests and responses. This opens the door for multiple requests and responses to be sent over a single connection at the same time, reducing the overhead of opening and closing connections for each request.

This feature of multiplexing in HTTP/2, and thus gRPC, improves the efficiency of communication between microservices as it allows multiple requests and responses to be sent over a single connection, reducing the overhead of opening and closing connections for each request. It also allows for bi-directional streaming, which enables real-time communication between the client and server.

Additionally, the use of flow control (which is defined in the HTTP/2 RFC) in gRPC ensures that the server does not become overloaded with too many requests, thus improving the overall performance of the communication. This is done by limiting the number of requests that can be sent over a single connection, and by controlling the rate at which data can be sent for each flow (stream, request) to the connection. HTTP/2 also defines the concept of stream priority, meaning we can prioritize certain gRPC calls, if the need arise.

There are others very interesting features that HTTP/2 defines that allows for even better performance, such as stream dependencies or reprioritization. Since gRPC was conceived with HTTP/2 in mind, a lot of these features can be leveraged. I recommend reading the specifications for more details.

This ensures that resources are used efficiently and that the server is not overwhelmed.

Protobuf

Protocol Buffers (protobuf) is a data serialization format developed by Google. It is designed to be

small, fast, and efficient, and is used to transmit data between different systems and programming languages.

Protobuf uses a language and platform-neutral binary format to serialize data, making it smaller and usually faster to serialize/deserialize than JSON. Additionally, protobuf includes a code generation tool that generates data access classes for various programming languages, making it easy to work with protobuf data in those languages.

One of the key advantages of using protobuf over other binary serialization format is its ability to evolve over time. When a field is added or removed from the data structure, older clients can still correctly parse the data they receive, and newer clients can correctly parse the older data. This allows for a more gradual rollout of updates, and reduces the risk of breaking existing clients.

Protobuf also supports the definition of message types, services, and enums, which can be used to define the structure of the data and the methods that are available to interact with it. This makes it easy to understand the structure of the data, and how it should be used.

Protobuf is not self-describing, meaning that you can't discover how a message should be parsed when receiving it. This means you need to know that beforehand, that's why you need your protobuf definition for both your client and server. This comes with its advantages and disadvantages, primarily having a well-defined contract for your inter-microservice communication (reducing the risk of misunderstanding and informal coordination between teams), at the cost of a more rigid development constraints. This makes it robust and improves the trust you can have in your production environments, but make it more cumbersome to develop with. Changing your APIs becomes hard, and payloads are binary-encoded, which make it harder for prototyping and debugging. It's a tradeoff that more and more organizations are willing to make.

Overall, protobuf is a powerful and efficient data serialization format that is used for a wide range of applications. It is widely used in high-performance systems, including distributed systems and microservices. gRPC uses protobuf as the data serialization format, which makes it efficient for communication between microservices.

xDS Support

Cementing gRPC relationship in inter-microservice communication, it is gradually implementing the xDS protocol, with a great set a features already production-ready. xDS (Extensible Dynamic Configuration) is a set of APIs and protocols for configuring and managing services in a distributed system. xDS provides a common interface for configuring and managing services, regardless of their underlying technology. It is widely used in service mesh architectures and service discovery mechanisms, and it is the foundation of service management in Envoy (in fact, xDS was created by the Envoy team), a popular open-source proxy. To put it more simply, it allows for management of service meshes.

We are going to dive deeper into service meshes later in this section, but for now it is important to point out that, by making gRPC xDS-ready, it is enabling for proxyless gRPC service meshes, which is a huge deal in term of performance, but also security. It means that we don't need to spawn a proxy for each of our microservice to intercept all the communications our microservice does, redirecting them to the correct endpoint. We reduce a lot of performance overhead, both in term of IO throughput and CPU usage (we are reducing the hops between user space and kernel space,

although eBPF is starting to help quite a lot here) and memory footprint. We also greatly reduce the complexity of our architecture. This also allows to have a true end-to-end encryption, since TLS termination is done in the application, and not at the proxy.

You can keep track of the current state of xDS in gRPC here. Keep in mind that support might vary depending on the language you are using, since not all libraries are up-to-speed with latest gRPC specifications.

Overall, having the possibility to directly integrate with xDS opens for a world of possibilities, pushing even more the efficiency of this protocol.

Gateways

Although not in the scope of this section, I want to talk about one drawbacks that some see with using gRPC - even though it is not really on to me. gRPC can't be used in the browser. This is due to multiple technical factors, mainly around the freedom that web browsers give to their Javascript sandbox around the way they can manipulate raw HTTP packets. To put it simply, they can't. There is no way to force the usage of HTTP/2, and no way to dictate how streams should be handled within an HTTP/2 connection.

gRPC was never meant to be a communication protocol between your frontend and your backend. This is why I'm suggesting that it is not really a problem. However, this implies that you need a layer somewhere, that takes your frontend requests, and "translates" them to appropriate gRPC protocol. The most popular options are to have your microservice serving both a gRPC and a REST API (for internet-facing microservices) or to use gateways. I would recommend using the latest, since they allow you to have a single entry point into your system, although this introduce a single point of failure. You gain the benefits of having centralized control, a easier time introducing routing and load balancing, centralize your authentication and authorization and not have to deal with that in your microservices, caching and an single entry point in your service-mesh, if you have one. This however introduces additional hops to your request, additional overhead to your system and, as mentioned previously, a single point of failure.

It's important to evaluate the tradeoffs and decide wether the benefits outweigh the drawbacks in your use case and their specific requirements, while foreshadowing your architecture evolution, as with everything in this domain.

Support

The last thing I want to touch on, which is important factor whatever the solution, is the support and adoption. gRPC has gained significant popularity and community adoption in recent years, which is a trend that is forecasted to continue. It is an actively developed open-source project, with contributions from a large community of developers. It is supported by Google and has a growing ecosystem of third-party libraries and tools. gRPC provides several libraries and tools to help developers work with the gRPC framework, such as the Protocol Buffer Compiler (protoc), the gRPC C++, Java, Go and C# libraries, and the grpc-gateway for building RESTful APIs on top of gRPC services. Protoc can be configured to compile the boilerplate code for a wide variety of languages, while being highly customizable, allowing for different code generation per language, generating for a specific framework, for example.

gRPC, wrapped up

gRPC seems to come with a lot of benefits for few drawbacks when it comes to inter-microservice communication. It is now well-established in robust distributed system and is becoming a de-facto protocol to use as soon as you need to have some kind of performance or costs requirements for medium to large scale systems. It continues to develop on its microservice-oriented features such as xDS, and is now too big to fail in the upcoming years due to its sheer adoption in major companies, like Google, who are the project-owner and active maintainer of it.

The options, operation layer

We've talked about what protocol we could used to communicate between our microservice. However, it is important to remember that our microservices are going to evolve in a distributed system, with most likely a layer below them to manage them, such as Kubernetes. This layer comes with new options on managing your inter-microservice communication, since they abstract away network concerns. We can put systems in place to remove complexity from our application layer, that should focus on business logic, and move it to our operation layer, where we can manage some aspects about how our system is ran.

In this chapter, we are going to explore a few options we have at this layer, diving deeper in concepts we've touch on previously, such as services mesh. I am making the assumptions that we are running our applications in a Kubernetes environments, since it is effectively the only solution you orchestrate your microservices at a large scale nowadays.

Kubernetes' service discovery

Kubernetes' service discovery is a feature that allows pods and services to discover and communicate with each other within a Kubernetes cluster. It is an important aspect of service-to-service communication in a microservices architecture.

In Kubernetes, services are used to expose pods to the network and to other pods within the cluster. Each service is assigned a unique IP address and a DNS name, and is exposed on a specific port. Pods can communicate with a service using its IP address and port, and can use the service's DNS name to communicate with it if the DNS service is enabled.

Kubernetes also provides built-in support for service discovery using the Kubernetes DNS service. When the DNS service is enabled, each pod is automatically configured to use the DNS service to resolve service names to IP addresses. This allows pods to communicate with services using their DNS names, rather than having to know their IP addresses and ports.

This means that, instead of having to keep a list of every IPs of your replicas for a particulate microservice, you have only a single hostname, which will be resolved to a service that corresponds to an entry point for all of your pods for that microservice. It allows for simple load-balancing, dynamically adding and removing pods (enabling scalability), port-mapping as well as some more advanced features.

Overall, Kubernetes' service discovery is a powerful feature that allows pods and services to discover and communicate with each other within a Kubernetes cluster, it provides a built-in DNS service making it easy for pods and services to communicate with each other in a microservices architecture. It is usually sufficient for basic use-case, although the simplicity in its design makes it unable to have more advanced features that you could enable by introducing additional hops to operational applications, such as proxies. That's where service meshes come in.

Service mesh

A service mesh is a dedicated infrastructure layer for service-to-service communication in a microservices architecture. It is designed to manage the complexity of service-to-service

communication, and to provide features such as traffic management, service discovery, load balancing, and security.

A service mesh typically consists of a set of proxies, known as "sidecar proxies", that are deployed alongside each service instance. These proxies handle service-to-service communication, and they are what enables features such as traffic management, service discovery, load balancing, and security. The proxies are deployed through Kubernetes configuration, and are managed using a central control plane.

The control plane is responsible for configuring and managing the proxies, and can be used to monitor the health and performance of the services and proxies, and to provide detailed visibility into service-to-service communication.

Picture are worth a thousand words, so here is a (very simplified) overview about how service meshes operates in a Kubernetes context:

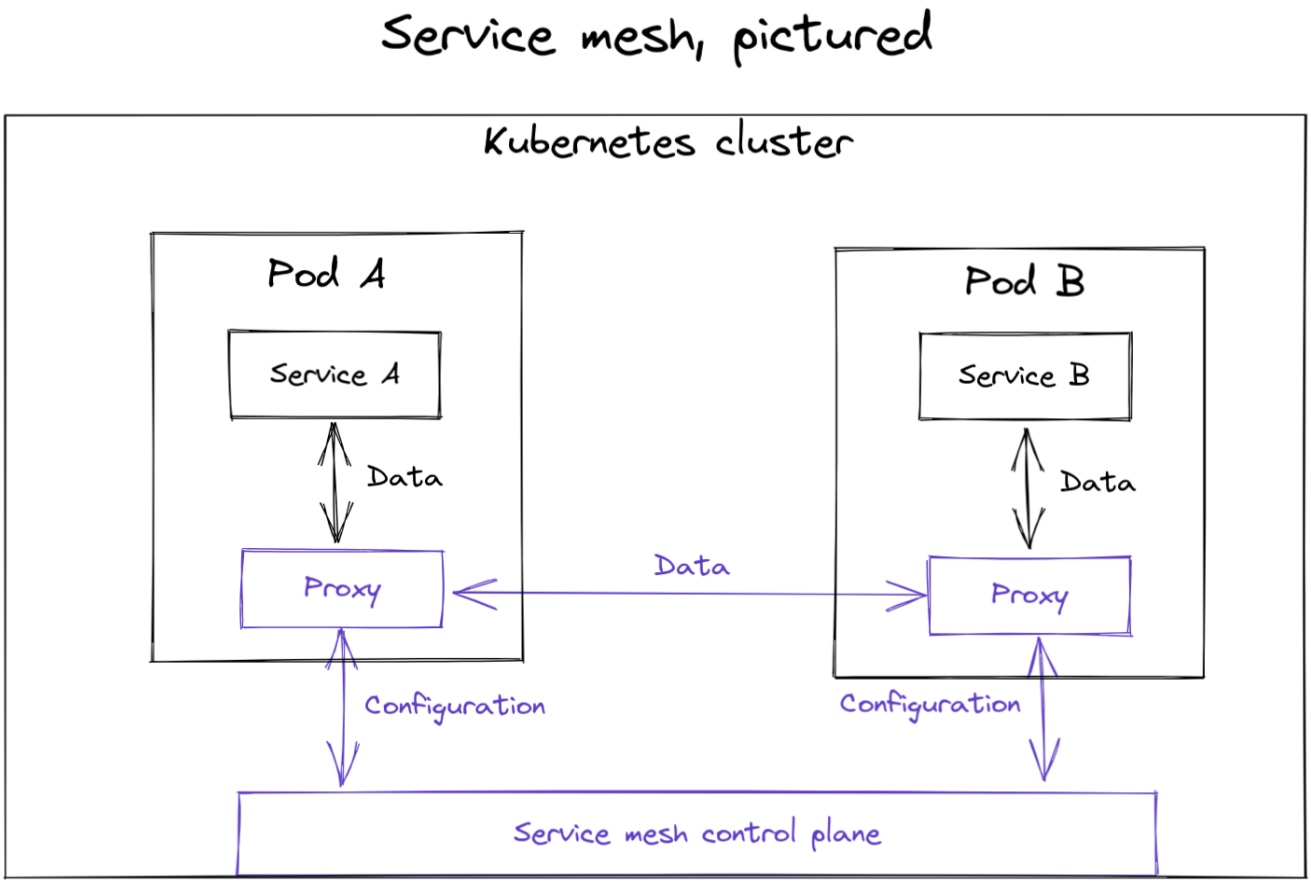


Figure 21. Service mesh, pictured

By introducing proxies into the mix, we gain significant control about how to manage our inter-microservice communication. This allows us to introduce encryption for packets transiting through the cluster, usually with mTLS. Traffic on the internet is usually encrypted with TLS, where a site you visit prove their identity by providing a certificate, that you can see like an ID Card. Using this certificate, the client can verify and trust the site, and initiate a negotiation with the server, to secure the communication via encryption. Typically, the user needs to authenticate with a username and password to access protected resources. mTLS keep the core of this process, except that the TLS negotiation is done both ways. This means that both microservices can trust that the other is who they say they are, moving the authentication process at the TLS level. During this process, they also negotiate a secure way to communicate.

What is very important to identify here, is that the identification is done at the session level, and the communication can be done normally after. There is no need to interfere with layer 7 data, since the authentication and encryption is done at lower level. Layer 7 is the application layer, so you can let your application freely send whatever data they might want to send.

Service meshes also provides way to implements resiliency and fault-tolerance features, such as retries and circuit-breakers. By having a proxy in front of each microservices, you can introduce advanced rules that dictates where packets goes, based on previous information, such as wether previous requests failed or not. Having this state and history, enables automatically sending new requests when previous one failed (retries), and identifying failing microservices and stop sending requests to this microservice for a while (circuit-breakers).

You can also test the robustness of your system, by injecting voluntary faults. This helps identifying bottlenecks and problems, before they actually arise. You basically have freedom about how you route your packets, with different solutions providing more or less features. This also allows real-time and precise observability about the state of your networks and packets traveling through them, once again at the operation layer, with no need to modify your application code.

To sum up, service meshes provide a transparent infrastructure layer for service-to-service communication, which allows developers to focus on building business logic, while the service mesh takes care of managing the complexity of service-to-service communication. The most popular service meshes are Istio, Linkerd and Consul. We are going to explore all three of them.

Consul

Consul is a service discovery and configuration tool developed by Hashicorp. It offers several features such as service discovery, health checking, key-value storage, load balancing, DNS interface, ACL system, event system, web UI and API. It is designed to be simple to operate, can be run as a single binary, doesn't require a separate data store, and can be run in a cluster for automatic failover. It is widely adopted and supported by a large community, it is also supported by many cloud providers and can be easily integrated with other Hashicorp tools. Consul Connect is a service mesh feature that allows for secure communication between services without requiring any code changes.

Consul connect uses Envoy under the hood, meaning that it communicates through the aforementioned xDS protocol. Where Consul Connects really shines is its integration with Consul, and the overall Hashicorp ecosystem. Configuration is done through a key-value store, making it easy to modify and configure on the fly. Consul Connect focuses on ease of use, with a nice UI coming bundled up with the solution, allowing you to easily edit your configuration and see the state of you system.

However, it can lack in advanced features and is considered as a solution coming with a lot of resource overhead.

Istio

Next up, I want to talk about Istio. Istio seems to be the more popular solution in production environment, thanks to its set of features and extensibility. Istio currently provides the biggest toolkit, with circuit breaking, retries and timeouts (both path and method based), fault injection,

delay injection, header and path based traffic splits, percentage based traffic splits, mTLS, RBAC and much more. Istio takes a declarative approach, meaning that configuration is done through the editing of Kubernetes' Custom Resource Definitions (CRDs). This has proved to be an easier way to automate deployments and configuration with existing infrastructure-as-code tools like Terraform.

Istio is also based on Envoy for its proxies, but is currently experimenting (with great results, although not production-ready) proxyless gRPC service mesh, by using the integration of the xDS protocol within gRPC. Istio's control plane can communicate directly with your gRPC application to configure it and route the traffic accordingly. This drastically reduces the performance overhead that comes with spawning an envoy proxy for each of your pods replicas. Istio also supports advanced routing for WebSockets and MongoDB.

Istio also comes with an UI that you can install that provides a very comprehensive real-time monitoring tool for your system. It is open-source and community-driven, integrated with most of the managed cloud ecosystem, and initiated by Google IBM and Lyft.

However, Istio might be the most overwhelming service mesh, making it the most difficult to manage and maintain. It can be challenging for teams that does not have the resources to explore the possibilities and configuration of Istio.

LinkerD

Another service mesh option, that is growing in popularity in recent years is LinkerD. It is fully open source, and a cloud native computing foundation (CNCF) project. LinkerD is designed to be non-invasive and is optimized for performance. It is really easy to adopt, and its sidecars proxies are very light and performant. It is an in-house "micro-proxy" written in Rust, but comes with a complete feature set that you can expect with services meshes, such as metrics, L4 and L7 advanced proxies, mTLS, retries, timeouts. The list of features is growing, with circuit breakers planned for the near future, for example.

However, I couldn't find a way to add a layer of compatibility with xDS, making it unusable with solutions that are developed around this protocol. This can be a major drawbacks depending on your needs.

Cilium

Lastly, I want to talk about cilium, which makes use of an interesting feature of the Linux kernel: eBPF. eBPF is a way to run your own code in a kernel sandbox. You can add and configure kernel features at runtime, allowing for example, to have some advanced routing logic. You can see where this is going, Cilium takes advantage of this feature to configure load balancing, mTLS and other advanced networking implementations at the eBPF layer, effectively never leaving the kernel, and avoiding two contexts switching between kernel space and user space in the process.

eBPF programs are executed in a sandbox, running on a JIT compiler. The performance are great, on the same order of magnitude as the kernel compiled code. More importantly, we have a very minimal memory and processing overhead, really squeezing out all the performance you can get from your resources, while still having service meshes' advantages.

Cilium is known to be quite difficult to configure, with a lot of custom resource definitions to set up

for enabling features, such as circuit breaking or retries. It can also lack some features such as fault injection. Cilium can also work with Envoy, but I would argue that if you were to use cilium with envoy, you are better off using another service mesh. I would advise using cilium when you have strong performance requirements.

As always, the best solutions depends on your specific requirements. Heavy performance constraints might directs you towards LinkerD or Cilium, complex and complete observability and configuration features towards Istio and ease of use towards Consul Connect, for example. Now that we have skimmed over the options available, we are going to take a closer look at what would make sense, and what are the tradeoffs we are willing to make for Polycode.

Polycode

With a better understanding of the solutions available, and with the overall concepts down, we are now going to talk about Polycode. This paper is about migrating Polycode to microservices after all. Firstly, I'm going to talk about what I suggest we use as technologies in our future Polycode architecture, and why those choices makes sense to me, by explaining my thoughts. I'm then going to show you some diagrams to better illustrate how my implementation would look like, as well as to make sure that the inner working of the system is understood. Lastly, I'm going to introduce you to a proof of concept using the technologies I chose, to show how it can be implemented.

My suggestions

Let's talk about technologies choices, starting with gRPC.

gRPC

I suggest that we use gRPC for all our inter-microservices communication. There are thoughts behind this suggestion, let's get into them.

To begin with, let's talk from a technical standpoint. On paper, gRPC is efficient, fast, provides a good developer experience and integrates natively with xDS. Performance is not an absolute requirement in our use case, but it is always important to keep it in mind when designing your system. Efficiency implicitly means a better usage of your resources, which are more often than not costly. This is not our case, but it is something to keep in mind when making your decisions. You might want to test the performance benefits of switching to gRPC in your environments to have some numbers you can compare and make a better educated decisions. I have a repo that provides some very rudimentary code to test the performance benefits of using gRPC vs REST written in Rust. Beware, it is not documented, although very simple. It might be useful as some boilerplate to build your own test. As a reminder, gRPC efficiency does not only come from raw performance, but also from multiplexing and what comes with HTTP/2 altogether.

Another major benefit of using gRPC is the duplex communication that it enables, both with unary or streaming request/responses. This can be very useful and important in a lot of scenarios. It enables, for example, streaming standard output and errors from the runner to the submission microservice, that can then stream this to the end user using websockets. Use case may arise, and using gRPC future-proofs our communication protocol.

I also see gRPC as a very powerful tool for synchronizing work between teams, which has historically been a hassle for us. By forcing us to define protobufs, we have a clear data structure that is going to be sent across our service. Better yet, with protoc, we will be sure that we are going to use the same structure in our code. This is arguably also possible with a REST API, using tools around the OpenAPI ecosystem. But we are too often constrained by time and we lack enough discipline and experience to identify that defining an API is often the most important part of the work, because a well-defined API means that we are not going into coding blind, that we have thought about caveats and edge cases, and more generally, architected our system. This might be looked at as a drawback to gRPC by some people, but in our case having this rigidity will be a blessing in a disguise for our team.

There are, however, also drawbacks of using gRPC in our case. I identified 3 of them:

- Our existing codebase has no gRPC code at all. This means we will need to rewrite all our code. This also might be a blessing in a disguise, since this is a way to eliminate the some of the technical debt that accumulated over time. As our first major project, we made a lot of mistakes in the way our code is structured, organized, and how it operates. This will also be a way for us to rethink about the mistakes we've made, reflect on them, and find ways to avoid them in our migration towards microservices.
- As we migrate our whole system towards microservices, we will inevitably encounter problems. Whether it is conceptual misunderstanding of the underlying technologies or simple bugs in our code, there needs to be strong debugging tools to help us find the problems. I would argue that gRPC is lacking in this field, compared to a REST API, where you can see in clear-text the payloads that are sent. This is useful for debugging and understanding what is transiting over the network, and to identify errors. This is due to gRPC being binary-encoded, and is a trade-off you have to do for the performance it provides.
- Lastly, during the migration process, we will be incentivized to copy-paste the old code. This is bound to happen, even if we clearly define it as a bad idea. This will lead to bringing bugs over, but more importantly, it will hinder the change of line of thoughts we need to have. REST APIs and gRPC Services are different concepts, and you don't build them in the same way. Since most of use has only used REST APIs as a way to communicate with other services, the transition is going to be difficult and we are going to be ending up with REST patterns in our gRPC services.

Overall, gRPC makes a lot of sense as our communication protocol and I would advise to use it as our sole synchronous inter-microservice communication protocol, mainly for its efficiency and rigidity, although the migration of the code base will be tedious for the team.

Istio

After some careful thinking, I've settled on using Istio at the operational layer. Let's discuss why.

Istio provides the biggest set of features out of all the service meshes solutions. Although we are probably not going to use them all, I think they provide interesting solutions for us to explore in term of load test in resiliency testing. We currently have very little users and it's hard for us to test the scalability and resiliency of our systems. Having artificial failures will help us better understand and identify bottlenecks in our system before they start to happen in real scenarios.

Using service mesh, and so Istio, we can improve the resiliency and fault-tolerance by easily implementing retries, circuit-breakers and timeouts. These patterns are usually requires a lot of technical, non business related code, when implemented at the application layer. We lift a big weight off our shoulder by not having to make a choice between time (in the form of coding) and unreliability.

Istio can be challenging to configure, but the main advantage is that its configuration is built in a declarative manner, using Kubernetes' CRDs. They can be scary to familiarize with, but the huge benefit for us is that it can be templated with Helm. We are using Helm to manage our deployments, and adding new templates to manage Istio resources into them is a matter of minutes. Configuring it is the hard part.

However, let's not forget the context of this project. This is an educational project, and setting some advanced challenges like this is hugely beneficial for us, if we manage to correctly understand what is going on behind the scenes and the concepts that revolves around Istio. Let's not forget that Istio is the most popular service mesh on the market today, and it is likely to stay this way for quite a while. Learning to work with it is a valuable skill, and will be useful for most of us in the future.

We are currently running on bare kubernetes service discovery, with no service mesh in-between. Although it is sufficient for what we are doing, the benefits and reasoning stated above is, for me, enough to justify its usage.

Since I've suggested using gRPC as our inter-microservice communication protocol, both of them blends in nicely with the proxyless features of Istio for gRPC services.

Unlike with gRPC, this suggestion is less about logical reasoning at a technical standpoint, but more about learning and gaining experience on a technology that we are likely to use in our professional careers.

Other suggestions

Unrelated to the technological suggestions above, we also need to focus on code quality, by delivering high-quality code, that is compatible with a microservice architecture. Code reviews are going to be important, and listening to everyone's suggestions and feedback will be more important than ever. We need to collaborate and push a environment where we foster upfront thinking about our architecture. We need to listen to everybody's worries and manage learning curves for each technology and the different burden it can have on some people.

Sequence diagram

To better illustrate how the flow of a request would be, I've made a sequence diagram highlighting all the hops a request will have to go through in our system. This is simplified and doesn't take into account how the proxies are configured, advanced configuration, and doesn't show the full scope of the request from a business logic standpoint. It focuses on the topic of this section, inter-microservice communication:

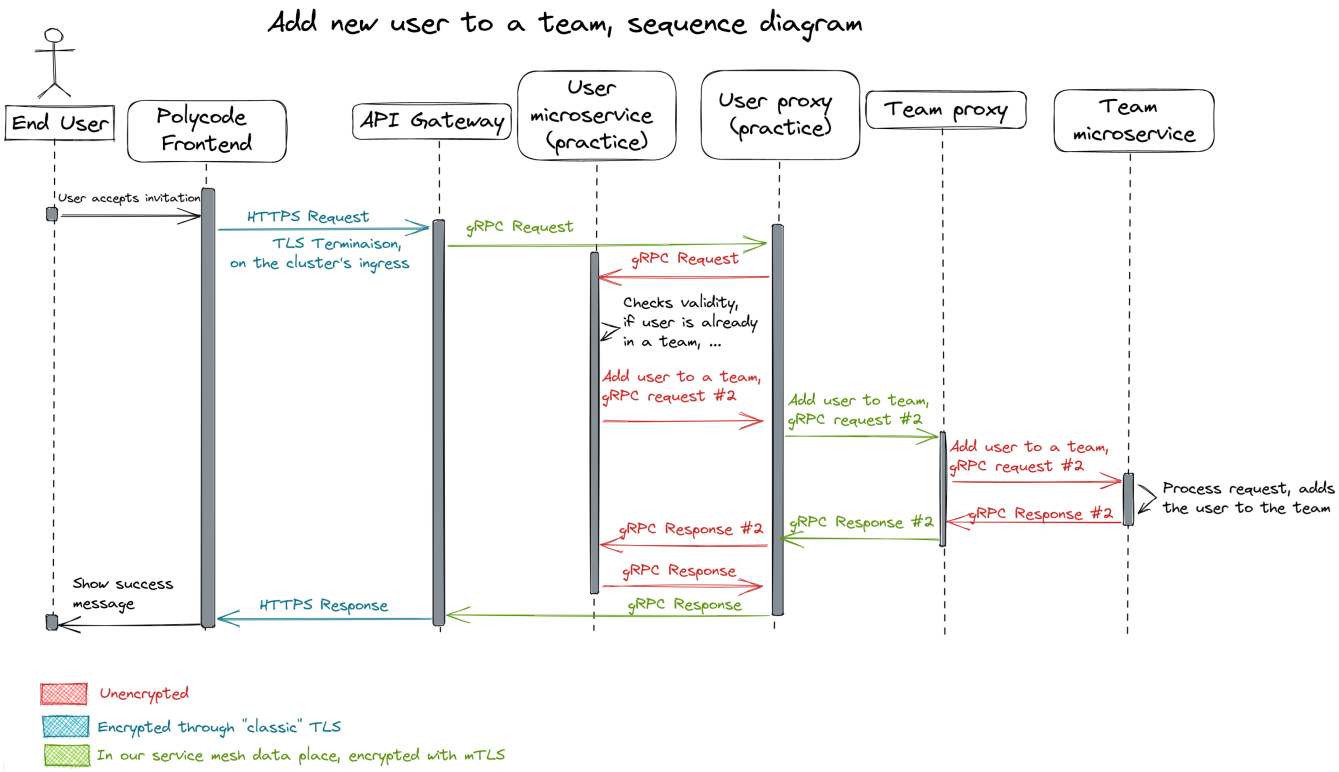


Figure 22. Inter-microservice Sequence Diagram

We can see all our actors here:

- The gateway into our system, that accepts HTTP Request and forwards them with the appropriate gRPC service. This could be an application, but it also needs to have a foot in our application layer, since this is also the entry point into our service mesh. Istio allows to create gateways that handles quite a lot of technical logic, such as routing and rate limiting, and we should offload what we can to Istio. I would still have an application behind this though, that will be able to implement complicated business logic related concerns, such as fined-grained authentication and authorization.
- The proxies of our service mesh. These are the sidecars proxy that we create for each of our replicas. In Istio case, these are Envoy proxies. They take care of the communication layer for our microservices. All the traffic downstream is within our service mesh data plane, and encrypted. The upstream traffic goes to our microservices, unencrypted.
- Our microservices. These are our business logic units, that exposes gRPC services. To talk to other microservices, they use gRPC. However, they don't have to worry about service discovery, load balancing and general communication concerns, since the proxies are handling these concerns for them.

This sequence diagram shows pretty well the additional hops a request has to go through with a classic service mesh deployment. Each gRPC request/response goes through at least 3 hops when communicating from microservices to microservices: one from microservice A to its proxy, then from proxy A to proxy B, and finally from proxy B to microservice B. This is where the performance overhead of service meshes comes in, and why eBPF-based or proxyless service-mesh are relevant solutions when tight on resources, since they eliminates these additional hops while retaining most of the features.

Proof of concept

As a way to demonstrate how my technological suggestions would operate, I've made a simple proof of concept (PoC), that deploys 2 microservices and 1 gateway, all within an Istio service mesh in a Kubernetes cluster.

You can find the repo here. You will find further explanation in the readme, as well as how to use it and deploy it on your own.

Conclusion

Inter-microservice communication is an essential part of designing your system. There are a lot of issues and concerns that arise from building a distributed system, that you don't find, or at least to a much lesser degree in a monolith application. We have identified these concerns, explained why they are important. Using protocols, technologies and patterns, we found ways to solve these problems, and settled on a few options that fulfills the requirements we have with Polycode. Inter-microservice communication is a critical aspect of an infrastructure and needs careful thinking, it is the backbone of your whole system.

Identifying problems and having ways to understand the state of your system goes hand-in-hand with this aspect. How do you know if your system is running as expected ?

Tracing and logging

Introduction

Managing a distributed system like a microservices architecture can introduce new challenges, particularly when it comes to monitoring and debugging. When a request is made to a microservices system, it may involve multiple services working together to fulfill the request. This can make it difficult to understand exactly what happened to the request as it made its way through the system.

Monitoring a distributed system is key to understand where you system is weak. Without monitoring, you have no idea of the state of your system, where are the errors coming from, why some requests may fail, and what needs to be done to fix this.

There are three pillars that are essential for effective monitoring of a microservice system: traces, logs, and metrics. Those three pillars are also referred as telemetry.

- Traces provide a detailed view of the path of a request through the system, showing the sequence of events that occurred and how long each step took. They can also carry data added at each step, to have a further understanding of the request.
- Logs provide a record of what happened within each service, including any errors or issues that may have occurred. Logging is going to provide actual errors and this is usually where you can really understand what is going on in your system.
- Metrics provide a high-level overview of the system's performance and resource usage, helping you to identify trends and potential issues. Metrics can play a big role in scaling effectively your system (but you should also take into account business metrics, not only resource usage).

By combining these three pillars of monitoring, you can gain a comprehensive understanding of your microservices system and how it is functioning. This can be crucial for identifying and resolving issues, optimizing performance, and ensuring the overall reliability and stability of your system.

Let's dive deeper into two of those: traces and logs, and see how we can implement a good solution into our Polycode system.

Tracing

Tracing is a technique that helps you understand the path of a request as it flows through a distributed system, such as a microservices architecture. It involves assigning a unique identifier to a request and propagating that identifier throughout the system as the request is processed. This allows you to see the complete lifecycle of a request and understand how it flowed through the system.

It helps tracking Service Level Indicators (SLI) of your system, with the helps of metrics, depending on the specific SLI. A SLI represents an information about your system, such as the time your page takes to load. SLIs are often linked to Service Level Objectives (SLO), which are a targeted objective that a service tries to provide. This all links to Service Level Agreements (SLA), which defines the standard a customer can expect from a provider. We might guarantee 99.9% uptime to our customers, that is our SLA, with a SLO that is 99.99% uptime for critical microservices, and we track this data using SLIs.

The terminology I will use is the following:

- Span is a unit of work within a trace. It typically corresponds to a single service operation or network request. Spans are typically identified by a unique span ID. Spans can also include metadata, such as timestamps and tags, which provide additional information about the span and how it fits into the overall trace. A new span is usually created for every microservice the request hop through.
- Trace is a series of spans that represent the path of a request through a distributed system. Each span in the trace is connected to the one before it by a parent-child relationship, which shows how the request flowed through the system. Traces are typically identified by a unique trace ID.
- Baggage is a term used in distributed tracing to refer to a piece of data that is carried along with a trace or span as it flows through a system. Baggage is often used to include contextual information about a trace or span that may be useful for debugging or analysis. For example, you might use baggage to include information about the user who made a request, the environment the request was made in, or the application version that was in use when the request was made.

Tracing can be a powerful tool for finding and debugging problems in a microservices system. By being able to see the complete lifecycle of a request and how it flowed through the system, you can identify bottlenecks, errors, and other issues that may be affecting the performance or reliability of your system. Tracing can also help you understand the impact of changes to your system, such as when you deploy a new version of a service or make changes to the system’s architecture.

Logging

Logging is the process of recording events and messages that occur within a system. In a microservices architecture, logging is an important tool for understanding what is happening within each service and how the services are interacting with each other. By capturing log messages at various points in the system, you can gain insight into the behavior and performance of your system and identify any issues or problems that may be occurring.

There are a few key concepts to understand when it comes to logging in a microservices system:

- Log message: A log message is a record of an event or message that occurs within a system. Log messages typically include a timestamp, a severity level, and a message string that describes the event.
- Log level: A log level is a way to classify log messages based on their importance or severity. Commonly used log levels are "debug", "info", "warning", and "error".
- Log sink: A log sink is a destination for log messages. Log sinks can be local files, databases, cloud storage, or any other location where log messages can be stored and accessed.
- Log aggregation: Log aggregation is the process of collecting log messages from multiple sources and storing them in a central location. This can be useful for making it easier to search and analyze log messages from across the system.

By carefully designing and implementing your logging strategy, you can use log messages to understand what is happening within your microservices system and identify issues and problems as they occur. This can be crucial for ensuring the reliability and stability of your system and for quickly identifying and resolving any issues that may arise.

OpenTelemetry

During this section, we will focus on solutions based on OpenTelemetry (which I will refer as OTEL). OpenTelemetry is an open source observability framework created to help developers instrument, generate, collect, and export telemetry data (such as metrics, traces, and logs) for their applications. It aims to provide a vendor-neutral and consistent way of generating, collecting, and exporting telemetry data across a variety of programming languages, platforms, and clouds.

OpenTelemetry was created through the merger of two open source projects: OpenCensus and OpenTracing. OpenCensus was initially developed by Google in 2016 as a way to standardize the collection of telemetry data across Google's services. OpenTracing, on the other hand, was developed by a group of companies in 2015 as a specification for distributed tracing of applications.

In May 2019, the OpenCensus and OpenTracing communities decided to merge their efforts and create a new project called OpenTelemetry. The goal of the merger was to combine the strengths of both projects and provide a single, unified observability framework that could be used by developers across a variety of platforms and languages.

Since its creation, OpenTelemetry has gained widespread adoption and is now supported by many cloud providers and open source projects. It has also become the de facto standard for observability in cloud-native applications.

You will most likely encounter proprietary solutions in the wild, but they tend to phase out due to the eagerly-adopted OTEL solution. OpenTelemetry offers a standardized protocol for metrics, traces but also logs to collect data and centralize them, and the industry has recognized the power of having an open-source standard. More and more solutions (even proprietary one) are moving to using this standard. Before OpenTelemetry, there were a number of different tools and approaches for collecting and exporting telemetry data from applications. Many companies and organizations developed their own proprietary solutions for this purpose, and there was no widely accepted standard for how to do it.

Since OpenTelemetry looks to be dominating the market and is pushing to be the one and only standard new applications and system are willing to use, I suggest to use it for all new applications that requires tracing, logging and metrics, such as Polycode. This is why I will now focus on OpenTelemetry as the de-facto framework we are going to use.

Vocabulary

Let's define some key vocabulary revolving around OpenTelemetry. This is not a exhaustive list and should be seen as a completion of the vocabulary previously defined.

- Instrumentation: The process of adding code to an application to collect telemetry data. In OpenTelemetry, instrumentation is typically done using SDKs or libraries provided by the project.
- Exporters: Components that receive telemetry data from an application and send it to a specific destination, such as a monitoring service or a log aggregator.
- Collection: The process of gathering telemetry data from an application and sending it to a

destination for storage or analysis.

- Sampling: The process of selecting a subset of telemetry data to be collected and exported. Sampling can be used to reduce the volume of data being collected and exported, and can be based on various criteria such as the rate of data being generated or the importance of the data.
- Context propagation: The process of carrying telemetry data, such as trace and span information, between different components or services in a distributed system. Context propagation is often used to maintain a consistent view of telemetry data as it moves through a system.
- Correlation: The process of linking together telemetry data from different sources or at different points in time to understand the relationships between them. In OpenTelemetry, this is often done using trace and span IDs.
- Tagging: The process of adding metadata to telemetry data in the form of key-value pairs. Tagging can be used to add context or additional information to telemetry data, and can be used to filter or group data when analyzing it. This can also be seen as baggages.
- Exporters: Components that receive telemetry data from an application and send it to a specific destination, such as a monitoring service or a log aggregator. OpenTelemetry provides a variety of exporters for different platforms and destinations.
- Tracing backend: A service or system that receives trace data from an application and stores it for analysis and visualization. OpenTelemetry supports a variety of tracing backends, including open source and commercial options.

Logging with OpenTelemetry

Historically, traces and logs were often treated as separate types of data and had different tools and systems for collecting, storing, and analyzing them. This could make it difficult to understand the relationships between trace data and log data, and to use them together to understand the behavior and performance of a system.

Making logs context and request-scoped would make them much more valuable and would allow for a much easier time to understand where your bottlenecks and errors comes from. This is why OTEL has been pushing solutions to include logs with traces and metrics to better understand and analyze what's going on in your system. We end up with correlated telemetry, which is enriched with data coming from the three pillars of monitoring.

Overview of a typical OpenTelemetry system

Here's a simplified overview of how a OTEL system typically works:

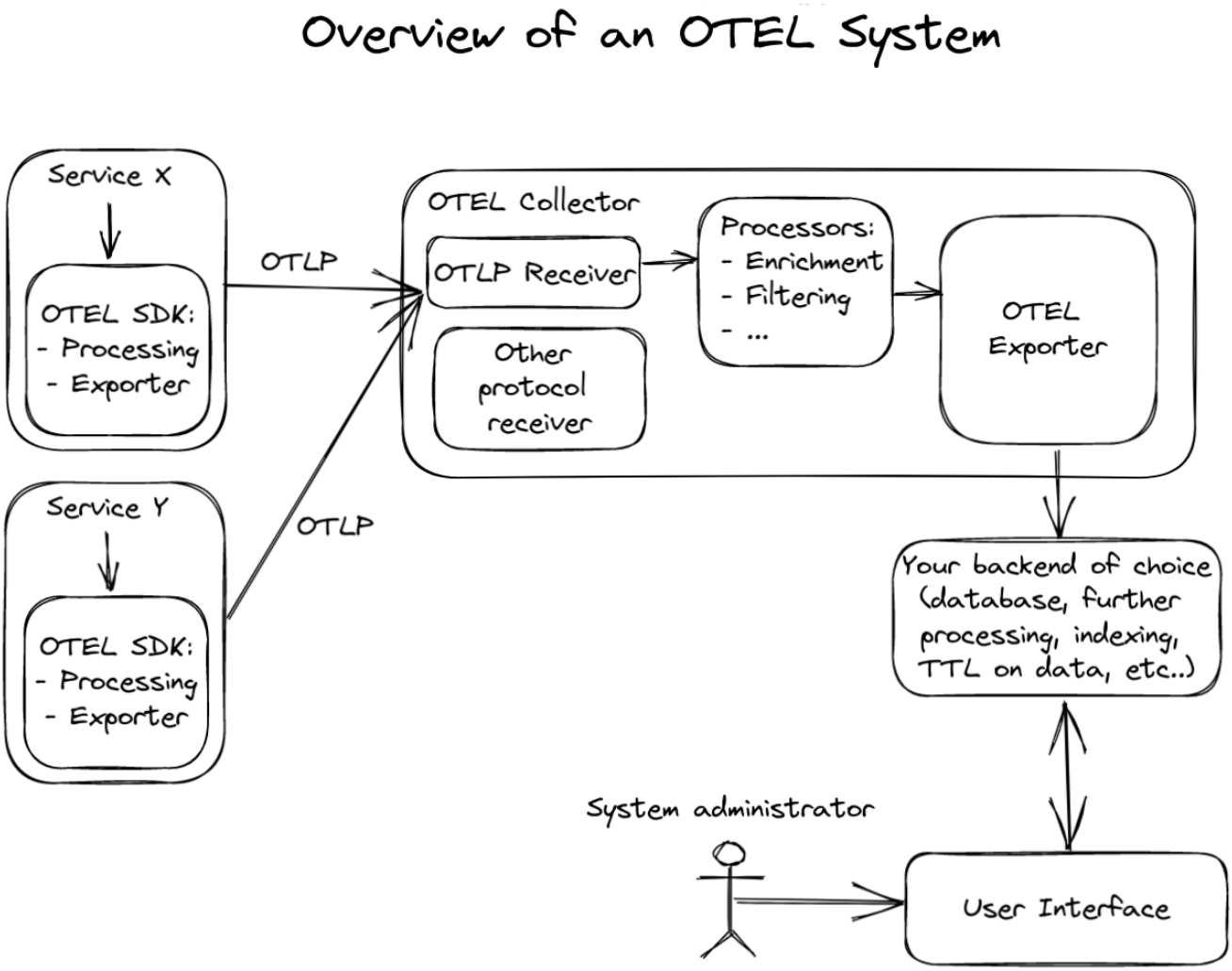


Figure 23. Overview of typical OpenTelemetry system

Each of your application usually integrates an OpenTelemetry SDK, allowing to easily integrate the protocol and the necessary processing within your application. For each requests, your application should notify your SDK of a new span, and take the necessary action. Your SDK will emit data to an OTEL Collector, which is a single centralized endpoints for all your application to communicate with. This collector will do some processing, and export its data to the backend of your choice. Once in here, you can access your User Interface and check your traces and logs.

As mentioned above, this is heavily simplified, and depending on the tools and configuration you choose, you can have some of these components coming in a single package. This diagram shows where and how you can choose multiple solutions, which can all work together thanks to the standardized OpenTelemetry Protocol. Usually, your OpenTelemetry SDKs within your application are totally unaware of the collector you're using, except that it understands the OpenTelemetry Protocol.

Your collector can also support other tracing backend, such as the Jaeger or Zipkin protocol for example. With this added layer, you can customize and choose how to handle your data before storing them into your backend. It is common to have OTEL tools that comes with your collector, your backend and your user interface in a single package. It is usually possible to customize and replace each of these bricks in such systems.

Tools

Visualization tools

One of the important aspect when monitoring the system, is not only to collect valuable information, but also have great tools to visualize and search this data. In this section we will look at the most complete solutions available on the market.

The ELK Stack

The Elastic Stack (formerly known as the ELK Stack) is a set of open source tools for collecting, storing, and analyzing logs and other types of data. It is developed by Elastic, a company that provides a range of products and services for search, analytics, and observability.

The Elastic Stack consists of the following components:

- Elasticsearch: A distributed, RESTful search and analytics engine that is used for storing and indexing logs and other types of data.
- Logstash: A data processing pipeline that can be used to collect, parse, and transform logs and other types of data before storing it in Elasticsearch. Logstash is highly configurable and can be used to process a wide variety of data sources.
- Kibana: A web-based visualization tool that can be used to analyze and visualize data stored in Elasticsearch. Kibana provides a range of features for searching, filtering, and visualizing data, and can be used to create dashboards and alerts for monitoring and alerting purposes.

The Elastic Stack is widely used for collecting, storing, and analyzing logs and other types of data. It is particularly popular in environments where there is a large volume of data being generated, and is often used in combination with other observability tools such as APM (Application Performance Management) solutions and tracing systems.

In terms of traces, the Elastic Stack can be used to store and visualize trace data in a number of ways. For example, trace data can be collected using an OpenTelemetry exporter and sent to Elasticsearch for storage. Kibana can then be used to visualize and analyze the trace data, using features such as filtering, aggregation, and graphing. The Elastic Stack can also be used in combination with other tracing tools, such as Jaeger or Zipkin, to provide a more comprehensive view of trace data.

Jaeger

Jaeger is an open source distributed tracing system developed by the Cloud Native Computing Foundation (CNCF). It is designed to help developers understand the behavior and performance of distributed systems by providing a way to collect and visualize trace data.

In the context of OpenTelemetry, Jaeger can be used as a tracing backend to store and analyze trace data generated by OpenTelemetry-instrumented applications. OpenTelemetry provides a Jaeger exporter that can be used to send trace data from an application to a Jaeger server for storage and analysis.

Jaeger provides a number of features for visualizing and analyzing trace data, including a user-friendly web interface for searching and filtering traces, and a variety of different visualization options for understanding trace data. It also has integrations with a variety of other observability tools, such as Prometheus and Grafana, which can be used to extend its capabilities or to integrate trace data with other types of data.

Jaeger uses a distributed architecture, with a collector component that receives trace data from clients and stores it in a storage backend, and a query component that provides a web interface for searching and visualizing trace data. The collector and query components can be deployed separately, allowing for flexibility in terms of scale and performance.

It natively supports two NoSQL databases as trace storage backend : Cassandra, and Elasticsearch. This means you can integrate Jaeger with your ELK stack. It also has backward compatibility with Zipkin (which has historically been the go-to tracing system, but is being more and more replaced by OTEL). It is not relevant to us, in our Polycode application, but it is important to note as it might be something you're looking for.

Polycode

I've decided to use the ELK stack for the Polycode application. This is because it is a complete solution, that allows for collecting logs, metrics and traces and correlating them together. This is however, a big stack and will consume resources, with the added benefits of being highly scalable. Traces, metrics and logs can easily be exported using adapted OTEL exporters in the code. The elastic ecosystem provides a multitude of tools to handle traces, metrics and logs their own way, but I've decided to use OTLP for exporting data out of my application. This way, I've not closed the door for other monitoring system if we decided that this stack wasn't suited for us or if we identified some problems with it. We could very easily plug a Jaeger collector instead of the APM Server and have our monitoring stack working again. APM has been supporting the OpenTelemetry Protocol for a while now, adding supports for the more recent logging protocol in version 8 of APM. Collecting logs this way eliminates the need to have an additional pipeline for logs. If we were to use the recommended elastic's log collection process, we would need to spin up a Filebeats for every application, sending their data to Logstash. The main advantage of using this process is that you can also easily export kernel logs from each of the container, but I don't think it is relevant and worth the hassle in Polycode.

Deployment

One important aspect to keep in mind, is that your monitoring system will be at the heart of debugging and discovering problems as soon as possible in your application. This means you need to have it up and running even if you have a major outage in your primary system. This is why I think your whole stack should be separated in an external system. This also ensure that your monitoring and operational system does not impact your application performance.

This is the deployment diagram:

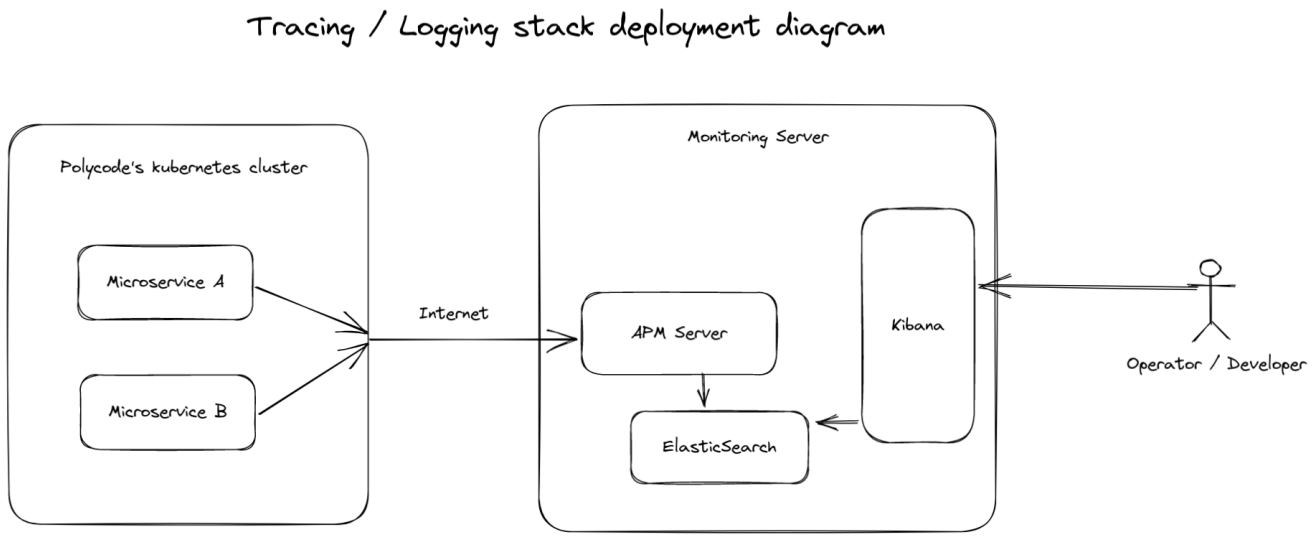


Figure 24. Deployment diagram of the monitoring system

As you can see, the whole monitoring stack is totally separated from the main application cluster, to make sure not to impact it, and to retain observability in the case of a major system failure. I've decided to run the ELK stack on only one ElasticSearch node, mainly to not over-complicate the already complicated architecture. It is certainly regrettable if we were to lose our data, as we would

lose the ability to correlate trends in our system over time, but this is data we can rebuild over time.

Sequence diagram

To better understand the flow of the system, I've decided to show you a sequence diagram of the action of accepting a team invite by a user. Please bare in mind that I've not shown any error case in this sequence diagram, but any error should be also traced and sent to the APM Server. Here's the diagram:

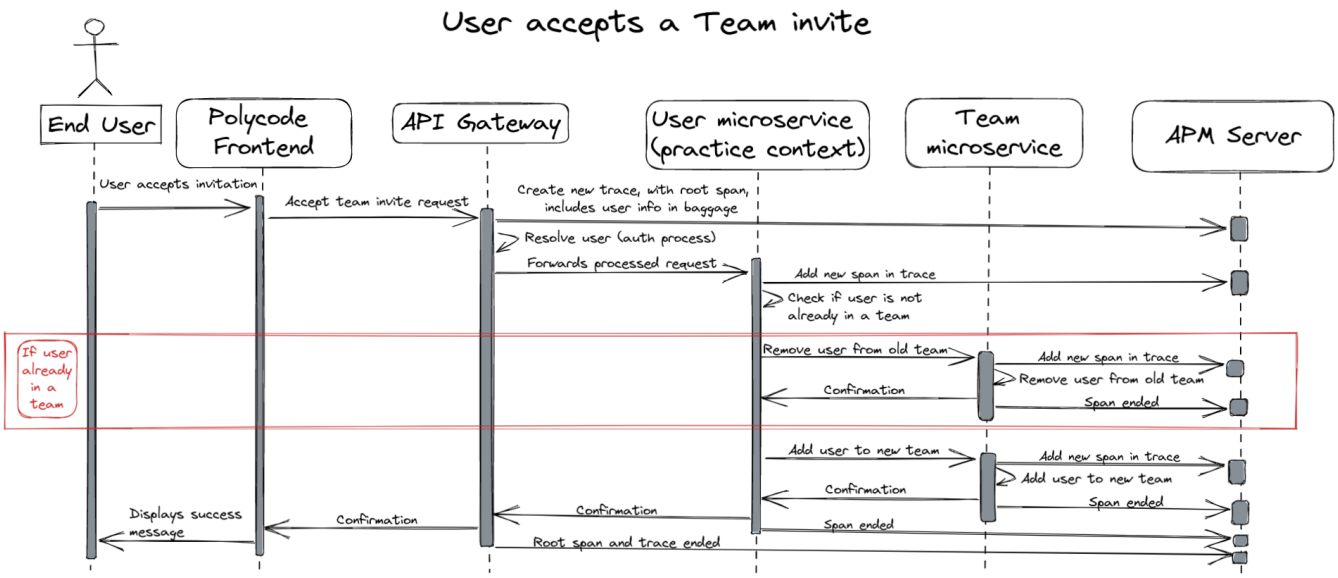


Figure 25. Sequence diagram of a request traced in our system

We find all our application components that are actors in this requests:

- The application frontend, which will take users action and send appropriate requests to the gateway.
- The API gateway, which is responsible for taking the HTTP Request of the frontend, authenticating the user if needed via its authentication cookie, and redirecting the request to the correct microservice as a gRPC request.
- The User microservice (practice context, not account context), which is our transactional boundaries for all operations done on an user in the practice context. To mutate a user and a team state, we must go through this aggregate. If it is not clear, or if you don't recall why, I suggest you refer to the first part of this paper and we go in details in the hows and whys of this architecture.
- The team microservice, which is responsible for handling the business logic of our team in our system.
- The APM Server, which in our stack corresponds to the OTEL Collector, and is in direct communication with elasticsearch to process and store the traces.

All communications with the APM Server is done with the OpenTelemetry Protocol (OTLP).

Once the requests reaches our backend, and more specifically our API Gateway, a new trace is created. The gateway creates its span, which will be the root span of the request, and since it is able

to resolve the user with its authentication cookie, it will include its information as baggage in our trace. It then forwards the request to the user microservice, which will create a new span within the trace. Each hop on a microservice will create a new span. This is important to understand the flow of the request through the system. This principle trickles down to the Team microservice, with each corresponding spans being closed when the response is sent back.

As mentioned above, if an error where to occur, it is important to bubble up the error to our trace, and a correct implementation of our tracing system in each of our microservice is crucial to achieve this goal. You usually want to consult traces when you have errors on your system, and if you can't find them or if they are incomplete, you are probably not using your SDKs properly.

If any logs were to be printed during this process and within the scope of this trace, the Trace ID and the Span ID would be included and sent to the OTEL collector to its log endpoint. This way, we can easily correlates logs with traces.

Not shown in this diagram is the subsequent processing of the request, how it is stored, indexed, filtered, etc.. This is specific to each implementation, and configuration of your solution. It doesn't matter here, since the overall sequence would be the same. In fact, you can replace our APM Server by any OTLP-Compliant collector and you will still have a working system.

If a system administrator wants to look at details of this request, they can log into Kibana, search for this specific request with appropriate filters, and see how long each spans took, the logs related to it, which user initiated the request, the eventual errors and other related information.

Conclusion

Observability is a key component in a distributed system, it is a crucial component to have insights about what is happening in your system. The more complex your system grow, the more you will be in the dark about how it behaves without implementing the necessary observability tools. I've decided to go towards the OTEL Standard in Polycode, for its standardized instrumentation, wide range of library, and for being open source and vendor neutral, despite the increased complexity of the system and its relatively recent ecosystem, although its production ready and pretty mature. As actual solutions and implementation, I've decided to go with the ELK stack (without Logstash, since we're using OTLP log support for logs), for its matureness, robustness, scalability, despite the paid features and the heaviness of the stack.

Search Engine

Introduction

In this section we will look at how we can integrate a search engine within Polycode. We will explore, why we might want a search engine, what is a search engine, what are the solutions, which one I chose for Polycode, and some examples with diagrams and proposals for the layout of the frontend.

What is a search engine

In the context of a web application, a search engine is a piece of software that allows the user to search the application's content for specific terms or phrases. It typically consists of a search interface that allows the user to enter their search query, and a search engine that processes the query and returns a list of results.

A search engine usually works by indexing the content of the web application (in our case, Polycode). When a user performs a search, the search engine looks for matches to the search query and returns a list of results. The search engine may use various algorithms and techniques to determine the relevance of the results, such as analyzing the frequency and proximity of the search terms within the content, the presence of synonyms or related terms, and the quality and authority of the source of the content.

Some search engines also allow users to specify additional parameters or filters to narrow their search, such as date range, language, or location. They may also offer features such as spell check and autocomplete to help users refine their search queries.

In addition to searching the content of a web application, some search engines may also search metadata associated with the content, such as the title, description, or tags. This can be useful for finding content that may not contain the search terms but is still relevant to the search query.

Why would we want it in Polycode

Currently, Polycode is populated with not a lot of contents, but it can already be quite hard to find what you are looking for. With the content list growing, and with more and more languages and modules you can learn, having a easy way to find what you are looking for becomes a primordial feature.

A web application user experience is not only defined by the features and design of the site, but also by the ease of browsing the website. If we can't provide an easy way for our users to look up specific terms, we are lacking a significant brick of user experience. A search engine can also help users discover new content that they may not have otherwise found. By providing suggestions for related or similar content, a search engine can help users explore the Polycode's offerings and discover new resources that they may find useful. If we open content creation to the regular users, some resources will be buried under the most popular contents and modules on Polycode, even though some niche community-made content might be the most fitting for a user particular need.

By making it easier for users to find the content they need, a search engine can help users feel more satisfied with the web application and encourage them to continue using it.

How to implement a search engine ?

We will now look at the most obvious or popular solutions on the market, identifying the pros and cons of each of these solutions. We will use our analysis to make a decision about our implementation in Polycode later on.

MongoDB

One of the most obvious solutions we have at our disposal, taking into account our current architecture, is using MongoDB. Indeed, all the data we want to index is stored within our MongoDB cluster, and querying the cluster when a user want to search for content seems logical.

MongoDB offers several tools for performing full text search, including the following:

- Text Indexes: MongoDB supports text indexes, which allow users to search for specific words or phrases within string content. Text indexes can be created on any field that contains string data, and support language-specific text search and stemming (the process of reducing words to their base form).
- $text Operator: The $text operator allows users to perform a text search on string content in the database. It can be used in a query to return documents that contain the specified search terms, and supports the use of logical operators (such as AND, OR, and NOT) to combine multiple search terms.
- $search Operator: The $search operator allows users to perform a full text search on string content in the database, using a search syntax similar to that of a search engine. It can be used in a query to return documents that match the specified search query, and supports the use of logical operators and parentheses to group search terms.

There are a few limitations to consider when using these tools for full text search in MongoDB:

- Text indexes can only be created on string fields, so they are not suitable for searching numerical or other non-string data.
- Text indexes do not support partial matching or wildcards, so they can only search for exact word or phrase matches.
- Text search is case-insensitive, so it will treat "apple" and "Apple" as the same word.
- Text search is limited to words and phrases that are at least three characters in length.

To use full-text search, you will also need to enable it and index the appropriate fields.

The performance of MongoDB's full text search feature will depend on several factors, including the size and complexity of the dataset being searched and the specific search terms and queries being used. In general, MongoDB's text search is designed to be fast and efficient, and is able to handle large volumes of data and concurrent search requests.

That being said, the performance of MongoDB's full text search may not be as fast as some other specialized search solutions, such as Elasticsearch or Apache Solr. These solutions are specifically designed for full text search and use advanced algorithms and techniques to optimize search

performance. They may be more suitable for very large datasets or applications with extremely high search volume.

It is worth noting that MongoDB's full text search is generally easier to set up and use than these specialized search solutions, and may be sufficient for many use cases. It is always a good idea to consider the specific requirements and performance needs of your application when deciding which search solution to use.

ElasticSearch

Elasticsearch is a powerful, open-source search and analytics engine that is widely used for full-text search, structured search, and analytics. Elasticsearch is commonly used for a variety of applications, including enterprise search, log analysis, website search, and real-time analytics. It can be integrated with other tools and systems, such as databases, data lakes, and application servers, to provide a powerful and flexible search and analytics platform.. We have already discussed a little bit about ElasticSearch, in the previous chapter about tracing and logging. We ended up using the ELK stack, which internally use ElasticSearch to store and search traces, logs and metrics.

Pros:

- Elasticsearch is highly scalable and can handle very large volumes of data and search requests.
- It offers a wide range of search and analytics capabilities, including full-text search, structured search, geospatial search, and aggregations.
- Elasticsearch has a flexible and powerful query language, which allows you to specify complex search criteria and fine-tune the ranking and scoring of the results.
- It supports distributed architecture, which allows you to scale out your search engine horizontally across multiple servers.
- Elasticsearch has a rich set of APIs and client libraries, which make it easy to integrate with your application.

Cons:

- Elasticsearch can be complex to set up and manage, especially for large or distributed deployments.
- It requires some knowledge of search concepts and terminology, such as indices, shards, and replicas, which may be unfamiliar to some users.
- Elasticsearch does not offer as many features and integrations as some other specialized search solutions, such as Solr.
- It may not be suitable for applications with very specific or custom search requirements, as it may not offer the necessary level of control or customization.

Like with MongoDB, we already have a ElasticSearch up and running in our infrastructure, for storing logs, traces and metrics. However, I would argue that it is not a good idea to use the same cluster for storing application data. Centralizing monitoring infrastructure and business logic infrastructure opens up a world of problems, where we are mixing up our concerns. We don't want

a bad actor or bad system provisioning from bringing down our monitoring system because users are overloading it, or vice-versa.

We would need to create a new ElasticSearch cluster, preferably deployed in another system than our current ElasticSearch cluster, within our Polycode application system.

However, the ElasticSearch ecosystem provides easy way to integrate with MongoDB using a connector such a MongoDB River, which allows you to index and search your MongoDB data in ElasticSearch. Separating your data store and your search engine give you more granularity and control about your resources, and help making sure that your data is always accessible, even if you can't browse it due to your ElasticSearch cluster being overloaded or in a failure state.

Overall, ElasticSearch is a great solution if you have a heavy dataset, and a high traffic of searches.

Apache Solr

Apache Solr is an open-source search platform that is built on top of the Apache Lucene library (just like ElasticSearch). It was developed by the Apache Software Foundation and is released under the Apache License. It was originally developed by CNET Networks in 2004 as an in-house search platform. It was later open-sourced and became a top-level project at the Apache Software Foundation in 2006. Since then, it has gained a large and active community of users and developers, who have contributed to the development of the software and provided support and resources for users. Solr is now widely used for a variety of search and analytics applications, and has been adopted by many major companies and organizations.

Solr is designed to be highly scalable and efficient, and is used for a wide range of search and analytics applications, including enterprise search, e-commerce search, and log analysis. It offers a wide range of search and analytics capabilities, such as full-text search, faceted search, geospatial search, and aggregations.

Solr is based on the Apache Lucene library, which provides the core search and indexing functionality. Solr adds additional features and functionality on top of Lucene, such as distributed search, a rich query language, and a REST-like API.

Pros:

- Solr is highly scalable and can handle very large volumes of data and search requests.
- It offers a wide range of search and analytics capabilities, including full-text search, structured search, geospatial search, and aggregations.
- Solr has a rich and powerful query language, which allows you to specify complex search criteria and fine-tune the ranking and scoring of the results.
- It supports distributed architecture, which allows you to scale out your search engine horizontally across multiple servers.
- Solr has a REST-like API and a wide range of client libraries, which make it easy to integrate with other systems and applications.

Cons:

- Solr can be complex to set up and manage, especially for large or distributed deployments.
- It requires some knowledge of search concepts and terminology, such as indices, shards, and replicas, which may be unfamiliar to some users.
- Solr may not be suitable for applications with very specific or custom search requirements, as it may not offer the necessary level of control or customization.

It can look very similar to ElasticSearch, and it is to an extent. Here are some key differences between Solr and Elasticsearch:

- Architecture: Solr is based on a traditional master-slave architecture, where a central server manages indexing and search requests are sent to a group of slave servers. Elasticsearch, on the other hand, uses a distributed architecture, where each server is a standalone node that can handle both indexing and search requests.
- Query language: Solr uses a rich and powerful query language called Lucene Query Syntax, which allows you to specify complex search criteria and fine-tune the ranking and scoring of the results. Elasticsearch has a more flexible and expressive query language called the Query DSL, which is based on JSON and allows you to create more sophisticated search queries.
- API: Solr has a REST-like API that allows you to interact with the search engine using HTTP requests. Elasticsearch has a more comprehensive API that includes both REST and native APIs, and also supports real-time search and analytics.
- Ecosystem: Solr has a smaller and more specialized ecosystem of tools and integrations compared to Elasticsearch. Elasticsearch has a larger and more diverse ecosystem, and is supported by a wider range of companies and organizations.

In terms of popularity and adoption, Elasticsearch is currently more widely used than Solr and may be the default choice for many development teams. However, Solr is also a well-established and widely used search platform, and may be a good choice for certain use cases or applications.

Polycode

Now that we have explored the available solutions on the market, we will now focus on Polycode and how we can integrate a search engine in the application. I will begin by showing how the user should interact with the search engine on the web page, then talk about the solutions that I've retained, and we will finish by looking at some sequence diagrams that shows the process of indexing and searching.

UI Mockups

I have defined simple UI mockups for the search bar and search functionalities. Here are the mockups from the search bar when it is closed and when the user make a search :

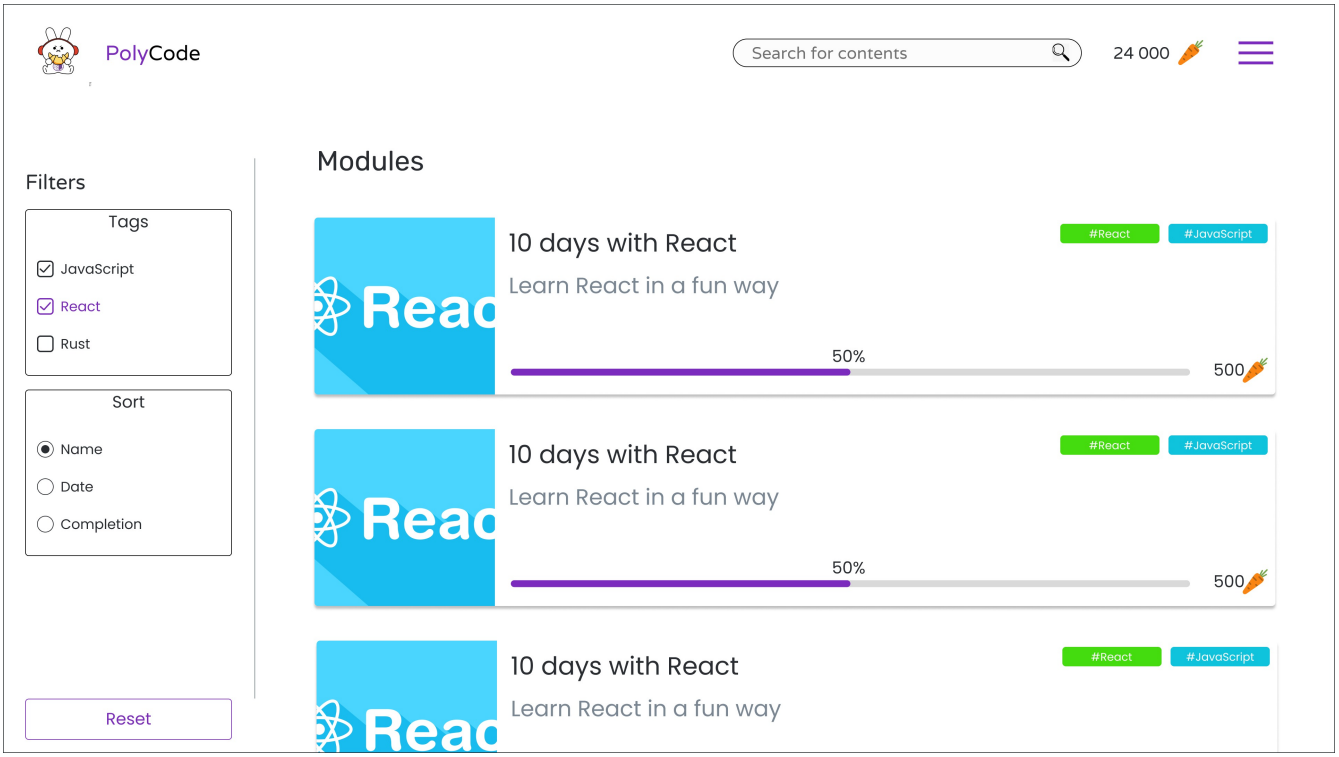


Figure 26. UI Mockup: search bar closed

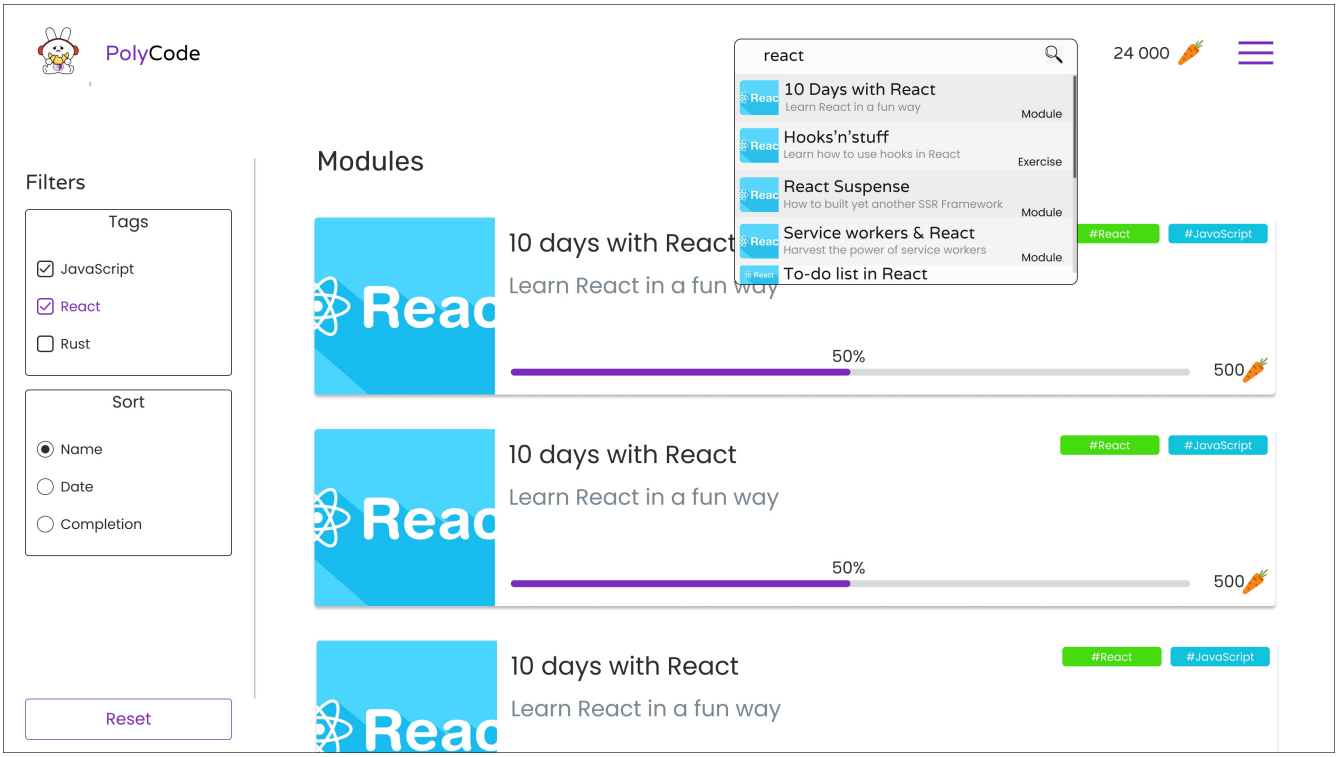


Figure 27. UI Mockup: search bar opened

The search box is located in the navigation bar, always shown to the user. This makes it easily accessible and allows for a faster browsing of the site. A magnifying glass icon is situated to the right to hint the user into figuring out this is a search bar, supported by the "Search for contents" placeholder text.

When clicked on, the user can type in keywords of what he is looking for. In order to not overload the search engine, we need to debounce and add delay to the search requests. The exact delay should be shorter than 1 second if possible, depending on the hit on the system performance it has. There is no "Search" button, but the user can press enter to send a request immediately.

As results, we include the title, description and image of the resource found, but also its type. This is a simple search box where the user can only type in text. It is very simple by design, I think we don't want to overload the search box with additional filters and the simple query system should be sufficient as is. I've not designed a search page, because I don't think it is relevant, and the search box should be enough. We can however create a page responsible of making advanced search for the user later on if we identify the need after user feedback and evolution of the system, without modifying the underlying search engine.

The stack

Now that we have defined what the user should expect, let's talk about how we implement it. I think it makes more sense to use MongoDB full search text feature. Here are the key points that tipped me in favor of this solution:

- We are dealing with a relatively small order of magnitude of data here. Even if all contents on the platform might sounds like much, it is still in the range of what we can expect MongoDB to handle pretty easily, given the right indexation. We should not be searching over the 1000-10,000 contents range, which I think is when we will start to hit the limit of MongoDB. The text

to search through is pretty heavy though, which might have a bigger impact that what I'm expecting. Actually implementing the solution and testing it to production scale will be necessary, to make sure that the current user load and content offering is not too much for MongoDB, but also to have a better understanding of where the limit of this system is, in order to take action before this limit is reached.

- The implementation is really straight-forward. Enable full-text search in MongoDB, index the fields you want to be able to search on, and add a few routes that make use of this capability in your microservices.

If I was to upgrade to a more robust solution, I would use ElasticSearch over Solr, mainly for its wider ecosystem, growing popularity, support and native SDKs for multiple languages and easy integration with MongoDB.

MongoDB allows for specifying weights on each fields that you index. It is useful in our use case, as matching the requested word have more significance when looking at the title of a content, followed by its description, and the the content itself.

This approach, however, locks us into searching only data stored in MongoDB, and if we wanted to search through data in our Postgres, we would need to add some other Postgres specific code. This is not the case as of right now, and I don't see a use case where we would need this in the near future.

On the application side, I would advise creating a common library for all of your microservices, that abstract the engine-specific operation you might want to do. This is to make sure that we have a clear way of moving around with search engines, since we have identified some limits of our current system that might be reached sooner than expected. Factorizing repeated and technical code is usually a good idea anyway, just like abstracting your code. This also makes debugging easier, since we have only one piece of code that is responsible for interacting with your search engine everywhere in your codebase. If you use different programming language, different framework or have different enough constraint between your microservices, this approach might not be possible.

Sequence diagrams

To better illustrate how this system works, we will look at two sequence diagrams: one for the indexation process, and on for a research request. Here's the indexation sequence diagram :

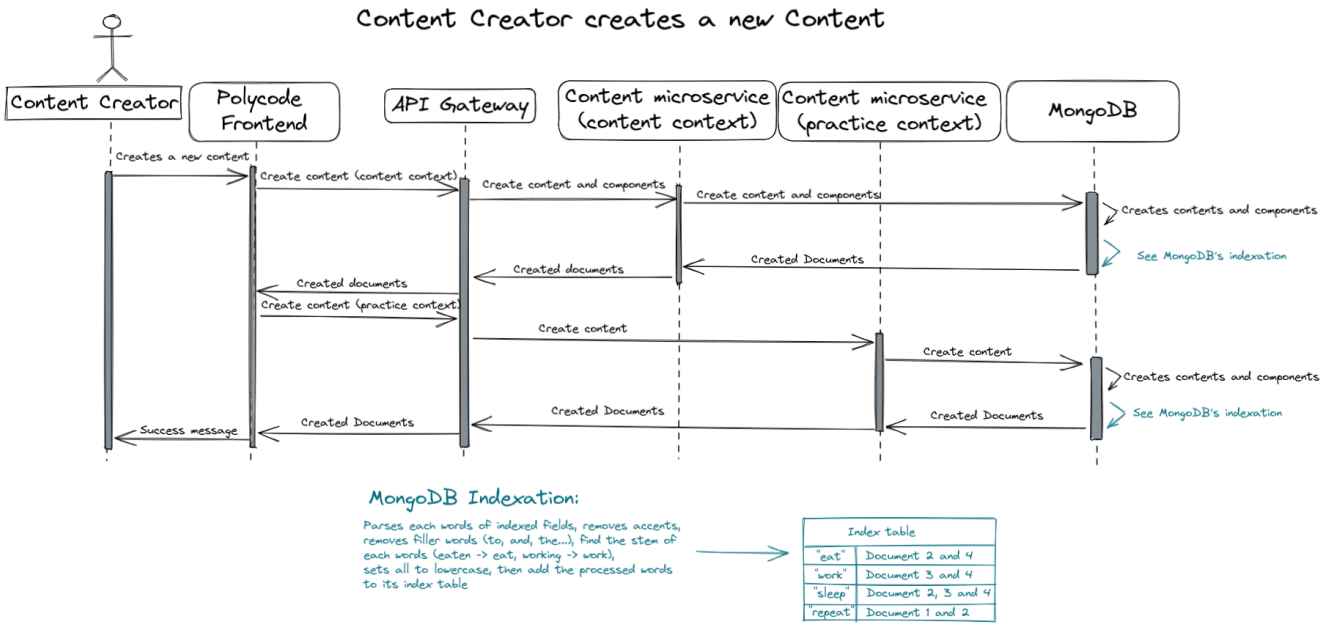


Figure 28. Indexation sequence diagram

As you can see, I've taken the creation of a new content by a content creator as our use case, since this is where our indexation will take place. This is where the content we want to index is created, so this is also where we index it. This is a two step process due to our architecture:

- First, the content needs to be created in our content context
- Then the content can be created in our practice context

If you don't understand why it is like this, please refer to the first chapter of this paper, where we define our domains and contexts.

However, this creates some complexity, since some of our indexed fields are in different contexts. This is not a big deal, and just adds a totally manageable programmatic overhead in our system. At the bottom of this diagram, you can see explained the process of indexation with MongoDB. This is a pretty heavy process, and adds a lot of overhead when creating contents. This is totally fine and acceptable, since there is a low volume of content being created, and it will continue to be like that even if the application grow. This operation also needs to be done when contents are updated, and this process is automated with MongoDB indexes. This does not impact read performances.

Let's now look at the sequence diagram when the user search for contents:

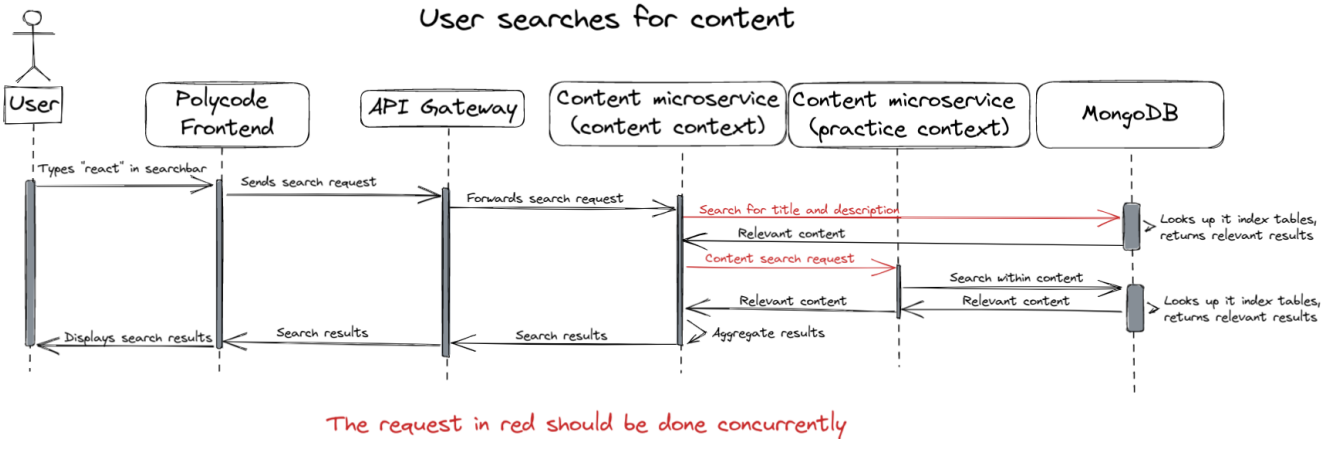


Figure 29. Research sequence diagram

This looks like the indexation sequence diagram, with the main difference being that it is the content microservice, in the content context, responsible for asking the content service in the content context. This is done because we need to aggregates the results of both research, and to link content from the practice context with the results of the content context. This is logic related to the practice context. We end up with two requests to our MongoDB cluster, one for searching its index in the title and description, and one for the data inside the content. The data returned to the frontend should be formatted in a way that no further processing is needed to be done by the frontend. Abstracting away how our search engine works from the frontend is important. This allows us to play and tweaks with the configuration of our search engine without needing to change our frontend everytime. The stack used in the backend should not matter to the frontend.

Conclusion

I think we should use MongoDB capabilities of full-text search and indexation, as it is adapted to the volume of data and the traffic we have. We should implement this in a way that allows us to easily migrate towards another solution, since MongoDB has limits that might be reached. However, this approach does not require a lot of work, since we are working with a stack that is already deployed and available to us.

Runner architecture

Introduction

Throughout this paper, I've glossed over a very important part of Polycode: the runners. I've told you multiple times that I wasn't diving in deeper the subject because there was going to be a section dedicated to it; here we are. In this section, I'm going to talk about what are runners, what are the constraints we need to respect, how can we create a system that scales in a secure and efficient manner and we will finish, as always, by taking a look at some diagrams to better understand the decisions I've made for Polycode.

Runners are the backbone of our application. This is what drives the interactivity and the engagement, since it is at the core of every piece of code that is executed on the website. For the user, it looks seamless, but there is a great deal of complexity and thoughts that must be put into it.

What are runners

We need to define what runners are and what they aren't. As you might have figured, runner run code. More specifically, they run the code fed by the user when they resolve an exercise. The code the user wrote is then being fed by the input of each validators that might exist for the exercise, and the standard output and standard error streams are returned.

There are a few key points that needs to be made clear. First off, runners are more or less agnostic from the underlying (or overlying, depending on how you see it) application. It does not now what validators are, or what contents are, they just take pieces of code and run it against some input. It could very well be used for some online Rust playground with some very minor tweaks. Although we must be conscious about the Polycode's requirements and constraints, we are far enough from the core business logic that most of the decisions will be made on a technical and human standpoint.

Furthermore, since the runners execute user input that can't be trusted, since anybody can use our platform, we need some strong security and isolation mechanism in place to make sure that:

- A user can't disrupt the stability of the platform
- A bad actor can't escape the isolation and access the system

As of right now, we need our runners to be able to run code written in Javascript, Rust, Java and Python, with project composed of one or multiple files. This means that our runner architecture needs to be able to compile Rust code, and our actual runtime needs to have the appropriate tools and interpreters to execute the code that is being given to them.

Program input is a string, that can include newlines. This is a pretty simple input system, and there is no way to influence the input based on the output of the running program. This means validators must be deterministic.

Running code in an isolated manner

As I've mentioned in the introduction, one of the key point and constraint we need to respect is to have the user code run in a totally isolated manner. It should not be able to interact with the host system. A great way to think about runners is a serverless platform that you can find in most public cloud nowadays, like AWS's lambda. You give AWS a piece of code, that will be executed somewhere in their cloud. This is the same principle that we have with Polycode: the user give us a piece of code and they want it executed somewhere in our system. Just like AWS and its customers, we don't want the code to be able to escape the boundaries we have given them. In this chapter, we will look at different ways we can achieve this goal, the pros and cons of each of them and what it implies.

Containers

The most basic and easy way to isolate programs, is to run them within a container. A container is nothing more than a process running on the host system, that has been configured in a way to typically run in its own virtual filesystem, with limited network connectivity, no knowledge of other running programs, limited in resources and isolated from the host machine. They allows running multiple isolated systems on a single host, even if those system tries to access the same resources, the same port for example.

This looks like everything we need, with no major drawback ! Right ? Not so easy unfortunately. The isolation is done at the operating system level, opening a great and somewhat high level surface of attack. Escaping from container isolation is something that can be done pretty easily, if the container configuration is faulty. This can be especially dangerous if the container is running as the root user, as the attacker would then have full access to the host system. Just like with any software, containers can contain vulnerabilities that need to be patched. If these patches are not applied in a timely manner, the container could be at risk. This is not a huge deal in our case, since the container lifespan is the one of the program of the user, which is pretty short. Moreover, the user could only compromise its own container, as long as they are not able to escape it.

Because containers share the host's kernel, a containerized application that is able to compromise the kernel could potentially gain access to other containers on the same host, as well as the host itself. This could allow an attacker to gain access to sensitive data or to compromise other systems on the network. This is exactly what we want to avoid, and it looks like running unsafe code in a container will not give us sufficient isolation to bring the risk down to comfortable levels.

Moreover, managing resource of a container can be tricky. The main and only available solution out there is to kill a container that is grabbing more memory or CPU resources than it should. Ideally, we would want a system that is actually able to cap the usage of a program, making sure that it can't grab more resources than it should in the first place.

One last concern to have about containers in our system, is that we are probably going to run them inside an already dockerized environment. This means we need to run our runtime in a Docker in Docker (DinD) setup. DinD can introduce additional security risks. Because the Docker daemon inside the container has access to the host system's Docker daemon, any vulnerabilities in the container could potentially be exploited to compromise the host system. This is especially concerning if the container is running with elevated privileges, such as the root user. Another drawback is that DinD can have performance overhead compared to running the Docker daemon

directly on the host system. Because the Docker daemon inside the container needs to communicate with the Docker daemon on the host system, there is additional overhead in the form of network communication and process management. This results in reduced resource efficiency.

Virtual Machines

Another big tool for creating isolated environments at our disposal is virtual machines (VMs). Today, the world runs on VMs. Every cloud providers, private or public, relies heavily on VMs to divide a bare-metal server resources into smaller, individual and isolated machines.

VMs are using modern processors virtualization capabilities, this means our workload are isolated thanks to hardware implementation, whatever the host and guests operating systems. This allows for a much tighter isolation, and security. There is a big notion in the previous statement: the guest runs its own operating system. This means that there is no collision at the kernel level that can occurs, since we would spin up a virtual machine for each of the user's request. It can compromise the whole guest system, and still be isolated, thanks once again to hardware virtualization. Escaping from hardware virtualization is significantly harder, at level that I think are comfortable.

VMs also allows us to better isolate resources, since we can configure the available resources in each VMs. Unlike with containers, the guest operating system will be responsible for managing those resources, and we have a clear boundary on them. It can't use more than what we gave it. However, operating systems typically have system in place to make sure to continue functioning properly even in the case where it needs resources it doesn't have. For example, if you need more RAM, your operating system usually have swap pages that it can use as if it was RAM, at a very significant performance cost.

This also means that we need that the overall resources usage will go up, since every code execution also means spinning up an operating system. However, Linux has some great option, providing kernels that not only takes fraction of seconds before becoming operational, but also very little resources. Linux images can be customized to our needs, tuning them to our needs, to further decrease resource usage and boot time. We can also imagine having special VMs that run special operating systems, such as Windows. This would allow for executing code that is targeted at win32 for example. This also opens up the possibility of running and executing code compiled for different processors architecture. This would allow to create content on RISC-V assembly for example, even if those VMs would be extremely slow, since we are not virtualizing anymore, but emulating. It also means we need to use a virtual machine manager capable of emulating such system. But we open a native experience for the user.

However, there is also a increased complexity with VMs: talking to them is not as easy as with processes. With containers, you could programmatically run commands easily, with solutions like Docker for example. With VMs, you either need to have a custom kernel that is built to run some commands at startup, and find a way to extract executions results out of them, or you need to find a way to communicate with an agent in the VM (still meaning a custom linux image), that will respond to commands from the host.

All of this complexity might be a blessing in disguise: this opens up a world of possibilities in how we run our runners. Having an agent system that takes in request allows for a lot of features and optimization: preemptive startup (starting VMs before actually needing them, with them waiting for requests), fine-tuning execution parameters on the fly, or even executing custom operations

depending on the code being executed, that can't be known ahead of time.

Overall, virtual machines gives us a satisfactory level of isolation, at the cost of increased resource usage and complexity. However, I would argue that the tradeoff is worth it, and it goes beyond that: we must accept every challenges to make sure we have good isolation and a secure system.

Serverless

As I've eluded earlier, what we are trying to accomplish is very similar to what cloud providers have been doing for years now: taking some small pieces of code, execute it and exit. Why not base ourselves on the work these companies has already done, instead of reinventing the wheel ? Let's look at the pros and cons of using serverless services cloud providers offer.

First, let's define what serverless actually is. Serverless is a cloud computing execution model in which the cloud provider dynamically manages the allocation of resources and charges for the execution of code. With serverless, you can run your code without having to worry about the underlying infrastructure, such as servers or virtual machines.

In a serverless model, you write and deploy your code in the form of functions, and the cloud provider executes the code in response to triggers, such as HTTP requests or events. The cloud provider automatically allocates the necessary resources to execute the code, and you are only charged for the actual execution time and the number of requests or events processed.

This fits particularly well what we are trying to achieve. However, there is an additional variable in all of this: cost. Indeed, until now, we didn't worry about what the cost of our solutions were, since we are currently using machines that are at our disposal. With these serverless services, you have to pay for the execution time of your code.

Another thing to demystified, is that it's not "send the code of the user to the serverless service and everything happens magically". We need to write a piece of code that takes a user input as a requests, figure out the inputs the program should be running, then start a runtime environment to run that code, while compiling any code before running, if applicable. We also need to need to properly isolate the code, and manage permissions of our serverless application properly, to make sure that the user doesn't have access to any other resources that is at disposal in our cloud platform account.

Serverless platform usually runs in a pretty trimmed-down system, and you have not a lot of guarantees and possibilities about the system that you're running on. This means you might not have the needed dependencies. Serverless was made for serving HTTP requests, opening some sockets to some databases or other HTTP connection along the way. Not for spawning another program. Serverless is more complex to design and implement than traditional architectures, because you need to consider the specific constraints and limitations of the serverless model. In addition, the additional abstraction introduced by the serverless model can make it more difficult to debug and troubleshoot issues, because you don't have direct access to the underlying infrastructure.

If we forget the fact that we have a base infrastructure already at our disposal, and that we don't currently have to pay for anything, cost become an interesting factor that might justify putting the work into making serverless work.

With serverless, you pay for execution time. This means that if you have a low volume of user using your service, you might need to execute code during 1 hour in the day. With serverless, you would pay the rate for code execution of 1 hour. If you have a server idling, waiting to spawn VMs, this server will run during 24 hours, and you will be charged for this period. Even if you can get much more with your money with VMs, if you don't have the usage to justify it, it is not cost-effective, and you might rather take the decision to use serverless services. Moreover, serverless is scalable basically infinitely, since you rely on your cloud providers resources. This comes with other benefits cloud providers typically gives you, such as a near-always uptime, hardware, hypervisors and operating system maintenance for this kind of highly-abstracted services, flexibility, global availability, etc..

The right balance, firmly on a cost of operation effectiveness standpoint is most likely somewhere in the middle. We will talk about it more later in this section.

Running code in a scalable way

One of the other challenge of our architecture, albeit for the runner of for the entire system, is to create a system that can scale seamlessly, whatever the count of users are on our platform. In this chapter, I want to take a look at how we can accomplish that, from a system architecture standpoint mostly, with some discussions about technological choices. We need to be confident in our runners, and their ability to handle all the work we throw at them.

We want to be able to scale horizontally, meaning we want to be able to spawn new runners into the system when the traffic surges. This also means we need to have a way to control this behavior, as well as having an entry point to this sub-system of our architecture. Concretely, we need a service that takes user code and input to be run against, schedules a machine to run the user's code, with the input, get the results once the machine has finished executing, returning it to the caller.

There are a lot of implications in this architecture, let's try to tackle them one by one.

Controller / worker architecture

What I've just described looks awfully similar to a controller worker architecture, where we have our scheduling/supervising microservice (let's call it the controller from now on), that gives directives to worker machines, about what to execute and how to execute it. However, there is a significant caveat that is usually not found in this type of architecture: we can have (and will have) multiple controllers. Controllers also needs to be scalable, we can't trust the resiliency and scalability of an architecture with a single point of failure. This means we need to coordinate somewhere, since we don't want our controllers giving two directives at the same time.

This would cause, depending on the implementation:

- Risking exceeding the worker machine resources, if we spawn one too many workload due to improper communication between controllers and the worker machine
- Having a failed execution request to one of the controller, if the worker simply refuse to run one of the requests

In either way, this is some unwanted behavior. How can we resolve that ? There is probably a piece of a solution that can be found with message queues, where the controller would send their requests to a message queue, and available runners would pick them up as long as they have available resources. This would mean that it is the runner's job to manage its resource, which totally makes sense, since he is the one that knows its available resources in real time. This creates a new problem though: how to retrieve the execution results ? Another message queue ? But since the request to the controller was made synchronously, we need the correct controller to pick back the answer. And in a timely manner.

All of this adds a lot of complexity, even if we would end up with a more robust system. For now, I would argue that the best solution is to simply have no synchronization, but make sure that the workers rejects one of the controllers request if it received two requests at the same time. The controller needs to have logic to handle this possibility, and reschedule a new runner immediately.

At our scale, and up until a hundred concurrent user on the platform, we probably won't even run

in this scenario. I think this system would work well enough to handle thousands of concurrent user pretty easily. Let's not forget that most runners most likely will have enough resources to run multiple machines at the same time, lowering the occurrences of this problem even more.

But is there a simpler solution ? Yes, and much more elegant. Instead of looking at the problem as worker picking up work when they can, what about simply coordinating controllers ? We have state that must be shared across all instances of our controllers, the overall status of the runner infrastructure. Let's use this state in a way that disallows having inconsistent data. I think a great candidate for the job is Redis. It is fast, we don't have concurrent access concerns since it is one threaded, we don't care if we lose data if it shuts down, since we have lost our infrastructure anyway and need to reset everything back to zero. The data we need to store can easily be stored with Redis' datatypes. We do add another piece of software in our system, but I think this is justified.

So to sum up this architecture: we have a controller microservice, that is responsible of receiving execution requests. It then looks at its available pool of machine, and choose one that is available. It forwards the request, and the worker execute the code, returning its output. The controller then returns the output back to the request emitter.

Machine pool

Another implication of this architecture, that I've touched just before, is the fact that we need to manage a pool of workers. These workers can spawn machines that executes the user code with some specified input. It means we need a way to control this pool, manage its state, adding new workers when needed, removing others when we don't need them, and detecting faulty ones.

We want to be able to dynamically register new machines to our system, without disrupting the service. This can be done pretty easily, since we are using Redis as a shared state. All we need to do is to hit up a controller with a "register new worker" request, and it adds it to the pool. However, we want to have some security mechanism in place. We want to add only trusted workers into our system. A basic authentication challenge should be enough, such as a certificate or a key.

Another key point, is to detect faulty workers. A worker can be faulty in a myriad of ways. For example, it might appears available, but there is a power outage, a network disruption or other external factors that made the worker not operational. If we don't have a way to detect these cases, we will end up with dead workers. There is also the possibility that the worker is available, that it can be contacted but for some reasons, all the requests ends up with errors. Something might be wrong on the machine.

I want to discuss how we can prevent and remediate this kind of failures. In the case of the whole worker machine going down, or being available, I think a good solution is to periodically send liveness requests to this worker. This would mean that, every x seconds, a simple request is send, with the expectation of having a response. If we have no response, this means that the worker is not operational anymore, and should be removed from the worker pool. This is similar to the heartbeat pattern. This would ensure we have a clean pool of machine to work with, at a low cost, since sending such simple requests is not resource-heavy at all, and even if we have multiple thousands workers, if we send a heartbeat request every 30 seconds, over multiple controller replicas, it computes down to a negligible number of requests for modern systems. As for the implementation, I would make use of Redis' sorted set datatype, using the elapsed time since last

heartbeat as our scoring system. With that, controllers could periodically check for the workers that are over this threshold quickly, and then send our heartbeat packets.

This, however, does not resolve the problem of faulty worker that fails to run workload that we throw at them. Resolving that is not as trivial, but the use of the circuit breaker pattern might be a great way to start tackling this problem. When an abnormal numbers of errors are detected on one runner, we would mark this worker as faulty, effectively disabling it for x amount of time. After this time has passed, we try to slowly send back requests to this worker again, and if it is still malfunctioning, we tripped the circuit again, for a longer period this time. This should be synced with other controllers, via redis once again.

There should be a strong monitoring and alerting system in place when a circuit breaker trips, so that system administrators are notified something's wrong with one of the worker, and to let them investigate if it is a false positive, or if there is actually a problem.

Circuit breakers are a great tool, but it might add a lot of work to implement it. I don't think it is a good idea to implement it right now in Polycode. However, this option should be kept in mind if we encounter this kind of problems as the application grows.

Scaling infinitely

One major problem with our infrastructure as it is, is that we are limited by the number of workers that are available. If the traffic was to exceed our workers' capacities, it would mean that our user would not be able to run code in quickly, or even at all. To circumvent this problem, we need a system in place that automatically finds new way and resources to run user's code.

Thankfully, cloud providers gives us a way to get access to machines quickly. We can use this strength at our advantage. I see two main way of using cloud providers:

- We have a system in place that creates and destroys workers on the cloud depending on the traffic
- We use the serverless offerings of those cloud providers

Both approaches have pros and cons, but let's talk about the similarity of both approaches. As we have discussed before, renting machines or execution time on the cloud is costly. We need to prioritize our own infra as much as possible. We have bare-metal servers available to us, we need to use their resource to the fullest before turning towards cloud-providers. This means we always make profitable our hardware investments, which is already paid for, before adding exploitation charges, in the form of services rental in the cloud. However, to handle peak traffic, it might not make much sense to invest in more on-premise servers, as it would be idling most of the time. This is where the usage of the cloud really shines in Polycode's case.

Let's talk about renting a whole machine on the cloud. The biggest issue we have here, is that we can't use VM-based isolation, since nested virtualization is not feasible on most cloud providers. We would need to rent bare-metal machines, which is definitely a doable and valid solution (c5.metal x86_64 machines go for ~4$/hour on AWS, pretty expensive, but can run a lot of workload concurrently). We can also go the docker route. However, as discussed previously, a dockerized environment is perhaps not safe enough for running untrusted code.

The other option is to run code via serverless services of cloud providers. As we have discussed previously, running via serverless functions can be somewhat tricky. Running the code itself is not very complicated, it's isolating it that is harder. But as mentioned, previously, do we really need isolation in a serverless environment ? As long as the serverless instance doesn't have any access to our AWS account (in the case of AWS), we should be safe enough. I think exploring the serverless solution is the best of the two.

So to sum up, we would use our on-premise infrastructure to its fullest. If traffic surges above our capacities, we offload some of the work to serverless services in the cloud.

Caching

Another technique commonly used to improve response time and reduce system load is caching. It is most effective when accessing a rarely mutating resource, with high volumes of read and the data resides in a foreign data store or involves heavy computation. At first glance, it is not really applicable to our use case here. However, I would argue we can try to find some way to cache requests, since the cost of running the user's code is very high, and also time consuming. I don't expect to catch a lot of similar requests between users, however, there are edge cases where we will significantly reduce the load on workers, such as users trying to run the base code that is given to them, or them rerunning their code out of frustration, for example. For simple exercises, having a cache system makes even more sense, since we can expect some users to come up with the exact same code. Our goal is to cache code execution results for a specific combination of user input, validator and content.

Since we already have a Redis cluster at our disposal, why not use it as our data store for our caching ? Redis is often used for caching as it is fast, and provides efficient way to store simple data, which is exactly what we need. Our cache would be shared between all our controllers, which is what we want.

I propose the following as a first draft as how to cache data:

We take the user input, stripping it of all newlines and spaces. We then hash it, using a fast algorithm such as md5. This will be used in our key in Redis. I suggest a key in this format : `{contentId}:{validatorId}:{language}:{hashedInput}`.

This way, we have a simple a deterministic way to lookup our cache. If there is a cache miss, we send the execution request to the runner. We get back the outputs, add them to the key we defined above.

There are different caveats that we need to think about and avoid. Firstly, the results might vary execution to execution, even if it is the same code. For example, if the user uses randoms or time in his code, every execution will yield different results. However, since this kind of behavior is not something validators can predict, it means that no validator will rely on this type of code. I think it is reasonable to ignore this edge case, since the user will never be pushed towards using those variable functions, and if they do, it will have minor impact on their experience.

Another caveat to consider, is when validators are updated. Indeed, as of right now, we cache via the validator's ID, and not its input. This means that someone can update a validator, updating its input, but since the ID is the same, we don't see the change and our cache is invalid. This would

mean that we would continue to return old values. I see three main way to solve that:

- Setting a low Time-To-Leave for our cache inputs. This is the easiest, but it means our cache becomes weaker, and we still have the problem, just for not too long.
- Notifying the controller, so that it invalidates the cache. This has immediate effect and we can also cleanup the old cache that we don't need anymore, but we need to add additional routes and communication within our content context.
- Caching using the inputs in the key instead of the validator. This has immediate effect, require no additional routes or code. But we aren't notified when a validator has changed, meaning that we don't have a way to cleanup the cache.

I would argue that the last option, caching using inputs in the key, is our best solution here. It requires no effort and is very effective. We do need to have a way to cleanup data though, but I don't have enough knowledge and experience to have an idea at how fast we would fill up the Redis cluster. I would test it in some production environment, to have a better understanding about how it behaves over time and make a decision based on that. Our key format would now be `{contentId}:{hashedValidatorInput}:{language}:{hashedInput}`. This option also has the added benefits of totally decoupling the notion of validators from the runner. We just take some input as parameters, we don't have to know from which validator it is from. This is probably what we want, and I would argue that the job of getting the validator's input should be done prior reaching the runner system.

Caching is something that should not be prioritized too much. As of right now, having a functional runner system is the priority. We don't have enough user to justify putting this much effort in a caching system.

Heating up VMs

The last point I would like to touch on in this chapter, is the process of heating up VMs. What I mean by heating up VMs, is starting execution VMs before needed, in order to reduce the response time by moving the VMs startup time before a request comes in. This will improve the user experience, by reducing the processing time significantly, but also improve our capacity, since we make some work done during off-time. This means we are able to better handle surges in traffic.

Workers could be configured to heat up some of their VMs. When they are ready, they would notify controller. When controllers receives runner requests, they would prioritize any already heat up VM for the execution. This also means that it is an implicit way to prioritize workers, since we can configure some that have limited resources or that we want to use only when absolutely needed to not heat up VMs.

Once again, this is not an absolute priority. This feature should be added incrementally, whenever we have time and resources to spare, just like caching requests.

The complete picture

For the runner architecture, I suggest using a controller/worker model, where workers are effectively in a pool of machines that controllers can choose to use for running workloads. This

model allows us to scale efficiently, with a system where we prioritize workers where there is heated up VMs, then workers where there is none and finally a system that sends the code on the cloud in a serverless environment, to be executed. To have a better-optimized and sustainable system, we would cache every requests and their output, and then try to read the cache to not execute code that has already been executed on subsequent request.

Polycode

Now that we have defined what are the constraints for our runner architecture, let's take a broader view and take a step back to completely integrate our architecture into Polycode. In this chapter, I will draw some diagrams showing how the architecture operates on a broader scale.

Architecture diagram

To start off, let's take a look at our complete architecture :

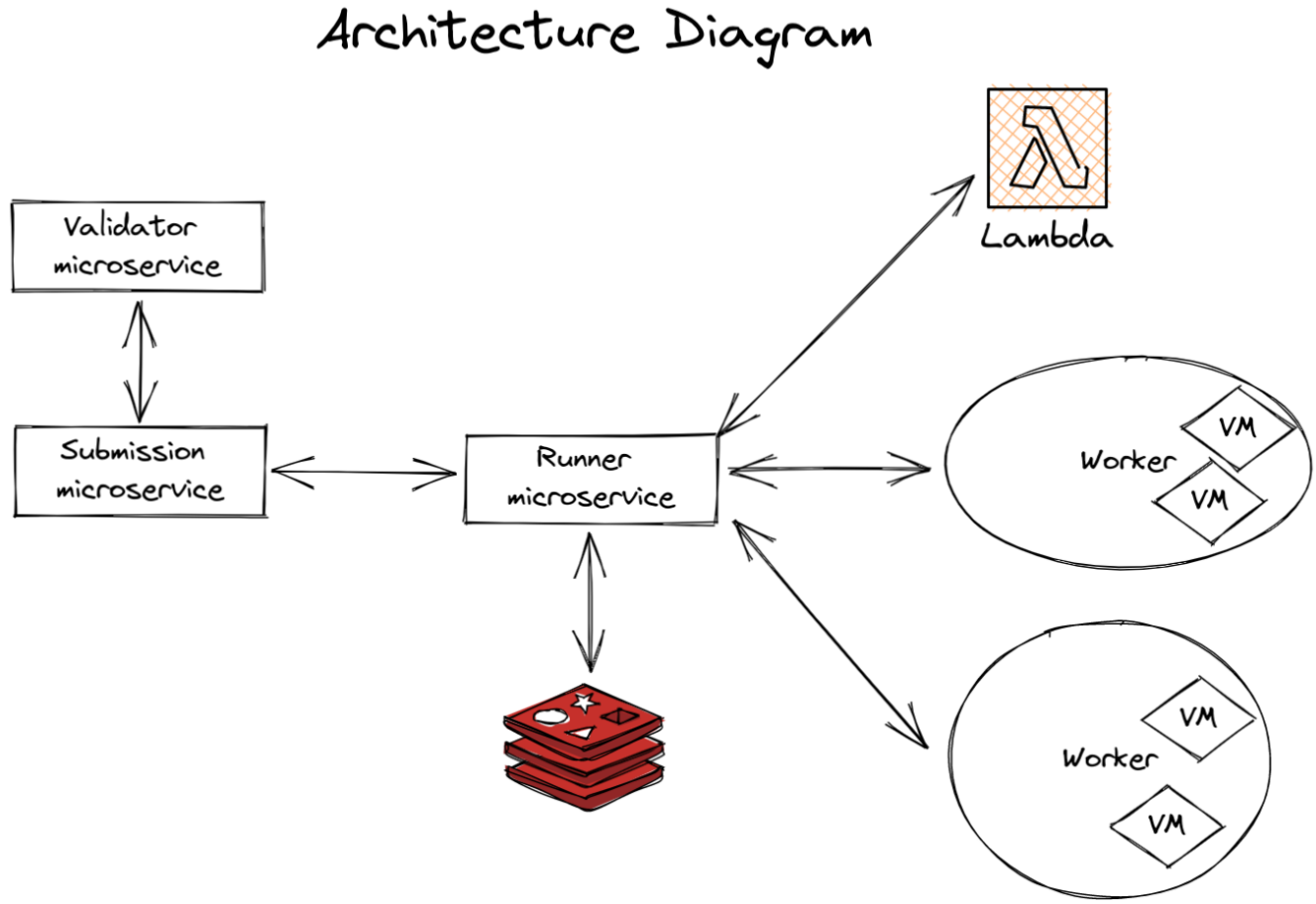


Figure 30. Runner architecture diagram

This is the complete diagram, with all the suggested features from above. I've chose AWS as our cloud provider, but this is not a definitive choice. It is here for example purposes, but we can chose any cloud providers that offers serverless services. I've made the decision that fetching the validators input should be done within the submission microservice. In fact, with this diagram, the question of "Should the runner have been its own bounded context ?" arise.

Looking at the diagram, it also becomes apparent that the runner microservice is clearly a controller (or a supervisor), talking to its worker to execute the workloads. We can see that the worker spawns VMs, but we also see that the runner microservice communicates with Lambda if the on-premise workers becomes overloaded.

Registering workers, sequence diagram

One thing that doesn't show in the architecture diagram is that workers can be added and removed dynamically from the system. To better illustrate that, I came up with a sequence diagram, of a new worker registering into the system:

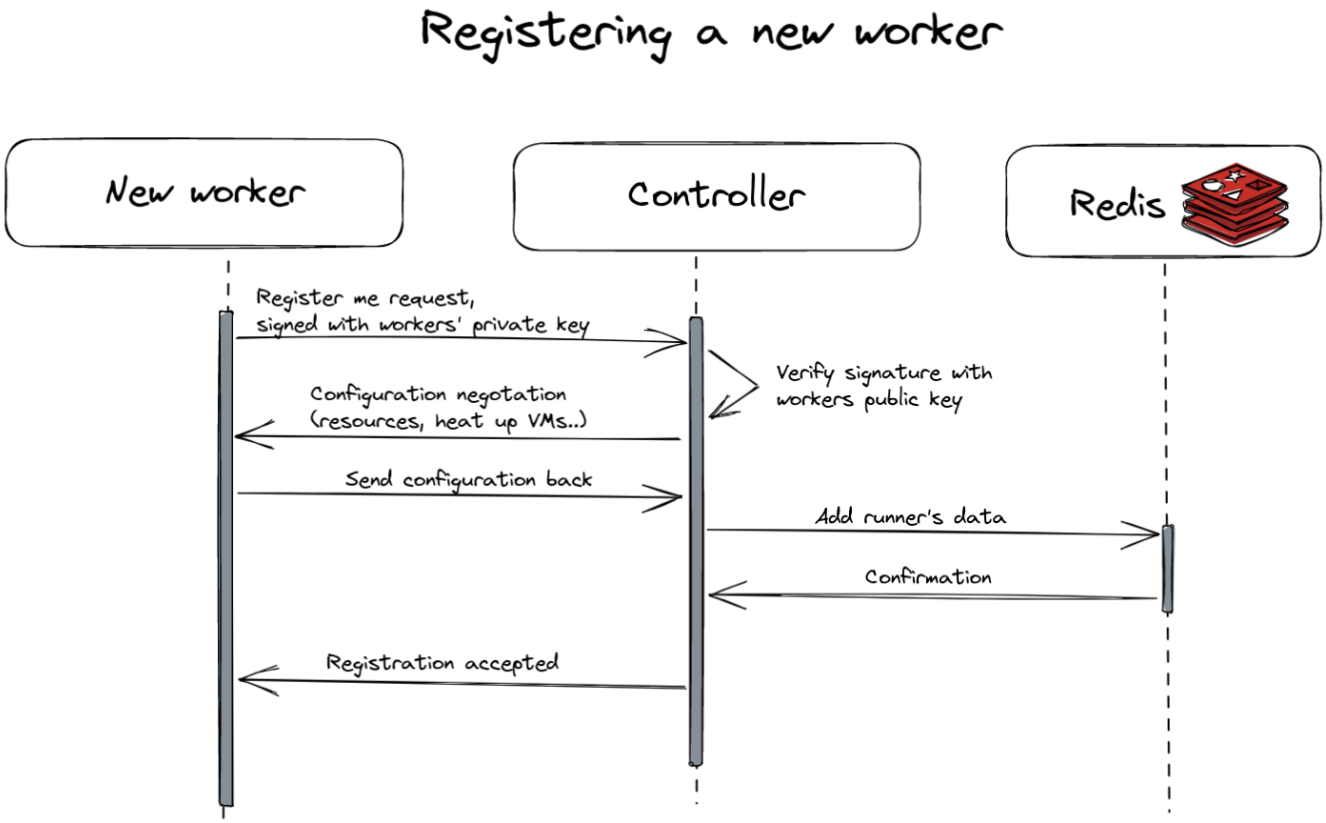


Figure 31. Registering a new worker, sequence diagram

They are a few key points to extract from this diagram, let's talk about them one by one.

Firstly, we can see that we have some kind of encryption at play when making a register request. As mentioned previously, we don't want anybody who finds out the IP address of our controller to register their worker, as this would open security risks, and it would mean that the bad actor could forge false responses to the code ran. To protect against that, I suggest using simple asymmetric keys. Workers would need this key to sign their request, and the controller would use the public key to verify the signature. This way, unless the key leaks, we can be sure that the worker is coming from an authorized party. The underlying protocol should be encrypted as well, with something that prevents replay attack. If the request is sent in clear-text, an attacker could grab the request and do a replay attack. Same thing if the underlying protocol doesn't prevent replay attacks. We can also imagine that the controller sends a nonce when sending the configuration negotiation, and the worker sending it back, signing the response to prevent such attacks.

Secondly, it is the controller that requests configuration to the worker, not the worker sending it to the controller when trying to register. This is done in order to make sure that the controller can get all the information it needs, and reject workers that do not comply with either the configuration structure or the configuration itself. This is also where the controller should check the nonce and signature that we just talked about.

Once the negotiation is finished, the controller adds the corresponding information in Redis,

effectively marking it as ready. If everything's good, the registration is finished and accepted.

Conclusion

The runner architecture, as you just saw, is a topic on its own. It needs thinking to come up with solution that are secure, in its way it runs code, but also in the way the system interacts, as well as being able to scale properly, whatever the traffic is. I suggest using a mix of a controller/worker architecture, on-premise workers and public cloud's serverless features to scale properly. Our on-premise workers should create VMs to ensure the execution is isolated to the best we can. To manage state, I would use Redis as a data store for the controllers. This Redis cluster can also be used for caching, to deduplicate requests and make sure that we keep the number of costly execution as low as possible, hand-in-hand with prioritizing on-premise workers before using serverless.

Data architecture

Introduction

One of the key components of a successful microservices implementation is a well-designed data architecture. This section of this paper will provide an in-depth examination of the various data architecture considerations that must be taken into account when designing and implementing microservices. We will begin by discussing the importance of data isolation and data consistency in a microservices environment. We will then explore different data storage options, such as relational databases, NoSQL databases, and in-memory databases, and their suitability for different types of microservices. We will also delve into the use of data caches, data replication, and data sharding to improve performance and availability. Finally, we will discuss how we can put into practice the theory we talked about by making a target architecture for our Polycode application. By the end of this section, readers should have a good understanding of the key data architecture considerations that must be taken into account when designing and implementing microservices, and be equipped with the knowledge to make informed decisions about data architecture in their own microservices projects, as well as a better understanding of my suggestions for Polycode.

Data architecture in microservices

They are some important concepts to understand when talking about data within a microservice architecture. We are going to go over them here, in order to lay some groundwork that we can then use to build suggestions and make informed decisions and criticisms of the solutions and patterns that we are going to explore during this section.

Data Isolation

Data isolation in a microservices architecture refers to the separation of data storage and access for each microservice. Each microservice should have its own data store, separated from other microservices, to make sure that changes made by one microservice do not affect the data of another microservice. This is important for several reasons:

- Decoupling: By isolating data, microservices can be developed and deployed independently, without the need for coordination with other microservices. This allows for more flexibility and faster development cycles.
- Scalability: Data isolation allows for each microservice to scale independently, without being constrained by the data storage requirements of other microservices.
- Security: Isolating data allows for better control over access to sensitive information.
- Data governance: Isolating data allows for better management and governance of data, as it is easier to understand and manage data when it is separated by service.

However, it is important to recognize that isolating data can make it more difficult to share data between microservices, and may require additional effort to maintain consistency across multiple data stores. As we are going to see later on, you may chose sacrifice your data isolation, as a trade-off for ease of access and consistency.

You can implement data isolation by having each microservice use its own database, or by partitioning a single database by service. This can be done by using database views, schema or by

using different database technologies for different services. It's also important to consider eventual consistency when designing the data isolation.

Data consistency

Data consistency in a microservices architecture refers to the state of data across multiple microservices and data stores. In a microservices architecture, data is often distributed across multiple services, which can make it difficult to maintain consistency. When data is updated in one service, it may take some time for that change to propagate to other services and data stores, leading to inconsistencies.

There are different consistency models that can be used to maintain consistency in a microservices architecture:

- Strong consistency: With strong consistency, all services and data stores always have the same version of the data. This ensures that data is always consistent, but it can be difficult to achieve in a distributed system and leads to reduced performance.
- Eventual consistency: With eventual consistency, data is eventually consistent, meaning that it may take some time for all services and data stores to have the same version of the data. This model is more practical for distributed systems and can lead to better performance, but requires additional effort to handle conflicts and ensure that data is eventually consistent.
- Base consistency: With base consistency, data is consistent only at certain points in time, such as when a transaction is committed. This model can lead to reduced performance but can be easier to implement and may be sufficient for certain types of data.

Different microservices may require different consistency models. For example, a service handling financial transactions may require strong consistency, while a service handling user-generated content is usually able to use eventual consistency.

When designing a microservices architecture, consider the data consistency requirements of each service, and choose the appropriate consistency model for each service. It is also important to implement mechanisms such as distributed locks and versioning to handle conflicts and ensure consistency.

The CAP Theorem

The CAP theorem is a concept in distributed systems that states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:

- Consistency: All nodes in the system have the same data.
- Availability: Every request to the system receives a response, without guarantee that it contains the most recent version of the data.
- Partition tolerance: The system continues to function despite arbitrary partitioning due to network failures.

The CAP theorem states that a distributed system can only provide two of these guarantees at the same time. For example, a system that prioritizes consistency and availability can't tolerate network partitions, while a system that prioritizes availability and partition tolerance can't

guarantee consistency.

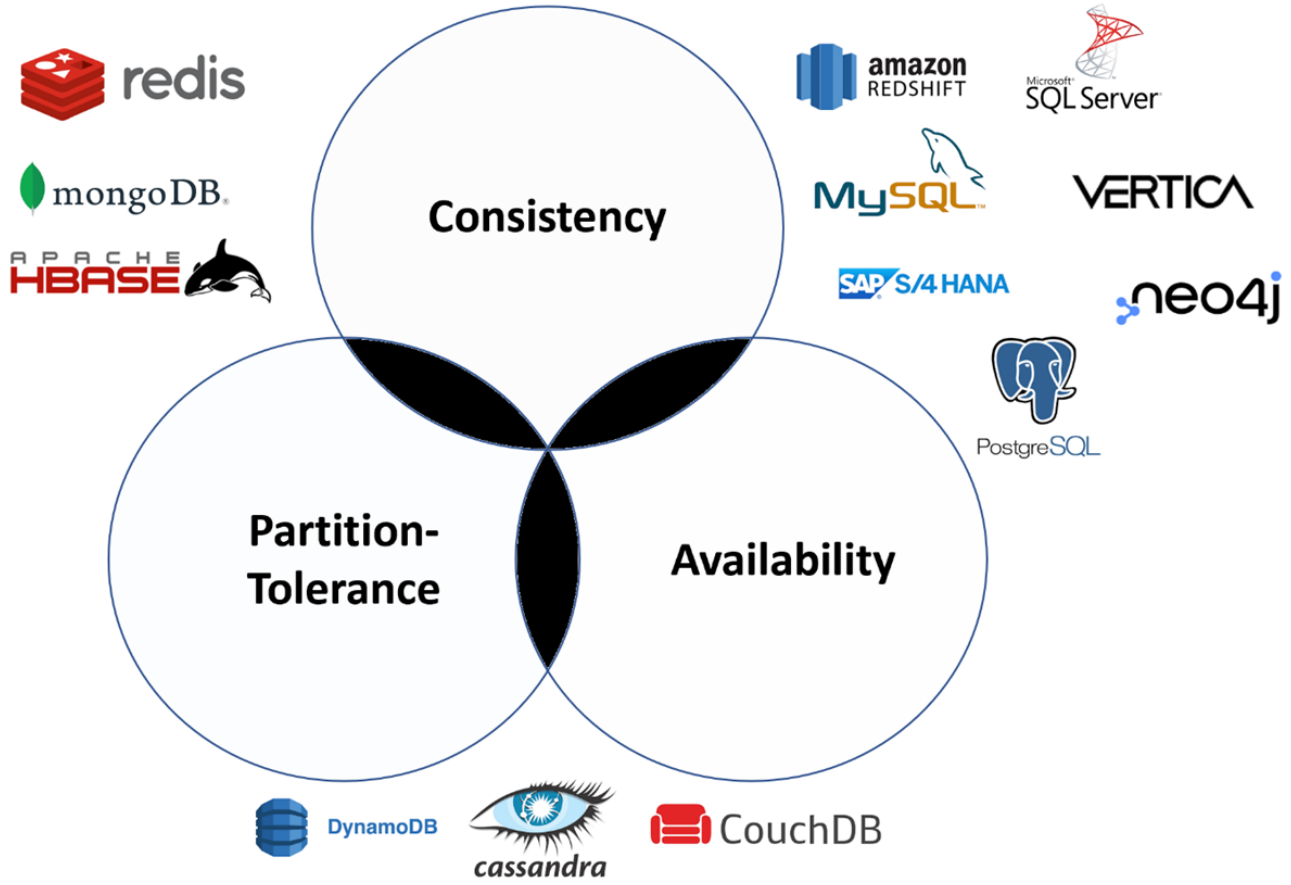


Figure 32. CAP Theorem

The CAP theorem is not a hard rule, but more of a guideline to help understand trade-offs of distributed systems. The theorem is a way to think about the different requirements of a system, and how the choices made for one aspect of the system may impact the others.

When designing a distributed system, it's important to understand the requirements of the system and the trade-offs that must be made in terms of consistency, availability, and partition tolerance. This can help to make informed decisions about the architecture and technologies used in the system, and ensure that the system is able to meet the needs of the organization.

The CAP Theorem is often used as a way to categorize databases solutions, and can play a role in choosing which database one will use. However, it is also useful when designing your data architecture system to better identify which of the previous characteristics you want to prioritize, and make the according decisions.

Relation with the microservice architecture

As you may have noticed, we already talked quite a lot about how we were going to architect our data within our system, during the first section of this paper, where we talked about how we can define our domains and microservices, with Polycode as support. We defined boundaries using both domain concerns, but also with transaction and consistency concerns.

Indeed, the data architecture for a microservices system should support the goals of the microservices architecture and vice-versa. We identified this relation, and made sure to create a

microservice architecture that were driven by those concerns. We should have an easier time defining our data architecture, and actually already have a microservice architecture that allows some wiggle room to use different solutions in our Polycode system.

Before diving deeper in how we can architect our data within our system, I want to make a step back and firstly look at the databases technologies available on the market today, and where their use case makes sense.

Different databases for different needs

In a microservices architecture, different microservices may have different data storage needs. Each type of database has its own set of strengths and weaknesses, and is better suited for certain types of data and workloads. In this chapter, I will explore the use cases, pros and cons of three types of databases: relational databases, document databases and in-memory databases. Each type of database has its own set of advantages and disadvantages. For example, document databases are good at handling flexible data models but doesn't provide the same level of performance as an in-memory database. On the other hand, relational databases provide robust querying capabilities but are not as well-suited for handling large amounts of unstructured data.

Relational databases

Relational databases are a type of database management system that store data in a structured format, using tables, rows, and columns. The most popular relational databases include MySQL, PostgreSQL, and Microsoft SQL Server.

Relational databases are based on the relational model, which is a mathematical model for representing data in a table-like format. Each table represents a specific entity, such as a user or a team, and each row represents an instance of that entity. The columns represent the attributes of the entity, such as the user's name or the team's captain. The relationships between entities are represented using foreign keys, which link rows in different tables together.

This type of databases are well-suited for applications that require complex querying and data relationships. The use of a relational model also allows for data validation and integrity constraints, which helps to ensure that the data stored in the database is correct and consistent.

Relational databases is usually queried using SQL, which is used to insert, update, retrieve and delete data from the database. SQL is a standard language that can be used across multiple relational databases.

However, relational databases are not always the best choice for every use case. They may not be as efficient as other types of databases at handling large amounts of unstructured data, and may not be able to scale as easily as some other types of databases, although we have some robust solutions nowadays. Additionally, the use of a fixed schema can make it more difficult to handle changes to the data model, and may require more effort to maintain backwards compatibility.

Overall, relational databases are a powerful and widely-used type of database management system, and are well-suited for applications that require complex querying and data relationships. However, it's important to carefully consider the specific needs of the application before choosing a relational database as the main data storage solution. In today's Polycode, we use relational database for storing users, teams, campaigns or transactions, since the schemas for those are well-defined, well-structured and present a strong relationship with other schemas in the database. We use PostgreSQL as our database solution.

Document databases

A document database is a type of NoSQL database that stores data in the form of documents, rather than tables and rows like in relational databases. The most popular document databases include MongoDB, Couchbase, and RavenDB.

Each document in a document database represents a single entity, such as a content or a module. The document can contain multiple fields, similar to columns in a relational database, to represent the attributes of the entity. Documents are stored in collections, similar to tables in a relational database. The collections can be searched and queried using a query language specific to the document database.

One of the main advantages of document databases is their ability to handle semi-structured or unstructured data. In contrast, relational databases rely on a fixed schema, which can make it difficult to handle changes to the data model. A document database can handle data fields that are missing or have different data types, and are more flexible when it comes to adding new fields or changing the structure of the data.

Another advantage of document databases is their ability to scale horizontally. They can handle high write loads and can easily scale by adding more machines to the cluster. This makes them a good choice for applications that have high write loads, need to handle large amounts of unstructured data, or need to scale quickly.

However, document databases have some trade-offs to consider as well. They doesn't provide the same level of performance as an in-memory database, and are not a good solution for handling complex data relationships as a relational database. Additionally, they doesn't provide the same level of data validation and integrity constraints as a relational database, which can lead to data inconsistencies.

In summary, document databases are a good choice for applications that require flexible data models, need to handle large amounts of unstructured data, or need to scale quickly. However, it's important to carefully consider the specific needs of the application before choosing a document database as the main data storage solution. We do use document database in the current state of Polycode, for storing contents, modules, submissions and validators for example. We use MongoDB as our document database solution.

In-memory databases

An in-memory database is a type of database management system that stores data in the main memory (RAM) of a computer, rather than on disk like traditional databases. This can make in-memory databases much faster than traditional databases, as data can be accessed and updated without the need for disk I/O. The most popular in-memory databases include Redis and Memcached.

In-memory databases are particularly well-suited for applications that require low-latency, high-performance data access. They are often used in applications such as real-time analytics, gaming, financial trading systems, and e-commerce platforms. For example, an in-memory database can be used to store real-time stock prices and perform real-time calculations on the data, or to store

session data for a web application and quickly retrieve it for a user.

In-memory databases can be used as a caching layer between the application and a traditional database, to improve the performance of read-heavy workloads. They can also be used as a primary data store for write-heavy workloads, where data needs to be quickly accessed and updated.

In-memory databases usually provide a key-value data model, which allows for fast and efficient data access. They can also provide a data structure such as a hash table, list, or set, to support more advanced data manipulation.

However, in-memory databases also have some limitations to consider. They are typically more expensive than traditional disk-based databases, as they require more memory. Additionally, they are limited by the amount of memory available on a single machine, which can make it more difficult to scale the system horizontally. In-memory databases also typically do not provide the same level of durability as traditional databases, as data is lost when the system is powered off or crashes, even if modern in-memory that focuses on storing application data as a primary database, such as Redis, provides way to periodically flush its memory to the disk.

Overall, in-memory databases are a good choice for applications that require low-latency, high-performance data access and can afford the higher cost of memory. They are often used as a caching layer or a primary data store for write-heavy workloads. However, it's important to carefully consider the specific needs of the application before choosing an in-memory database as the main data storage solution. We do not use any in-memory database in the current state of Polycode. However, we have seen that we might have interest in using one in the runner architecture, and would make sense in other places, that we will discuss later.

Every types of databases have their strength and weaknesses, exacerbated by the fact that we are running in a microservice architecture. We are now going to explore what are the constraints that this brings onto our data architecture concerns.

Availability and performance

When designing your data architecture, whatever the database type or solutions that you use, you need to think about the implications it will have on your overall system. In a microservice architecture, we want to scale, to be resilient and to be elastic. Performance is also a factor. Your data architecture needs to answer these constraints, else it will become a bottleneck in your system, since microservice are typically stateless, meaning that they can't function properly if the underlying data layer is not operational.

In this chapter, I want to focus on some solutions that are available in most widely adopted systems, who helps solving these problems, as well as some patterns you can implement yourself for improving performance, reduce system load and overall improve your resiliency.

Data replication

Data replication is the process of copying data from one database to one or more other databases, to ensure that the data is available in multiple locations. Data replication is a key aspect of data architecture in microservices, as it can be used to improve the availability and performance of the system.

There are several types of data replication, each with its own set of advantages and use cases:

- Master-slave replication: In master-slave replication, one database server acts as the master and one or more other servers act as slaves. The master server receives write requests and updates the data, while the slaves replicate the data from the master and can be used to handle read requests. This type of replication is useful for improving read performance, as well as providing a backup in case the master server fails.
- Multi-master replication: In multi-master replication, multiple servers can act as both master and slave. This type of replication allows for multiple servers to handle write requests, which can improve write performance and provide a higher level of availability. However, it can also lead to conflicts when multiple servers try to update the same data at the same time.
- Global distributed replication: This type of replication is used to replicate data across multiple data centers in different geographic locations, which can improve performance by reducing the distance data has to travel, and also increase availability by providing a backup in case of a regional failure.

When choosing a data replication strategy, it's important to consider factors such as the consistency model, the network latency, the security of the data and the business continuity requirements. Additionally, it's important to consider the trade-offs between availability and consistency, as well as the cost of the replication solution. Different solutions might not provides all of these replication types, but you usually can find solutions in every type of database that fits your needs.

Overall, data replication is a powerful technique for improving the availability and performance of a microservices-based system. By replicating data across multiple locations, it can help to ensure that data is always available and can provide a backup in case of a failure.

Data sharding

Data sharding is a technique used to horizontally partition a large dataset and spread it across multiple servers, or shards. The goal of data sharding is to improve the performance, scalability, and availability of a system by distributing the data across multiple machines.

When a dataset becomes too large to fit on a single server, data sharding can be used to split the data into smaller, more manageable chunks called shards. Each shard is stored on a separate machine, and the data is distributed among them.

There are several strategies for data sharding:

- Range-based sharding: With range-based sharding, the data is partitioned based on a range of values, such as a date range or a numerical range. For example, all data with an ID between 1 and 10,000 could be stored on one shard, while data with an ID between 10,001 and 20,000 could be stored on another shard.
- Hash-based sharding: With hash-based sharding, a hash function is used to determine which shard a piece of data belongs to. The function takes a piece of data, such as a user ID, and maps it to a specific shard.
- Directory-based sharding: With directory-based sharding, a separate shard is designated as the directory and responsible for routing requests to the appropriate shard.
- Sharding by functionality: Data is partitioned based on the functionality of the application. For example, all data related to user accounts could be stored on one shard, while data related to product information could be stored on another shard.

Data sharding improves the performance, scalability and availability of a system by distributing the data across multiple machines, but it also comes with its own set of challenges. One of the main challenges is to ensure data consistency across the shards. This can be achieved by implementing a distributed transactional system, or by using a consistency model such as Base consistency or Eventual consistency.

Another challenge is to ensure that the sharding strategy is flexible enough to handle changes to the data, such as the addition or removal of shards.

In summary, data sharding is a powerful technique that can help to improve the performance, scalability, and availability of a system by distributing the data across multiple machines. However, it's important to carefully consider the specific needs of the application and to plan for the challenges that come with data sharding. Sharding is usually used in Document databases such as MongoDB, where the eventual consistency model is used. You can also find it in Redis.

Data caching

Data caching is a technique used to temporarily store frequently accessed data in a fast-access memory store, such as RAM, in order to speed up data retrieval and reduce the load on the underlying data store. Data caching is a key aspect of data architecture in microservices, as it can be used to improve the performance and scalability of the system.

There are several types of data caching:

- In-memory caching: This type of caching stores data in the main memory (RAM) of a machine. In-memory caching is the fastest type of caching, as data can be accessed and updated without the need for disk I/O. However, it is also the most expensive type of caching, as it requires more memory.
- Disk-based caching: This type of caching stores data on disk, typically in a file system or a specialized data store such as SQLite. Disk-based caching is slower than in-memory caching, but it is also less expensive, as it requires less memory.
- Distributed caching: This type of caching stores data across multiple machines, using a distributed cache management system such as Memcached or Redis. Distributed caching can improve scalability and availability, but it also requires more complex configuration and management.

When designing a caching strategy, it's important to consider factors such as the size of the cache, the expected cache hit rate, the eviction policy, and the cache invalidation strategy.

Cache eviction policy is a technique to decide which item should be removed from the cache when it is full and new items need to be added. Popular eviction policies include Least Recently Used (LRU), Least Frequently Used (LFU) and random eviction.

Cache invalidation strategy is a technique to decide when and how the cache should be updated. Popular invalidation strategies include time-based invalidation, where items are removed from the cache after a certain period of time, and event-based invalidation, where items are removed from the cache when certain events occur.

Another important consideration is the consistency model of the cache. A read-through cache will always read the data from the underlying data store and update the cache, while a write-through cache will always write the data to the underlying data store and the cache.

To my knowledge, there is no self-hosted solutions that provides a package that wraps both a document or relational database with a strong caching system in front. Introducing a cache layer in your system requires careful considerations about where you use this layer and when you fetch or mutate your data directly with the database. It also requires writing application specific code to handle cache misses and the caching process.

Overall, data caching is a powerful technique for improving the performance and scalability of a microservices-based system. By temporarily storing frequently accessed data in a fast-access memory store, it can help to reduce the load on the underlying data store and improve data retrieval times. However, it's important to carefully consider the specific needs of the application and plan for the challenges that come with data caching such as cache eviction and invalidation strategy, consistency model and the size of the cache.

Architecture patterns

To better architect and standardized language between engineers, the system design community has defined multiple patterns that helps describing architectures, and how to solve certain problems. In this chapter, we will look at a few of them, with the advantages and drawbacks that comes with them.

Shared database pattern

The first pattern we are going to look at is the shared database pattern. As the name suggests, the shared database pattern is a microservices architecture pattern where multiple microservices share a single database. This pattern is useful when multiple microservices need to access the same data, and there is a need for consistency and transactional integrity across the data.

In this pattern, all microservices that need to access the same data are connected to the same database. Each microservice has its own schema, and the data is partitioned across different tables and rows. This allows each microservice to have its own set of tables, with its own data model, while still being able to access the shared data as needed.

One of the main advantages of the shared database pattern is that it allows for easy sharing of data across multiple microservices. It also allows for consistency and transactional integrity across the data, as all microservices are accessing the same database instance.

However, the shared database pattern also has some drawbacks to consider. One of the main drawbacks is that it leads to tight coupling between microservices, which make it difficult to change or evolve the system. Additionally, it can also lead to contention for resources and reduced scalability, as all microservices are accessing the same database instance.

To mitigate these risks, it's important to use a database that can handle the high read and write loads, and it's also important to plan for failover and replication. You also need to carefully manage the data model and use a database that supports data migrations. Lastly, it's important to monitor the database performance and to have a plan in place for dealing with bottlenecks or failures.

Another drawback is that it can lead to increased complexity in data management and data governance, as the shared data may be subject to multiple data models and data access patterns across different microservices, which may lead to conflicts and inconsistencies.

Overall, the shared database pattern is a useful pattern for microservices architecture when multiple microservices need to access the same data, and there is a need for consistency and transactional integrity across the data. However, it's important to carefully consider the specific needs of the application and to plan for the challenges that come with the shared database pattern. You might want to consider other approaches and resolve data consistency at the application level, rather than on the database level.

Database per Service pattern

The database per service pattern is a variation of the microservices architecture, where each service has its own dedicated database. Each service is responsible for its own data, and there is no

sharing of data between services. This pattern can be useful when services have different data models and different performance requirements, and when data consistency across services is not a concern, or handled properly at the application layer.

One of the main advantages of the database per service pattern is that it allows for a high degree of autonomy and flexibility for each service. Each service can use a database that is best suited to its specific needs, and can evolve its data model independently of other services. Additionally, since each service has its own database, a failure or bottleneck in one service's database will not impact other services, which can improve availability and performance.

However, the database per service pattern also has some drawbacks. Since each service has its own database, there is no centralized query engine, which can make it more difficult to perform complex queries across services. Additionally, since each service has its own database, there is no centralized data management, which can make it more difficult to manage data consistency across services.

To mitigate this problems, creating a strong API systems between your microservices is important to be able to do advanced queries when required. Using a messaging system can be a powerful when you need to propagate events on your resources, so that services can notify each other of data changes and keep their databases in sync. It's also important to have a plan in place for dealing with data migrations and to monitor the performance of each service's database.

The saga pattern

I want to stop in this explanation of the database per service pattern to describe a pattern that helps resolving the data consistency problem: the saga pattern. The saga pattern is a way to handle long-running, complex transactions that involve multiple microservices in a distributed system. It is a way to ensure that a series of steps that need to be executed in a certain order are completed, even when individual steps fail.

The saga pattern is based on the idea that a long-running transaction can be broken down into a series of smaller, independent transactions, called sagas. Each saga represents a step in the overall process and is executed by a separate microservice. The sagas are orchestrated by a saga manager, which is responsible for coordinating the execution of the sagas, and for ensuring that the overall process is completed successfully, or that it is compensated in case of failure.

When a saga is executed, it may update the state of one or more services, and it may also publish events that are listened by other sagas. The events are used to trigger the next step in the process, or to trigger a compensation step in case of failure.

The saga pattern can be used to handle a wide variety of use cases, such as order processing, customer registration, and account management. It is used to make the state of the system always consistent, even in the presence of failures, and it can also be used to ensure that the system can be easily recovered in case of failures.

The saga pattern also has drawbacks, mostly in the form of complexity, especially when dealing with large and complex transactions. Additionally, it is difficult to test and debug, since it involves multiple microservices and a complex coordination mechanism. It also requires the implementation of a saga manager and a mechanism for storing and managing saga state.

In summary, the saga pattern is a way to handle long-running, complex transactions that involve multiple microservices in a distributed system. It is based on the idea of breaking a long-running transaction into smaller, independent transactions that are orchestrated by a saga manager. It can ensure that the state of the system is always consistent, even in the presence of failures, but it also has some drawbacks such as complexity to implement, test, and debug, and increase in system complexity. Let's now wrap up this chapter about the database per service pattern:

To sum up, the database per service pattern is a variation of the microservices architecture, where each service has its own dedicated database. It can be useful when services have different data models and different performance requirements, and when data consistency across services is not a concern. However, it also has some drawbacks, such as the difficulty of performing complex queries across services and the difficulty of managing data consistency across services. Careful planning and management are required to mitigate these risks.

Polycode

Let's get back to Polycode and apply what we learnt to our system. In this chapter, we are going to look at the current data architecture, try to pinpoint the current flaws, as well as try to understand why the data architecture was made like this in the first place. Then, we are going to see what would fit with our new microservice architecture, and with our increased knowledge in how to make a data architecture that respects the requirements and constraints I have described previously.

Current data architecture

Here's the current data architecture:

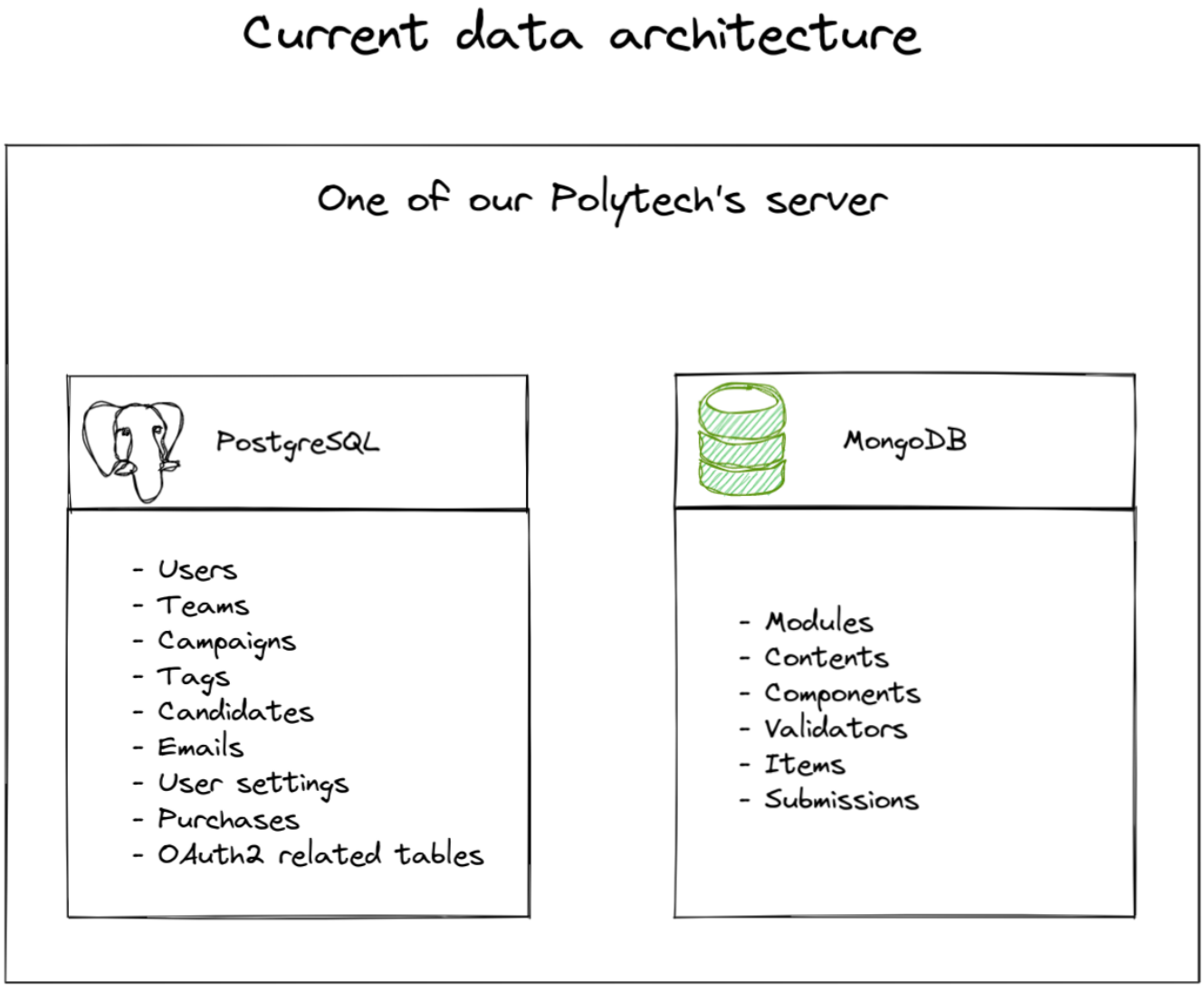


Figure 33. Current data architecture

As you can see, it's a very rudimentary architecture. We have two databases, a relational one (PostgreSQL) and a document one (MongoDB). Let's talk about these choices.

We picked a relational database for all the data that are inherently relational, with a well-defined structure. This includes users and teams for example, where we know all the fields, and we have a structure that is not changing over time, or at least have not yet changed. As we saw previously, this

is the kind of data where relational databases really shines, as well as providing us with a strong consistency that we can rely on for delicate operations, such as making a purchase, for example.

On the other hand, we have a document database (MongoDB), where we store all not so well defined data, such as validators, that can take different structure depending on the context or the type of content that is being validated. This is something that is cumbersome to do in a relational database. We still have some kind of relations between our collections, for example modules are composed of contents. MongoDB allows us to map this relationship between our collections. We don't have a strong consistency like we do with PostgreSQL, since MongoDB operates with on an eventual consistency model.

Currently, both our PostgreSQL and MongoDB cluster does not have any kind of replication, sharding or caching mechanism in place. This is primarily due to time constraint. With our current user base and volume of data, this has not been a problem, neither in term of performance or availability. However, this is still a huge point of improvement for our future system.

Target data architecture

Now that we have seen currently running data architecture and system, let's take a look at my suggestion for our target data architecture :

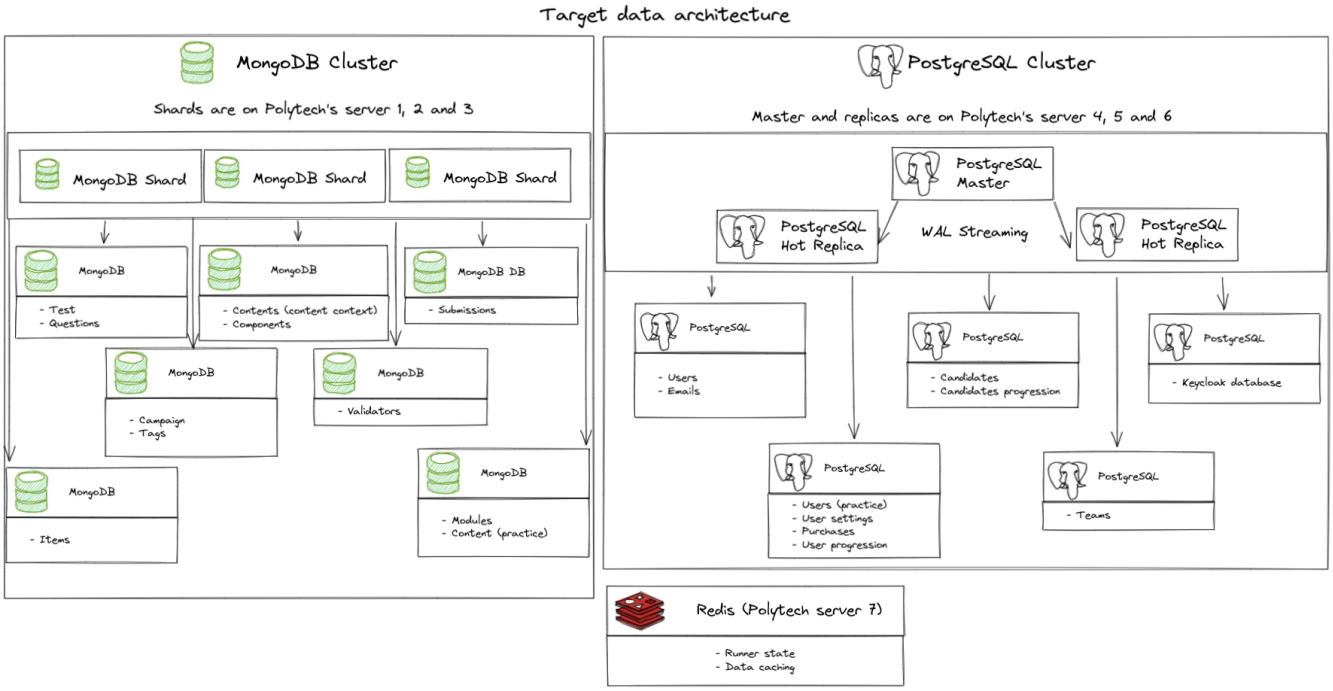


Figure 34. Target data architecture

As you can see, there's a lot to unpack here. Let's start by the pattern I chose to use. I you look closely, you can see that the numbers of databases (not cluster), almost match the number of microservices I've defined. Indeed, I've chose the Database per Service pattern. Let's start by addressing the main drawbacks of this pattern, and how I suggest to overcome them.

This pattern means that data consistency is not handled always by the database itself. However, using a proper division of our microservices, and with propers pattern in our application code, such as the saga pattern, we can ensure transactional boundaries at the application level. Accessing data from other microservices become a costly operations. We could replicate our data in other

databases, using message queues, but since we are not allowed to use message queues in our answers, I've discarded this option since it is not hindering the operation of the system too much. However, we still have to design a strong API system, and rely on caching techniques to accelerate access and ease the load on databases, more on that later.

By using this pattern, we gain the benefits of being able to evolve our microservices independently, decoupling them as much as possible, and we can make changes to the underlying data without breaking other systems as long as we don't change our API (which we should avoid).

I've decided that, although each microservice has its own database, to put all these databases in the same cluster for each database engine. This has the drawback of sharing resources and reducing performances, and decreasing the overall availability of the system. However, this is mitigated by running these clusters on replication and sharding models. All masters, replicas and shards runs on Polytech's server, since they have no cost for us. This is a problem in the case of a regional outage, but as an educational project and with the current reach of the website, this is acceptable.

The benefit of running in the same cluster, is an easier data management, since we actually run only 3 clusters, one for each type of database.

Overall, I've not made any changes about the data structure, and on what type of database data is stored. I think the current system will work fine as it is, although it is difficult to predict the ramification of switching to a microservice architecture. If we identify a major technical debt coming from this, we need to react quickly before building more features on the application, which will make it more and more difficult to solve the problem at the root instead of applying band-aid solutions that comes with a major burden on maintainability or performance.

I've also added a Redis server, which is not replicated neither sharded. Data store on this cluster are the runner state and caches for other databases. None of them are persistent data, and should be invalidated in case of a system failure anyway. It does not make any sense to replicate it. I think running a single node cluster will provide performance that are well over our needs as of right now.

Conclusion

We have seen the major concepts related to data architecture in a microservice system, identifying the accompanying problems and requirements. We have defined patterns that helps solve some of our problems, themselves coming with their trade-offs (as with everything in a microservice architecture). We migrated our current Polycode architecture to work with these constraints, coming up with a suggestion about how we can structure Polycode data architecture. As with all suggestions for Polycode in this paper, further discussing with the Polycode's team is needed before settling on a solution, but this provides groundwork for a possible architecture.

Adding a mobile app to Polycode

Introduction

The internet and websites are now mostly accessed through mobile devices. More and more people are letting their laptop in the drawer, using their phone as their only device to access the web. To reach this audience, creating a responsive website is not enough. People expect a native app experience, with a offline mode, push notifications, and a smooth user experience.

However, before diving into how we can integrate a mobile app into the system, I would like to discuss if it really makes sense to do so. Indeed, I would argue that Polycode functionalities are not suited at all for a mobile user-agent. Coding on a phone is a horrible experience that nobody is willing to endure. Following currently available lessons would also be a pain, since we provide interactive coding environments. It looks like all the current Polycode functionalities are not suited for a mobile app.

To me, justifying a mobile app would only makes sense if you are also providing a new set of functionalities that are mobile friendly.

What is important ?

Let's start by identifying what is important in a mobile app. Users has come to expect some behaviors from mobile applications, and failing to match those behaviors will cause the user to have a bad experience.

Offline

Although we live in a country (France) where cellular data is really cheap, we need to consider that a lot of people around the world does not have this privilege. Creating an application that relies on being connected at all time may seem not that big of a deal, but in reality, it is.

We need to provide a set of functionalities that are available offline, the goal being to have the same experience whether you're online or not.

Sets of contents packaged with the app

The first obvious approach is to bundle the application content with the app. This makes the app not reliant on connectivity to work. However, it comes with its disadvantages. The app is bigger to download, updating content means going through each platform long and painful validation process, which also makes it very difficult to rollback or fix broken updates, we will touch on that later. We don't want the user to update the application everyday for the eventual daily content either.

Fetch data when possible

We can take the opposite approach and not ship anything but the application logic in the app, and download contents whenever the device retrieves connectivity. After all, the user is most likely to open the app right after downloading it, meaning he has connectivity. Moreover, I would argue that the user will need to log in before using any features, guarantying that the app has the ability to download contents before the user needs it.

Best of both worlds ?

The previous approach looks like the best for the user experience, but it has some issues:

- This increases significantly the load on the infrastructure, for contents that are stable, bug free, known to be working and that is going to be downloaded anyway
- The user might have a very limited bandwidth, which may cause the application to be too data-hungry for them

Mixing both approaches will limit or eliminate their drawbacks, but it is important to note that it is at the cost of the app complexity.

Handling releases

The mobile app distribution is controlled by basically two actors (Google and Apple), who both have strict review processes for each applications and each of their updates. This means that relying on

updating the application to release new content or fix mundane bugs becomes a long process, and you need something that can circumvent those limitations.

This is the problem of some frameworks out there, that promises a fast and easy development, for both platform, but actually bundles the rendering engine with the application as well as your application code, meaning that there is no way to update the rendering, except updating the app. This might be a tradeoff you're willing to make, but I don't think it's worth the pain.

On the other side, you can opt to develop an app that is closer to a web view, that can be updated on the fly, without needing to go through the painful verification process of each store. Using the right framework, you can have the same code base (modulo some minors tweaks) for both iOS and Android, while using natives bindings for the rendering, and having views that can be updated on the fly.

With that being said, let's switch to another important aspect of a mobile app.

Short in time

We want users to be able to use the application as much as possible. This means, the user must be able to spend some time on the application whenever they have a few minutes to spare. By extension, we must give them functionalities and things to do that are short in time and that can be completed within a few minutes, maximum. This creates an incentive for the user to come to the app when they have a little bit of time to spare, or when they want to take a break of whatever they are doing. This, plus the educational aspect of the application, contributes to fight the feeling of wasting time and procrastination the user might have.

Repetitive

As an application, you want your users to come back to it as much as possible. What can we do to make our users come back ?

Create an habit

The first thing that is important to recognize is that for a user to come back, we need them to create a habit. A habit is triggered by an event, and results in the brain associating this event with an action. For example, a smoking student might say that, at every class break (the trigger), he goes smoking (the action).

We need the user to create this kind of habit. To do so, we need to provide something that incentives the creation of this habit, like for examples, daily contents, that is rewarded whenever it's completed, with a system of streak that pushes the user to not miss the next day content.

Deconstructing a habit is hard, harder that it is to create it. Once the user has this habit, we can try to use it as a way to push them towards the desktop application, where more in-depth content is available, and where he can find the full set of functionalities of the application.

Push notifications

One powerful tool that is available on mobile, are the push notifications. This is a way to remind the user of our daily contents. This might be both the tool that will make the user creates the habit, but also the trigger on which the user will then open the app.

One way of increasing the effectiveness of push notifications, is to time it well. You don’t want to send this notifications at 9h30, right after the user got to work. You might want to send this notifications at 7h30 in the morning, while commuting, at 1pm, during lunch break or at 6pm when commuting back from work. This can be fine tuned based on the user behaviors, and might take multiple days for the app to get it right. But creating an algorithm that tries to figure out where the user is most likely to have free time and send that push notification should give a great trade-off between the resources put into creating it, and its reward, although difficult to measures.

Mobile only ?

A concern you might have, is how to integrate these functionalities into the Polycode website ? I would argue that the question is more along the lines of *Do we need to integrate these functionalities into the Polycode website ?*.

Gate-keeping these functionalities will create an incentive for desktop users to download and use the application. If you're already spending time on the website, pushing them towards another platform where they can interact with our service when they have a little bit of free time, like discussed previously, will not be difficult. This would both increase the time the user interacts with our service, and create new habits so they keep coming using our service. They should not be pushed to aggressively, as this will impact the user experience, and also create a surface for backlash. I would like to point out that this is mostly a executive decision, and not a technical one. When building the backend for those new features, you should do it in a agnostic manner: any user-agent should be able to interact with it, letting us the possibility to integrate easily other platforms in the future.

Mobile functionalities

As discussed above, to justify a mobile app, we need to provide a new set of functionalities that are mobile friendly, with new contents available daily. Here are some ideas, we will dive into each of them, in details, later in this document :

- Daily short coding lessons
- Daily coding quizzes
- Find the bug

Daily coding lessons

Fitting into the short-in-time and repetitive constraint we have, I would propose daily coding lessons, which would be short (5 to 10 minutes) lessons the user can follow. You should not have to follow the previous lessons to be able to follow the current one. This might look unintuitive at first, but I think we trade the benefits of having a tailored progression for each user for having a easier to make, broader content that we can really focus on each day. This would also have the benefits of making it a topic of conversation among the Polycode mobile app users. If the lesson is the same for everybody, it means that they can talk about it every day, discussing the significance of what they learnt and how they can better use it in their life. However, making unique lessons that have no progression curve might also be boring for the user after a certain time, and finding new interesting content that has not be covered will become more and more difficult over time. This is a trade-off, and we keep the other approach in mind when implementing it, making sure we can easily extend the functionality to be based on the user's level later on. The lessons should be able in offline as well. As soon as the app retrieves connectivity, it will try to download new lessons, which can be published multiple days in advance. This way, the user will always have a lesson to do, even if they are offline for a few days. To incentivize the user coming back everyday, we should establish a streak system, keeping count of how long you have been doing your daily lessons. The user will not want to lose its streak and try to find time to complete its lesson. I think we should also grant the user Polypoints at different stages of progression, that they can use to buy things in the shop (which is currently only composed of hints for desktop exercises).

Daily coding quizzes

On the same model as the daily coding lessons, daily coding quizzes would be a small quiz, 5 minutes maximum, that would also be pushed to the user every day. The quiz would relate to the lesson of that day, and have the same streak system than the daily coding lessons. However, they would be pushed later on in the day than the lessons, to make sure the user correctly understood and remembered the lesson of earlier. For every answer, a short explanation of why this was the correct answer should be displayed, even if the user was right. The quiz should be downloaded when the device retrieves connectivity, ideally at the same time as the lesson.

Find the bug

The daily contents are a great feature and idea for our mobile app, but we also need contents that is available whenever the user wants to use the app, and not to be limited to the daily content. For a

start, we could provide a way to go back in time and catch up the lessons and quizzes missed. But I would also propose a small game, where you are provided a code that is not working and you have to figure out what is the problem.

The idea is simple: show a small sample of code, give some context about what is should be achieving, explain unwanted outcome of the sample of code, and let the user figure out what is wrong.

Click on the wrong line

The user will try to read the code, and when he identifies the faulty line of code that's causing the problem, he clicks on it. The user is given 3 tries before losing the game. If the user manages to find the faulty line, a small MCQ appears.

Explaining the problem

To make sure the user actually understood the problem, he will have to select between 4 explanations of what the faulty line is doing wrong. They will be given one chance to answer correctly, and if they do, they are granted a reward. More in-depth features can be explored, such as using Polypoints to have hints. Since the progression is tracked per-user, we can introduce more and more difficult bugs, starting with simple iteration loop, to buffer overflows and dandling pointers.

Application flow

To better illustrate the ideas I've just talked about, and the overall feel of the application, I've made mockups of the application UI. This is in no way a definitive design, and please forgive me for the not so aesthetic layouts, I'm not a designer. However, it conveys the flow and the feel of the app, and that's what I'm interested about.

General layout

Before diving into the aforementioned features, we are going to look at what the user is greeted with the first time he opens the application, the connection page:

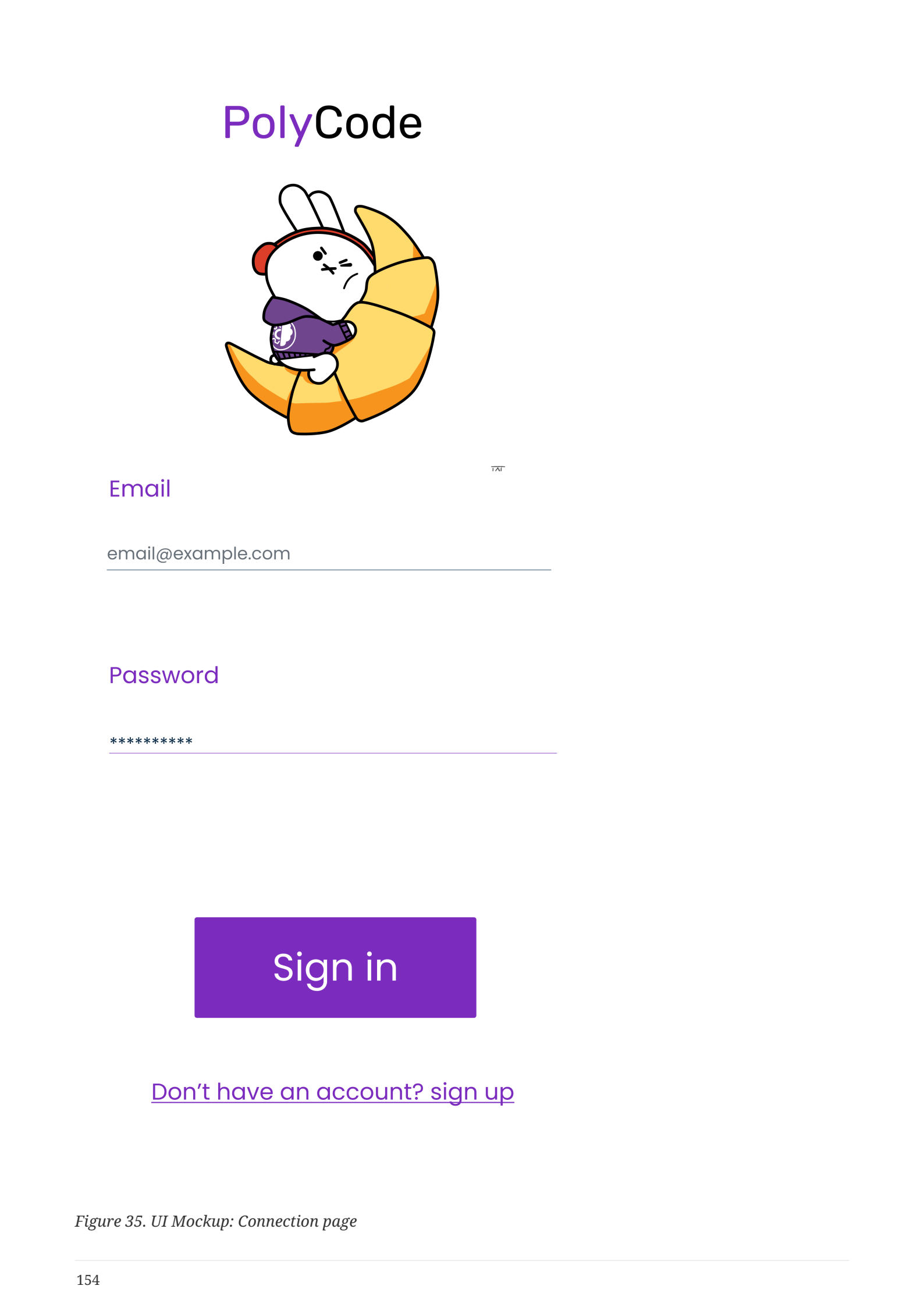


Figure 35. UI Mockup: Connection page

As you can see, I kept the overall theme and components of the web application. This is great for multiple reasons:

- We keep our overall brand identity
- It makes it easier for users to navigate to and from the web app to the mobile app
- The user will feel less frustration, since it recognizes some images, logos and themes. It will increase conversion rate for users who are already using another platform.
- Less design to do, we only need to adapt existing layout and themes to match the mobile version.

I think that, with some little adjustments, the actual design and graphic identity of Polycode is suited to a mobile app. I think this is further demonstrated by the example of an home page that I've made:

Figure 36. UI Mockup: Home page

At the top, we find a header that will be present all throughout the app. We will see later that, when navigating to the different features, this banner changes to display the part where the user currently is. This helps we navigating the application, even though its very rudimentary. On the main page, we find displayed all of our main features. I've included the mobile-only features that I've mentioned and explained above, as well as a leader board. For the daily lesson, the title of the current one is displayed. For every "daily" features, a tick is shown if the user has finished today's content (like on the daily lesson). A small text is displayed to inform the user about its streak, the text being adapted on whether or not the user has done the content. Find the bug and the leader board are static texts. The main page is very simple, with only the four aforementioned tiles that the user can interact with. As the application grows and more featured are being added, we might want to revisit this display to have a better arrangement, but also include some general information, news, notifications or whatever would make sense on the main page.

We find our purple-accented theme in titles and logos. The logos shown here should be worked on, as they don't feel quite right but they still give a great preview of the objective.

At the bottom, we find a navigation bar, that, just like the header, will also be displayed all throughout the application. This navbar allows the user to navigate to the main page, the 3 mobile features and also its profile. Those are effectively tabs, that you could switch from one another by tapping icons or by swiping left or right. This is a usual flow in modern portrait application that have different features, such as Snapchat or Instagram, that the user should already be familiar with.

Now that we have seen the general layout, let's take a look at all our mobile features !

Daily lessons

Introducing to you, our daily lessons user interface:

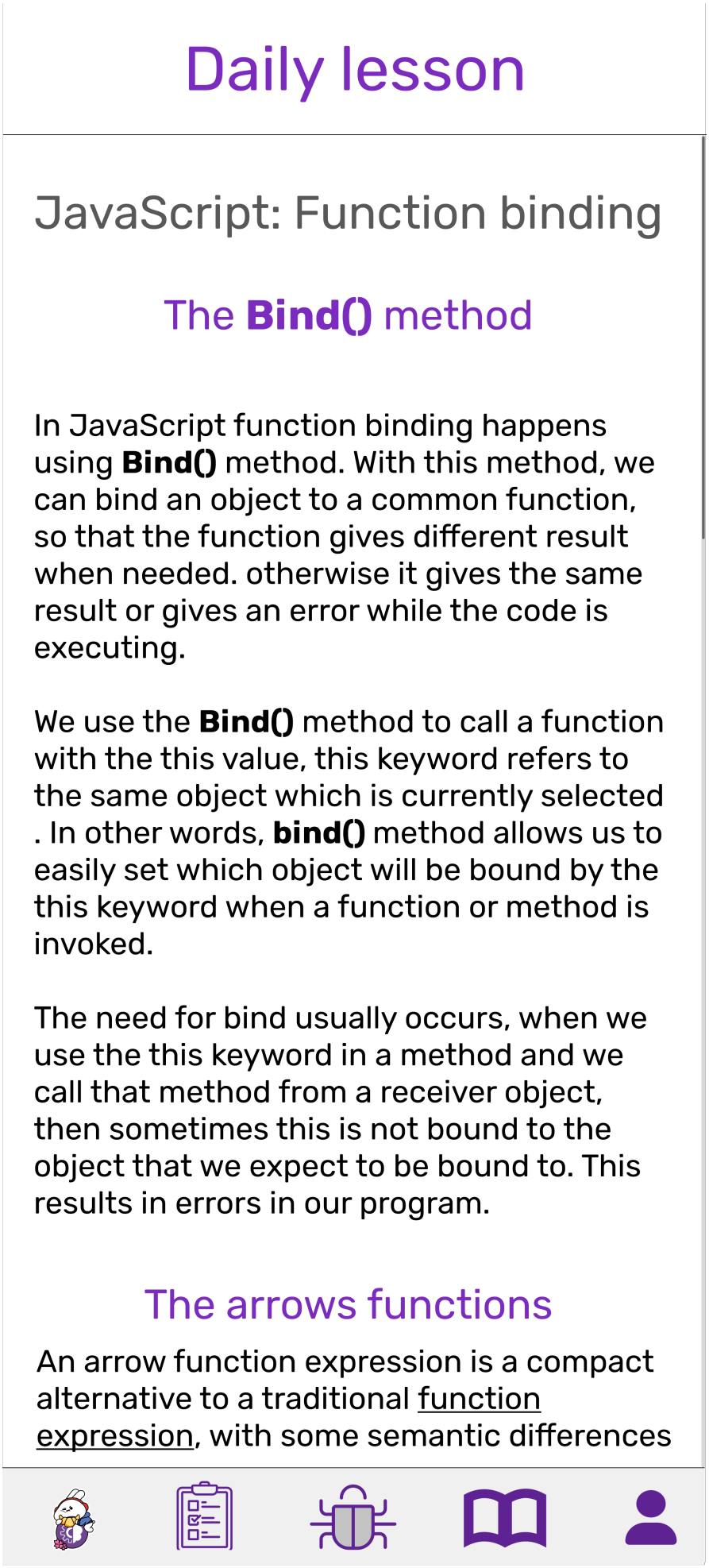


Figure 37. UI Mockup: Daily lessons

As you can see, it's actually pretty simple. The main goal here is to display text properly for a phone format. This is just a scrollable article, that can integrate small pieces of code, but the gist of it will be text organized with parts. We need to define a layout and style consistent across all our daily lessons, to give a easier reading experience for the user on the long-term.

- The title of the lesson is at the top, centered, in a bigger font size.
- The title of a section is displayed in Polycode's purple, also centered.
- Paragraph is written in black. Important word are in bold.
- Code can be formatted within paragraph, the styling must be consistent.
- The font should always be the same (the same that everywhere on the app).
- References to other lessons (or external sources) should simply be underlined.

A discrete scrollbar is present on the right, and is persistent. Making it persistent allows the user to be able to judge where in the lesson he currently is, without much inconveniences.

Once again, this is a pretty simple page. Let's move on to the next feature.

Daily quiz

In a logical manner, I will now show you and discuss what my mockup for the daily quizzes is:

Figure 38. UI Mockup: Daily quiz

As explained before, we find our navigation bar at the bottom and our header stating that we are in the daily quiz at the top. The main content is divided in two parts: the question and the answers. For this quiz, the question includes a code sample. The user is asked to explain what will be printed to the console. The code needs to be big enough that its comfortable to read on a phone, and the size used here is big enough, since it is about the same font size as the main font size in the application. I used rounded corners for a better feel for both the code window and the answers, to make the application feel less boxy. The answers needs to be separated from one another to leave as little room as possible for user accidental pressing on the wrong answer. This is why I've used all the space possible in the window. We must not forget that the quiz does not necessarily include a code sample, but since the daily lessons and daily quizzes are related, this quiz made sense, and allows me to show you what code samples would look like. The overall layout should be following the same rules as for the daily lessons, but can be tailored to fit a particular use case.

Don't forget that what I show you are suggestions. I'm not a designer and the final layout and styling could be different. With that said, let's move on to our last mobile feature.

Find the bug

As explained previously, find the bug is a small integrated game where the user is explained a problem with a piece of code. They then click on the faulty line, where a question about what the problem actually is is shown, where they have to demonstrate their understanding of the problem. Here's a mockup for the first part:

Figure 39. UI Mockup: Find the bug, first step

It is also a pretty simple page. There is a description of the problem at the top. In this case, we have a segmentation fault somewhere in our code. Right below it, is the piece of code that is responsible of the problem. It is displayed in a similar way that what we say in the daily quiz, except that the line spacing is way larger. This is due to the fact that the user needs to click on the line. It means we need to format the code in a way that make it adapted for this use case. With the spacing shown in this example, the user should have no problem touching the line he wants to touch, without worrying about pressing the wrong line by accident.

Once the user clicks on the faulty line, the user is taken to the next step, displayed on a new page:

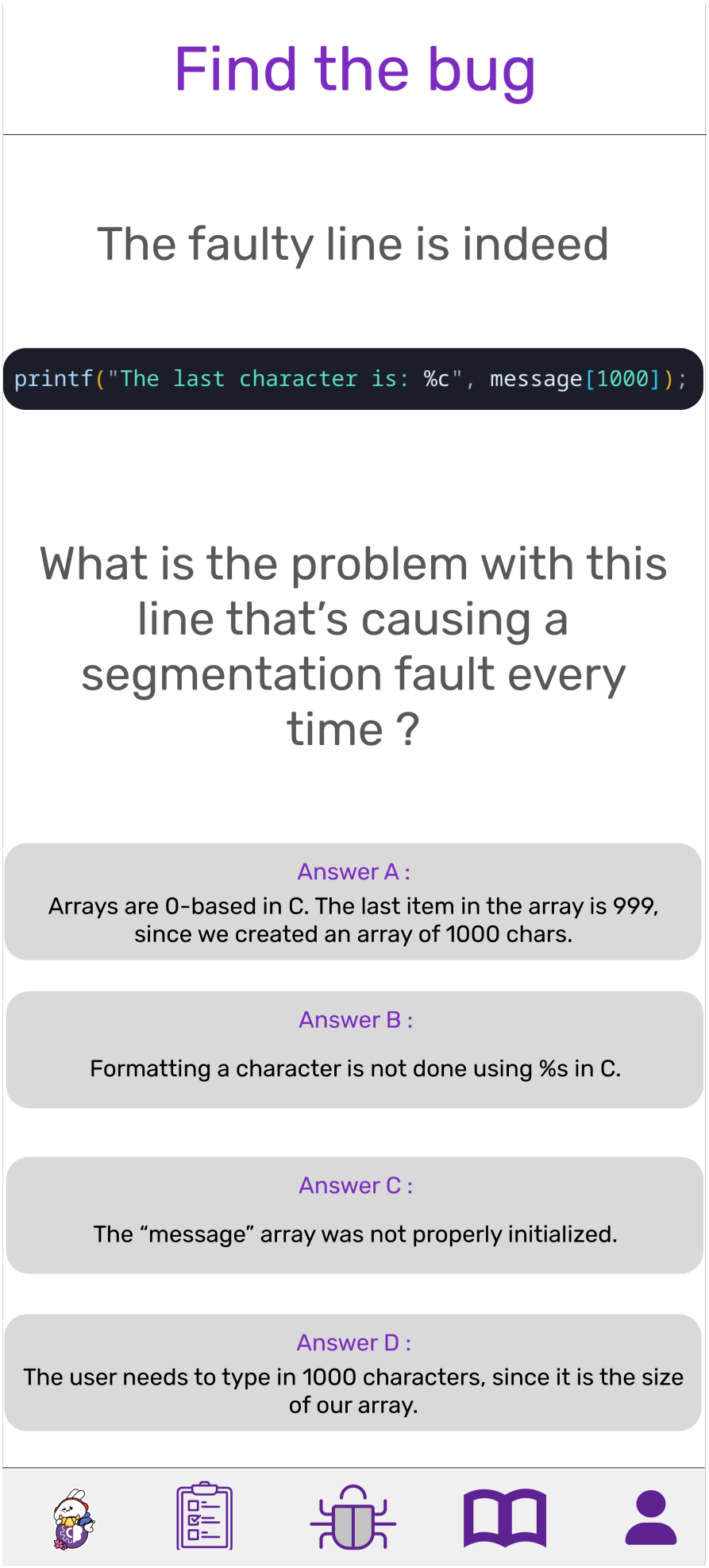


Figure 40. UI Mockup: Find the bug, second step

At the top of the main content, we have a reminder of what the user clicked on. The user can validate that it is indeed the line he thought he clicked. But more importantly, we need to show it here so he can better inspect the faulty line. Some feedback would be needed to see if users feel like they need to have the whole code sample to better contextualize the faulty line. For now, I've decided it was not needed.

Just below, we find the question that verifies that the user understood why it was faulty. It is in the same form, with the same layout and the same styling as with the daily quiz.

Let's now look at the endpoints that needs to be implemented in the backend for our features to work.

API

Designing an API properly is a task that is often overlooked. Your API defines the way your developers (or external developers) interact with and access the functionality provided by the underlying Polycode service, in our case. A well-designed API saves developers time and effort by providing clear and simple interfaces for performing common tasks, and can also promote the adoption of the service or application by making it more accessible to a wider range of users.

The main goals of API design are to make the API easy to use and understand, efficient, flexible, and stable. A good API should be easy to learn and use, with clear and concise documentation and error messages. It should be efficient, with minimal overhead and fast response times. It should be flexible, allowing for a wide range of use cases and integration with other systems, and it should be stable, with minimal changes or breaking changes over time.

I'm going to present the necessary endpoints I've identified for our mobile features. Those will be the one the application will interact with, and not the administrative one (such as creating a new find the bug challenge, for example). All the already existing API that the mobile app will need to consume are not described here, such as authentication endpoints or contents endpoints. Keep in mind that this is a draft, and discussion with domain experts, technical leaders and overall, the Polycode team will be needed to refine this proposal.

The API will be described in the Open API format. The file is available here. I will provide related images of the endpoint or OpenAPI resource that I'm talking about during this section for readability purposes.

This API is divided in 3 different resources: daily lessons, daily quizzes and find the bug. Those are our 3 mobile features, and they are conceptually also 3 different resources. Let's first take a look at our daily resources first:

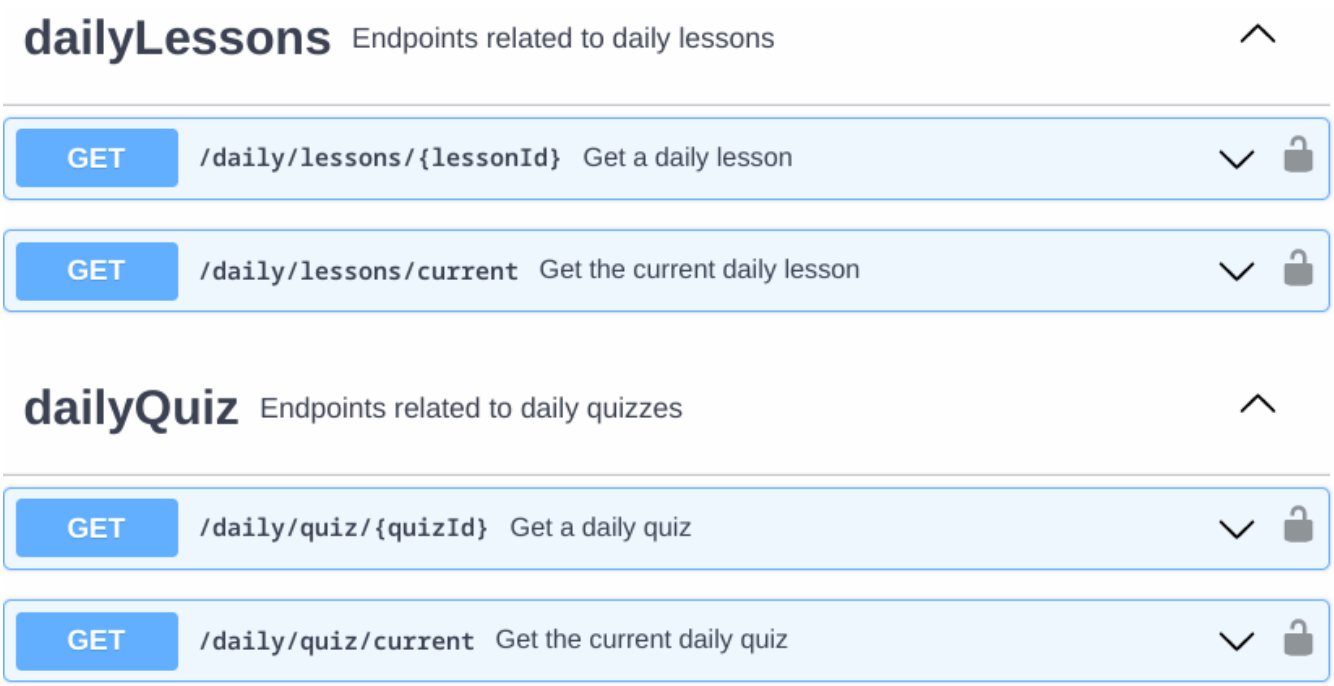


Figure 41. API for daily resources

We have only defined two endpoints for each of them ! It might be surprising, but we will also consume existing endpoints when submitting answers or fetching contents, for example. For both of our dailies, the endpoints are actually very similar. One allow for accessing one of these resources by ID, and the other is there to get the currently running daily lesson or daily quiz.

The mobile app will get resources' IDs when receiving notification from our backend. With this ID, it can then fetch contents when it the application thinks its suitable.

Let's look at the schemas for the daily resources:

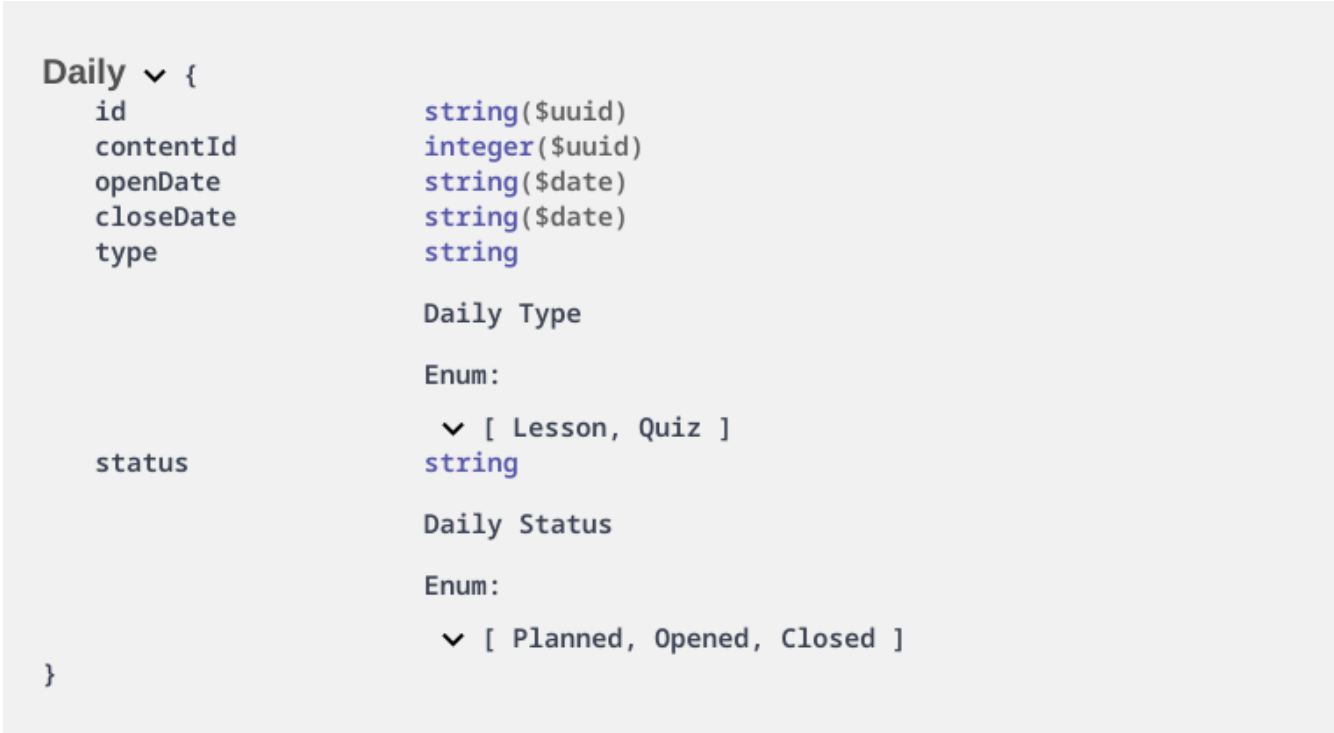


Figure 42. API schemas for daily resources

There is only one for both of those resources. Indeed, they need to store the same kind of

information. The `contentId` field, which directs to a content that will be responsible of showing the actual content, `openDate` and `closeDate`, that deals with having timestamp of when the daily challenge opens and close (this should be a 24-hour window, but I decided to include both to have more flexibility). If something's has failed when getting the notification to the application about the next dailies, the application can fallback to using the "current" endpoint to get the currently running daily. The last fields `type` and `status`, are responsible for identifying the resource, and its state, respectively. They are a bit of redundant information, but I figured it would make the life of developers easier. Once again, keep in mind that this is a draft and further discussion with the team is required, for this kind of specifics.

As for all the endpoints defined in this OpenAPI design, they are protected with our authentication cookie. This is mostly for rate limiting and authenticating consumers of our API.

Let's look at our last two endpoints, related to the "find the bug" feature:

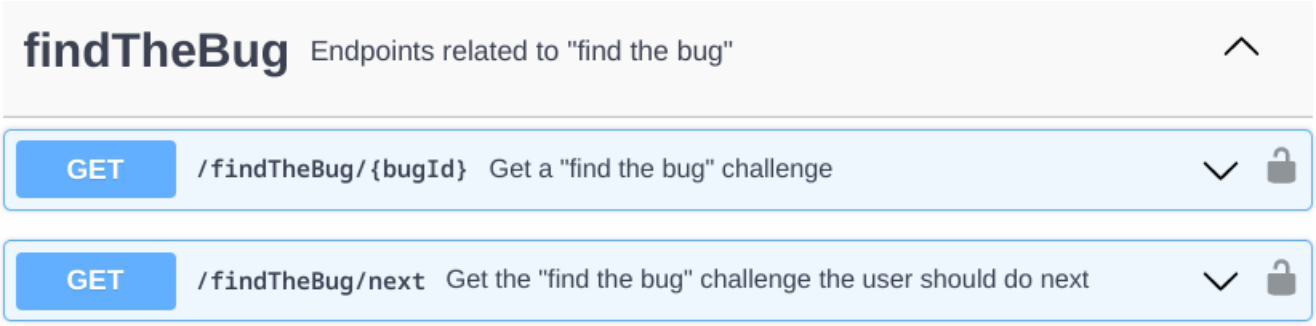


Figure 43. API for find the bug

As with the dailies feature, we have an endpoint for fetching a specific find the bug resource by ID. But I want to focus on the other endpoint, the "next" endpoint. Unlike with dailies, each user has its own progression, meaning that we can introduce harder and harder challenges. This responsibility should be handled by the backend, and is abstracted away in this single endpoint, that returns the resource it identified as the best for the user to do next, taking in variables such as already completed challenges of the user.

Here's the schema for the find the bug resource:

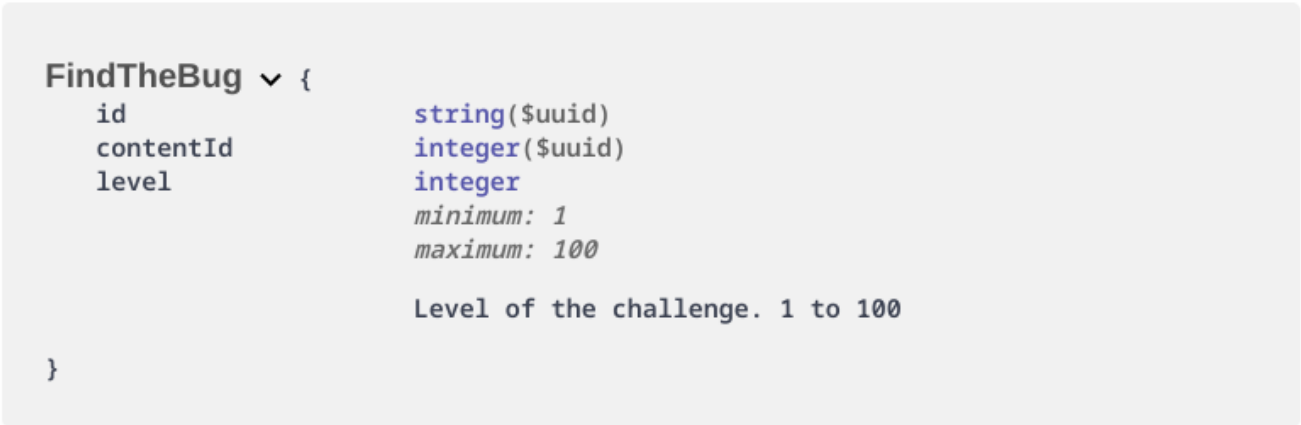


Figure 44. API schemas for find the bug

It is very simple. Just like with dailies, we have a unique identifier and `contentId` that is responsible of storing which content is actually holding the content of this resource. We have an additional

field, `level`, which indicates how hard this challenge is, on a scale from 1 to 100.

I'll repeat myself here, but I want this to be clear: this is a suggestion, a draft that needs to be worked on, and requires collaboration with the whole team. This can serve as groundwork for further refinement.

Now that we have defined the needed endpoints for our mobile features, let's look at how we can handle authentication within our mobile app.

Authentication

Requirements

The typical mobile user has come to expect a smooth user experience, with a native app feel. This means that we need to provide a login experience that is as smooth as possible.

Keeping the user logged in

One of the main behavior users expect, is not to have to login every time they open the app. All the the application you use daily, even those who might be sensitive, does not require logging in after the initial setup. We want the user to access the application as fast as possible, and logging in is a barrier to that. Moreover, this would cause a displeasing experience for the user, entering your credentials is not fun.

We need to find a way to make sure the user needs to log in only once.

Not storing the password

With that being said, one big requirement we need to respect is not to store the user's password at all. Doing so is a big security risk if their device were to be compromised, albeit by a malicious actor or by someone gaining physical access to the device. This is harmful to the user, since he might be reusing this password elsewhere, but also for us, since the attacker have access to the whole account, not only the session in the application.

How to do it ?

With the main requirements laid out, we can now design a system that respects them. Thankfully, our current authentication system allows us to do that with close to no changes ! Indeed, the mobile app should just consume the authentication endpoint, getting back their session cookie that we can store and use everytime the user opens the app.

To better identify where the connection is coming from on the backend, I suggest using a custom user-agent that would identify the connection as coming from the application. This would let us add some custom code in order to have some differences in behavior between the two ways of connecting, if needed.

Sequence diagram

Here's a reminder on the authentication and authorization process:

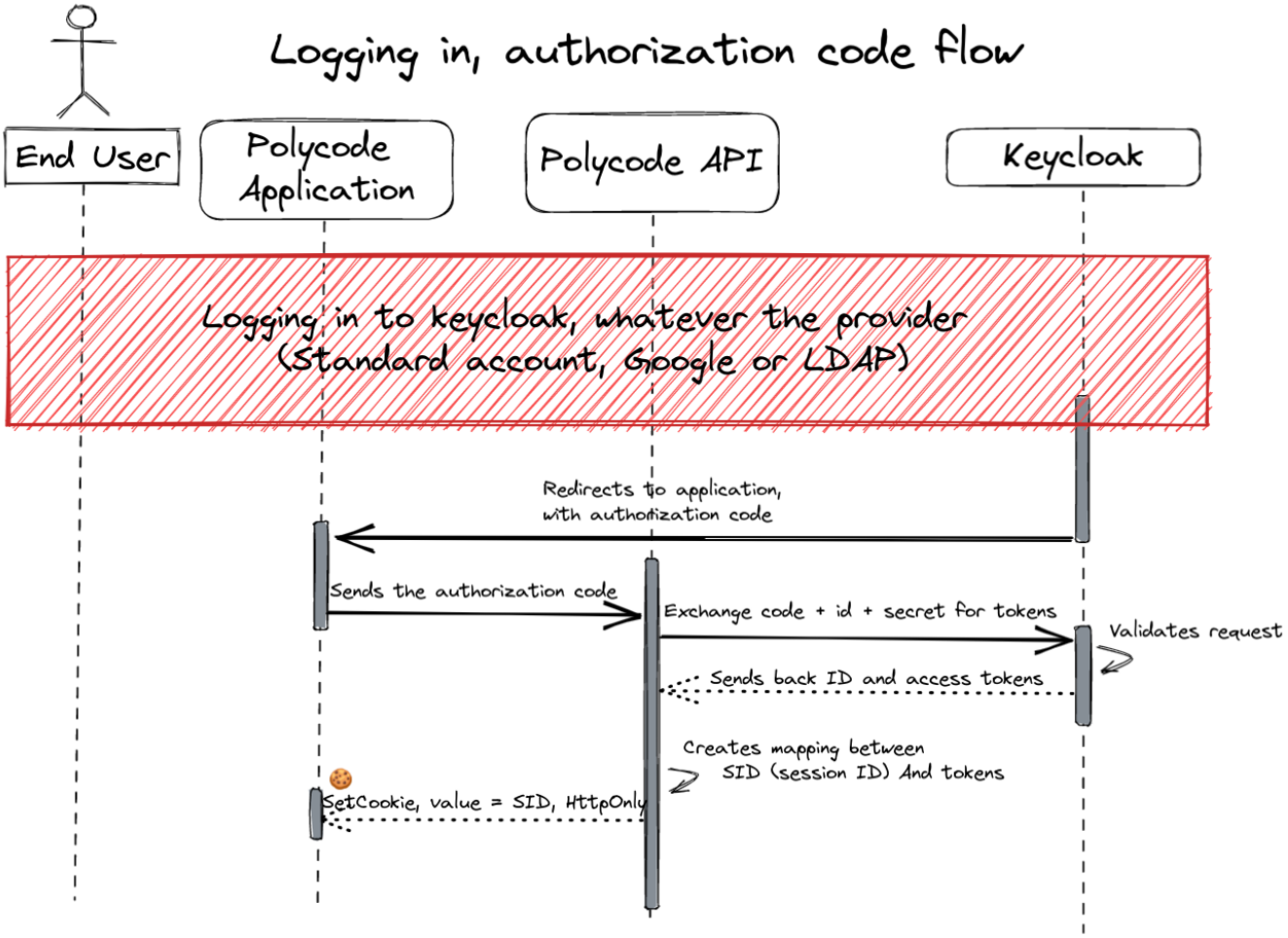


Figure 45. Authentication on mobile, sequence diagram

It is the same as in section 2, except that the user interacts with the application and not the web frontend. For more details, I suggest checking the section 2 where we go in depth on the whole process.

Conclusion

This section showed you the importance of identifying that mobile application doesn't have the same requirements and usage as a website. We need to identify what the use case is for our app, and how to correctly respond to that. You also need to recognize that you can't always expect the user experience to be the same on your mobile app and on your website, and you need to pick features or create new one accordingly.

For Polycode, I decided to target the mobile application towards a repetitive but short usage by the user, during break time or commutes for examples. With that in mind, and the fact that coding on a phone is not enjoyable, I've defined some features that are designed with the previous constraints in mind. We saw how we can implement them at a higher level, by defining the contract that will be used to communicate with our backend, via an API definition, as well as our authentication flow, respecting today's expectations for mobile applications.

Security

Introduction

In the previous sections, we addressed a lot of subjects related to microservices and Polycode. However, there is a very important aspect of all compute systems that we haven't talked about yet (or very little): security. In this section, I'm going to have a deeper look into securing our system, in a world where cyber-security attacks are done at a massive scale, and where malicious users and bad actors are more and more proactive.

What even is security ?

Security, in the context of a web application with a microservice backend, refers to a set of practices and technologies that are used to protect the application and its users from unauthorized access, data breaches, and other malicious activities.

Each deployed services has its own set of responsibilities and are ran in its own isolated environment. This architecture requires a different approach to security than a traditional monolithic application, as it requires coordination between multiple teams and a focus on securing individual services rather than the entire system. Security happens at multiple levels in our architecture:

- At the frontend
- At the backend
- At the operation level
- In the communication between all our actors

Security is not as simple as implementing something somewhere, and your system becomes secure. It requires an accurate understanding of how your system interacts. A team with a security mindset leads to a secure system, since every task, implementation, deployment and configuration have security concerns coming with it. Security should be a concern that every member working on a project should have. Security should be an integral part of the development and deployment process for any web application.

There is also no definitive way to have a 100% secure system. 0-risk doesn't exist, and it is important to realize that. With this fact in mind, the general objective changes. It becomes about reducing the surface of attack of your system, careful monitoring and auditing of your system and having well-defined incident response plan in case of an emergency.

Why is security important ?

Now that we have a better understanding about what security is, let's talk about why is security important. I'm going to assume that we are a company and not an educational project.

Security is important for several reasons:

- Protection of sensitive data: Security measures protects sensitive information, such as personal data, financial information, and confidential business information, from unauthorized access or breaches.

- Compliance: Many industries have regulations that require organizations to implement certain security measures to protect sensitive data, such as GDPR. Compliance with these regulations is important to avoid fines and reputational damage.
- Business continuity: Security measures helps making the organization's operations continuous in the event of a security incident. This includes disaster recovery plans, backups and other mechanisms to ensure the availability of systems and data.
- Protecting against cyber threats: Security measures help protect against cyber attacks, such as malware, which can cause data loss, system downtime, and more.
- Trust and reputation: Security is essential to building trust with customers, partners and stakeholders. It's important for organizations to be transparent about their security measures and to demonstrate that they take security seriously.
- Protecting intellectual property: Security measures help protect an organization's intellectual property, such as trade secrets, patents, trademarks or the codebase from theft or unauthorized use.

Let's get back in the context of Polycode. I'm going to talk about the biggest security concerns that we might encounter, categorized in 3 categories: frontend, backend and operation.

Frontend security

Let's discuss security matters that relates to the frontend. When I talk about the frontend, I refer with the website that the user is going to interacts with, and how this website is interacting with our backend.

Cross-site scripting (XSS Attacks)

Cross-Site Scripting (XSS) attacks are a type of security vulnerability that allow attackers to inject malicious code into a web page viewed by other users. The malicious code, usually in the form of a script, is executed by the browser, allowing the attacker to steal sensitive information, such as user login credentials, or perform other malicious actions, such as redirecting the user to a phishing site.

There are two main types of XSS attacks:

- Stored XSS: where the attacker injects malicious code into a web page that is then stored on the server and served to all users who visit the page.
- Reflected XSS: where the attacker injects malicious code into a web page by tricking a user into clicking on a link or submitting a form that contains the code. The code is then reflected back to the user's browser and executed.

To prevent XSS attacks, you can:

- Validate input: Validate all user input to make sure that it does not contain any malicious code. Use a whitelist approach to only allow known good input, rather than trying to identify bad input. This should be applied where it makes sense, we don't want to invalidate Javascript code in our Javascript playgrounds.
- Encode output: Any user input that is displayed on a web page must be properly encoded to prevent the browser from interpreting it as code.
- Use content security policy (CSP): CSP is a security feature that helps prevent XSS attacks by specifying which sources of content are allowed to be loaded by the browser.
- Use `HttpOnly` cookies. This prevents any malicious Javascript from accessing the cookies set by the web page.

Cross-site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) attacks tricks the user into performing an unintended action, such as transferring funds or changing their password, by tricking the user into clicking on a malicious link or submitting a form. The attacker takes advantage of the user's authenticated session with the web application to perform the action on the user's behalf.

Here are the main ways to prevent CSRF attacks:

- Use of anti-CSRF tokens: This is the most common method to prevent CSRF attacks. The server generates a unique token and sends it to the client as a hidden field in a form (hardly applies to us since we are using NextJS) or as a cookie. The token must then be included in any subsequent

requests that modify data on the server.

- Same-site cookies: This is a browser-based security feature that prevents a web application from sending a user's cookies to a different site. This helps prevent an attacker from tricking the user into visiting a malicious site that then sends a request to the web application using the user's cookies.
- Use of the `referrer` header: This method involves checking the `referrer` header on the server to ensure that the request was made from the same site as the web application.

Browser storage attacks

Browser storage attacks occur when an attacker is able to exploit a vulnerability in the way that a web application stores data in the browser to gain access to sensitive data. The most common types of browser storage are cookies, local storage, session storage, and IndexedDB.

The main way to secure the data you store on the end user's user agent is to encrypt the data. This prevents an attacker from being able to read the data if they are able to access it. You should also regularly audit your code by reviewing it to make sure that data is stored and accessed securely and that sensitive data is not stored unnecessarily.

Man-in-the-middle attacks

Man-in-the-middle attacks are a category of attacks that can be performed to the data while it is transiting over the network.

The most commons attacks are:

- Packet sniffing: an attacker listens to the packets transiting on the network, allowing him to steal credentials or perform replays attacks.
- Malicious code injection: an attack inject a malicious script to the website that is being visited, or can send its own phishing site altogether.

To prevent this kind of attacks, you should:

- Use an encrypted communication protocol (HTTPS), making it much more difficult to intercept and read the data in transit.
- Enable HTTP Strict Transport Security (HSTS): this technique allows a website to tell the browser that it should only be accessed over HTTPS and not HTTP. This prevents an attacker from intercepting a HTTPS connection and downgrading it to HTTP.

Backend security

Securing your backend is very important, since this is where all sensitive data is handled, and the business logic is executed. Finding a breach in your backend paves the way for an attacker to gain unauthorized access to other users data, for example. In this chapter, we are going to look at the usual vector of attacks and how you can prevent them.

Broken access control / broken authentication

Some attacks rely on bypassing or exploiting weaknesses in the authorization process to gain unauthorized access to resources or perform actions that they should not be able to. This can be done by manipulating the authentication process or exploiting vulnerabilities in the application's code. In our case, we are running a Role-Based Access Policies (RBAC) authorization framework.

Authentication and authorization is a very important aspect of security. You should use robust and well-maintained tools to handle that for you, unless you can have a team with the resources necessary to audit and attack your in-house authentication and authorization engine. You should obey to the least privilege principle, meaning that you give the bare-minimum authorization a user should have access to.

Broken access control is ranked #1 on OWASP's top ten web application security risks.

Cryptographic failures

Cryptographic failure refers to the situation when the cryptographic mechanisms used to secure a system fail to provide the level of security that was intended. This can happen due to various reasons such as weak encryption algorithms, poor key management, or implementation errors.

There are several types of cryptographic failures, including:

- Weak encryption: This occurs when the encryption algorithm used is weak and can be easily broken by an attacker.
- Poor key management: This occurs when the keys used to encrypt and decrypt data are not properly protected and can be easily obtained by an attacker.
- Implementation errors: This occurs when the cryptographic mechanisms are implemented incorrectly, leading to vulnerabilities that can be exploited by an attacker.

To prevent cryptographic failures, it's important to follow these best practices:

- Use of strong encryption algorithms: Use encryption algorithms that have been thoroughly vetted and are considered to be strong, such as AES, RSA, and ECC.
- Use of good key management: Use key management systems that provide secure key storage, key rotation, and key destruction.
- Use of tested and well-vetted cryptographic libraries: Use cryptographic libraries that have been thoroughly vetted and are considered to be secure.
- Regularly review and update encryption algorithms and key management practices to ensure

that they are still considered secure.

Cryptographic failures are ranked #2 on OWASP's top ten web application security risks

Injection

Injection attacks are a type of security vulnerability that occur when an attacker is able to inject malicious code or data into a web application, in order to exploit vulnerabilities in the application's code or in the underlying system. These attacks can be used to steal sensitive information, such as login credentials, and can also be used to inject malware or perform other malicious actions.

There are several types of injection attacks, including:

- SQL injection: This type of attack involves injecting malicious SQL code into a web application to manipulate the database and steal sensitive information. This is by far the most popular type of injection.
- Command injection: This type of attack involves injecting malicious commands into a web application to execute arbitrary code on the underlying system.
- File inclusion: This type of attack involves injecting a malicious file path into a web application to include and execute arbitrary code on the underlying system.
- Header injection: This type of attack involves injecting malicious data into headers of HTTP requests

We are not vulnerable to any kind of these injections. This is due to our usage of specialized libraries (ORMs for database related payloads, which implement prepared statements or parametrized queries), that take care of securing user input.

Injection is ranked #3 on OWASP's top ten web application security risks.

Outdated and vulnerable software

Outdated and vulnerable software significantly increases security risks by making it easier for attackers to exploit known vulnerabilities. As software ages, new vulnerabilities are discovered and patches are released to address them. However, if a system is not kept up-to-date with the latest security patches, it may still be vulnerable to these known vulnerabilities. Additionally, older software may not have the same level of security features as newer versions, making it more vulnerable to attacks.

This both a concern in the backend and at the operational level. For the backend, you should:

- Keep libraries up-to-date: Regularly check for and apply the latest security patches to ensure that known vulnerabilities are addressed.
- Use an automated patch management system: Automated patch management systems can help ensure that all your dependencies are kept up-to-date with the latest security patches.
- Use a vulnerability scanner: Vulnerability scanners can help identify vulnerabilities in a system and can be used to generate reports that can be used to prioritize which vulnerabilities should be addressed first.

- Use libraries that are supported and actively maintained by the community to ensure that security patches are released in a timely manner.

Outdated and vulnerable software are ranked #6 on OWASP's top ten web application security risks.

Server-side request forgery (SSRF)

Server-side request forgery (SSRF) is a type of security vulnerability that occurs when an attacker is able to send a request to a server from a vulnerable application, in order to access resources or perform actions that the attacker should not be able to access. This type of attack is typically used to gain unauthorized access to internal systems, such as databases or other sensitive resources, by exploiting the trust relationship between the server and the vulnerable application. This is typically done by tricking a server into sending a request to an internal service.

To prevent SSRFs attacks, you should have an overall good and secure system that checks and validates user inputs and that is protected from injections. You can also make use of WAF (Web Application Firewall) to inspect and drop any requests that doesn't match any routes, parameters and HTTP Methods combination known to the system.

SSRF is ranked #10 on OWASP's top ten web application security risks.

Man-in-the-middle attacks

Just like with the frontend, you need to prevent any man-in-the-middle attacks within your backend architecture. You don't know where your containers are going to be ran, meaning it might go through the internet or untrusted networks to communicate. You need to secure your backchannel properly, as it will transport all kind of sensitive information about the system and the business logic. Every request that you make could be sent across a untrusted network. To prevent that, more specifically in a Kubernetes environment, you can:

- Use service meshes that provides mTLS support (such as Istio)
- Use a 3rd-party network plugin for kubernetes that encrypts traffic between nodes out of the box such as WeaveNet

All your traffic should be encrypted.

Input Validation

Input validation are not a kind of attack but more a general good practice that you always need to follow. It refers to the process of ensuring that all user input received by a web application is valid and safe to use. It is a security measure that helps to prevent malicious input from being used to exploit vulnerabilities in the application's code or in the underlying system. User input can't be trusted, and you need to treat it as such. Failing to treat properly user's input can lead to a variety of attacks, such as XSS, CSRF, SSRF or Injections of any kind.

You can validate user's input in multiple ways, depending on the context, such as checking for a valid range, correct date, password strength check and more. Validation should be done in the

backend, since you can't trust what's happening in the front-end. The user might try to send a hand-made request, or the user-agent might have been compromised. Frontend validation is great for the user, since it allows to have a immediate feedback if he mistakenly did something wrong. But backend user input validation is where you really need to focus on, because this is where the security measure actually is.

Operation security

The last level of security I want to talk about is at the operation layer. At this layer, we are going to talk about the way you manage your infrastructure, and not the code that is running within it. This is arguably the most important aspect of your security layer, since gaining access to a host machine usually translates to having full access to its resources and services, and usually allows the attacker to move laterally in your system.

Security misconfiguration

The first kind of security concerns that I want to talk about is general misconfiguration. This usually takes the form of weak passwords, default accounts, open ports, incorrect permissions or letting unnecessary services running. It can lead to unauthorized access, denial of service or privilege escalation, where the attacker gain even more access to your system.

You should follow security guidelines for every of your running applications and services, and be thorough when configuring a new application on your application. What I saw happening in my (short) experience, is that people want to get a service running to try it and see how it behaves. They get it to work, never change the configuration again, and it end up in the production environment, publicly exposed and with poor security practices. You should discourage this kind of behavior and be very weary of what you are deploying and installing in your system, and how you do it. You should also regularly review and audit your configurations, trying to catch any misconfiguration or door left open for an attacker to come in.

Security misconfigurations are ranked #5 on OWASP's top ten web application security risks.

Outdated and vulnerable software

We've already talked about this attack vector, but it takes a different shape when we talk about it at the operational layer. Whereas in the backend, we were mostly worried about dependencies, at the operation level, we are worried about software CVEs and unpatched vulnerabilities.

You are dependent on a lot of software to run your application, and each of them might have special access to your system. No system is totally safe, as is the software you use. This is why it is important to constantly update your software, from the operating layer to your service mesh, to ensure that no known vulnerabilities can be exploited by hackers. As an example, Wordpress is known for having a constant stream of critical CVEs, and it is of the uppermost importance install security patches as soon as they are up. However, the Wordpress community is also known to be quite slow to react to this kind of issues, and this is why a lot of Wordpress sites ends-up compromised.

The same principle applies to your Linux kernel and Kubernetes cluster for example, since they both have full-access on your machine, and if compromised, would lead to catastrophic security failures.

For your operation layer, you should:

- Keep all your software up-to-date: Regularly check for and apply the latest security patches to

ensure that known vulnerabilities are addressed.

- Use an automated patch management system: Automated patch management systems can help ensure that all your dependencies are kept up-to-date with the latest security patches.
- Use a vulnerability scanner: Vulnerability scanners can help identify vulnerabilities in a system and can be used to generate reports that can be used to prioritize which vulnerabilities should be addressed first.
- Use software that are supported and actively maintained by the community or their vendor to ensure that security patches are released in a timely manner.

Outdated and vulnerable software are ranked #6 on OWASP's top ten web application security risks.

Data and system integrity failures

Data integrity failure refers to the situation when the data stored or transmitted by a system is modified or destroyed in an unauthorized or unintended way, resulting in the loss of accuracy, completeness, or consistency of the data. System integrity failure refers to the situation when the system or its components (hardware, software, firmware) are altered or damaged in an unauthorized or unintended way, resulting in the loss of confidentiality, availability, and integrity of the system.

Both types of failures can happen due to various reasons such as data corruption, hacking, software bugs, malware or unauthorized access.

To prevent this type of failures, it is important to follow general security guidelines, but to also make sure that your data is durable by using relevant data replication and backups system in place.

Data and system integrity failures are ranked #8 on OWASP's top ten web application security risks.

Security logging and monitoring failures

Security logging and monitoring failures refer to the situation where the logging and monitoring mechanisms used to secure a system fail to provide the level of security that was intended. This can happen due to various reasons such as poor logging configuration, lack of monitoring or alerting, or lack of proper incident response plan.

There are several types of security logging and monitoring failures, including:

- Inadequate logging: This occurs when the system does not log enough data, or does not log the right data, to detect and respond to security incidents.
- Lack of monitoring: This occurs when the system is not monitored for suspicious activity, which can make it difficult to detect and respond to security incidents.
- Lack of alerting: This occurs when the system does not have the capability to alert security personnel when suspicious activity is detected.
- Lack of incident response plan: This occurs when the system does not have a plan in place for responding to security incidents.

To prevent security logging and monitoring failures, it's important to follow these best practices:

- Use of comprehensive logging: Use logging mechanisms that capture a comprehensive set of data, including user actions, system actions, and network activity.
- Use of real-time monitoring: Use monitoring mechanisms that can detect suspicious activity in real-time, such as using intrusion detection systems (IDS) or security information and event management (SIEM) systems.
- Use of alerting mechanisms: Use alerting tooling that can notify security personnel when suspicious activity is detected.
- Use of incident response plan: Use incident response plan that outlines steps to be taken in case of security incident.
- Regularly review and audit logs: Make sure that they are complete and accurate and address any issues that may arise.

Outdated and vulnerable software are ranked #9s on OWASP's top ten web application security risks.

With all these security concerns in mind, let's get back to Polycode.

Security and migrating Polycode to microservices

In this section, I want to take a closer look at how we can secure our microservice architecture in Polycode. With the security concerns and concepts I just talked about, we have the tools in our hand to build and design our system in a secure manner. I am mainly going to focus on the new aspects of security that we have to manage, caused to the shift towards a distributed system design: inter-microservice communication.

That might ring a bell, since this is what we focused on section 3. Indeed, we talked about communication protocols and the data layer we could put under our microservices to support our internal communication.

When securing inter-microservice communication, the main thing to look at is how our data is transmitted across the wire. To sum up what we learned in section 3 in terms of security: Kubernetes doesn't provide a way to communicate data securely natively in most of the distribution, and to ensure that data is encrypted we must use service meshes or support an encrypted protocol within the application layer. I settled on using Istio, a service mesh that provides a wide range of features, including mTLS support, which make sure that all the data transmitted is authenticated and encrypted.

We also saw that, by using the sidecar patterns that most services meshes use, the encryption is not truly end-to-end. However, I would argue that it is sufficient enough for our use case, since our proxies and applications are running within a single pod, which means that nothing is sent on the wire. Someone who has compromised the machine could sniff those clear-text packet, but if this was the case we would have much bigger security problems anyway (and the attacker could have more invasive a destructive way to interact with our system). We could use the proxyless features of Istio and gRPC, to ensure end-to-end encryption, but I don't think it is worth the hassle.

Here's a brief reminder of what an inter-microservice communication would look like :

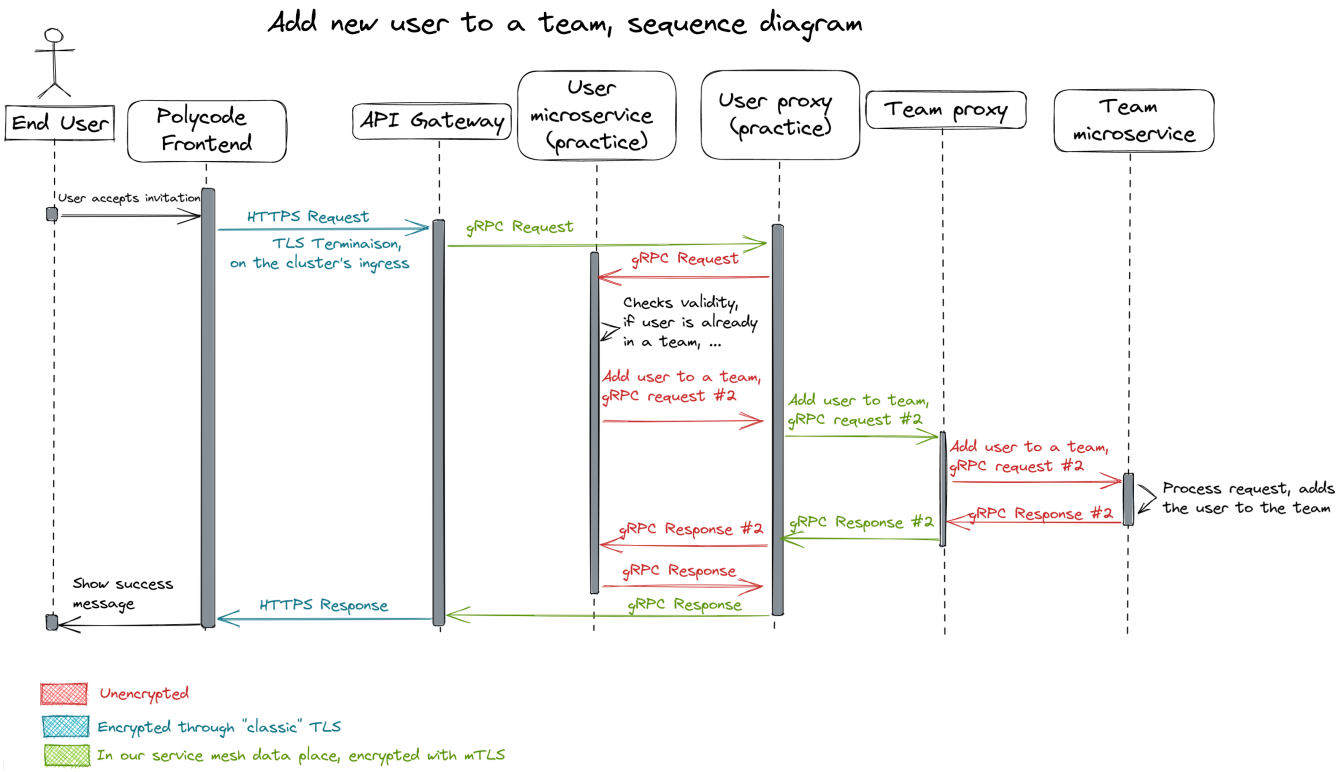


Figure 46. Reminder of section 3 Inter-microservice Sequence Diagram

You can find a proof of concept about this architecture here. This is the same proof of concept as the section 3.

So, are our inter-microservice communications already secure ? To an extent, yes. We rely on secure protocols and implementations.

Where we should be more cautious and more delicate, is the way we configure our services and applications running on our machines. Thankfully, the network the servers are ran on is very strict to the internet, allowing only 3 ports: 53/UDP, 80/TCP and 443/TCP. This means that we have a pretty low attack surface for outside attacks. But on the inside, some of our machines are running a lot of services, some are not kept up-to-date and some are badly configured.

I would recommend auditing each other's machine, trying to eliminate unnecessary services running on them. I would also push towards using more isolated environments for each of our applications. Most of them are running in containers, but I would argue that, to minimize the risk of poisoning and to prevent attackers from moving laterally in our system, to run our Polycode nodes on a separate VM, that would be running a security optimized, carefully monitored Security-Enhanced Linux.

I would also recommend auditing how our system handle user inputs, if we are vulnerable to CSRF, XSS or SSRF attacks. I am confident that the frameworks that we use are fitted with the necessary protections, but auditing the system would be a good way to identify this kind of attack vectors.

Conclusion

In this section, we have seen what security is, what are the biggest concerns to have and how we can make all of this relate to Polycode. The main takeaway I want every reader to have is that, security is not about occasionally auditing your system and forget about it afterward. It is a constant mindset to have, whatever the task you are doing. This is the same thing as with coding in a cloud-native manner. When you make your decisions, you need to be able to take into account the security risks associated with them. When implementing or installing something in your system, you need to make sure that you've taken all the steps to properly secure it. Security should be taken very seriously, much more seriously than what is done currently. We let is slide as a educational project, and if it were to get compromised, it would not matter too much. But this is a very dangerous mindset to have, and we must not make an habit out of this project, and instead start thinking with this security mindset now.

UI Integration

Introduction

Throughout this whole paper, I have been talking a lot about the migration of Polycode from a monolithic backend towards a microservice architecture. We have solved a lot of problems related to this migration, and have resolved a lot concerns we had prior to this migration. However, this shifts in mentality and paradigm in the backend raises multiple questions about how we can keep a UI that is closely integrated with the backend, in a way that is transparent for the end-user ?

In this section, I'm going to talk about the side-effects this migration might have on the web UI, and more importantly, if we should shift our way of thinking and developing the UI into something that fits better our new architecture.

How does the UI relates to the backend ?

An important question that we must ask ourselves, is how does the frontend interacts with the backend. After all, your web UI is where all your hard work that you put in your backend microservices are going to be used, and a poorly integrated UI will spoil everything. The frontend of your application is where the end users journey begin. It should be seen as an integral part of your system.

Currently in Polycode, both the backend and the frontend are developed in a monolithic manner. This is what we thought would be the organization fits the project the best, given that we are a small team and could coordinate pretty easily. At the beginning, we separated ourselves into 3 smaller teams: the backend team, the frontend team and the operational team. We realized during the first few months of the project that is was not the best organization to have, since each team was not implicated well-enough in the overall end-product, and we lost the goal and instead focused on technical decisions.

This is where we switched towards a more product-oriented approach, where we divided ourselves into 2 teams. However, instead of having each team responsible of a different stack, each team was responsible of a different products. One team would be responsible for implementing the candidates invite system for example, from the frontend to the backend. This has, for now, improved the overall code quality, and implication of the team, although it is too early to state that this is a definitive improvement. However, it makes much more sense to think like that, and allows for teams to focus on the end goal.

I think this is the direction we need to take going forward, and this is further exacerbated by the migration towards microservices, whose one of the main benefits is to allow a fine division of responsibilities, where small team can be fully responsible of an element in your domain and contexts. This is especially fitting in larger project, and Polycode is starting to become one. Now, one team can be responsible to developing a whole product, owning their own microservice, their own database, and their own frontend.. Or not quite yet.

Indeed, we saw this shift in paradigm in how we handle our backend development, operation management and project organization. The last piece of this puzzle that has not yet evolved to match this new way of organizing our development is the frontend. Actually, a team owns a microservice, owns its database, but share the frontend. This will lead to inevitable collisions, and we need to find a way to either work around this problem or completely eliminate it.

I hope that it all makes sense to you on an organizational standpoint, but it might still not be clear how this will improve the end-user's experience.

The end user experience

There are several ways this will improve the end-user experience.

First off, let's talk about the feature release schedule. By eliminating coordination and collision concerns between teams, each team can fully focus on developing its part of the application. This will improve the speed at which each team can develop their features, and reduce the time spent on fixing bugs and integrations concerns with the other part of the system. By speeding up this process, the end-user can expect to have features developed more efficiently, and by extension, have new features released more often.

Secondly, as mentioned above, developing the frontend using the same paradigms as with microservices will lead to fewer bugs, since the applications can be developed in a isolated manner, not worrying about other parts of the system. This will free up more time for the developers to work on other things, since they won't have to chase side-effects of pieces of code that was written by another team 6 months ago, but also generally reduce the number of bugs that slip through the crack and end up on the production environment.

Polycode

This all looks good on paper, even though we haven't considered the drawbacks yet. But I want to take a moment and stop to think wether or not it really makes sense to shifts towards a more microservice-oriented way to compose our frontend.

Our current frontend stack is a NextJS application. The frontend is rapidly growing in size, and we have gained a significant technical debt due some poor decisions we made. However, this is slowly being addressed by some members of the team to try to scoop off some of this debt. I would still think that a complete overhaul of the underlying way we manage our frontend would be beneficial, but we can keep most of our components as is.

It is still totally workable, and we must keep in mind that there are usually not more than two features being implemented in the frontend at the same time, something that will not move in time, since the team can only shrink in size. We have the problems of side-effects and integrating correctly with other parts of the frontend, but this is not a big problem at the moment.

Another key point is that, with the page-based routing of NextJS, we actually a basic way to divide our application into multiple parts. We sort of already have the possibility to work the way described above, considering the size of the team and of the codebase. I would argue that we have bigger concerns today.

However, it is interesting to explore the solutions we can adopt, that would allows us to develop in a way that totally decouples each UI Parts, effectively thinking of the frontend the same way we now think about the backend, using a microservice architecture.

Micro frontends

An emerging pattern that is currently trying to answer the same problems that the microservice architecture, but for frontend application, is the micro frontends architecture. As the name implies, it revolves around the same concept as microservices: divide your frontend code into smaller, independent and decoupled applications. There are multiple ways to achieve this goal, I will try to dive into each of them, but they all try to solve the same problem: allow team to work simultaneously on a large and complex product, by breaking them into much more manageable pieces of code.

Let's dive into the main way to achieve this, starting with server-side template composition.

Server-side template composition

Server-side template composition refers the process of combining multiple micro-frontends into a single page on the server side before sending the final HTML to the browser. This allows for the integration of different micro-frontends into a cohesive user interface, while still allowing each micro-frontend to be developed, deployed, and scaled independently. This approach can also improve the performance and SEO of the application by reducing the amount of JavaScript needed to be loaded on the client side. This approach tries to mix the server side rendering (SSR) approach of meta-framework like NextJS, and the micro-frontend architecture. This achieves similar benefits of server side rendering.

Unlike with classic SSR, this is done by having a "container" server, which is the server that a user-agent will hit when requesting a website. This container server will then compose the page by fetching each necessary piece of the HTML template to the responsible micro frontend server. This approach focus on reducing the payload size sent to the client, by rendering the page on the server, but also significantly improves SEO (Search engine optimization, how well your site is ranked by search engines), since search engines crawlers still have limited ways to execute Javascript, which is why SPAs are usually not well ranked by search engines.

This approach totally decouples each of your micro-frontends, but sharing data between your micro-frontends becomes a very difficult task. You might argue that with a micro-frontend approach, this should not happen, but in reality there is always a need for sharing some kind of data or message from one micro frontend to another, for example the logged in user.

This approach allows for nested micro-frontend servers, where the container server would fetch data from another server, which itself fetches data from another server. This is great when breaking down features, but can increase the time needed to render a page. You need to carefully design a robust caching system when using this pattern, else the time to first byte to the end-user can quickly get out of hand.

Run-time integration

Runtime integration in a micro-frontend context refers to the process of combining multiple micro-frontends into a single page at runtime, in the browser, rather than on the server-side. This allows for a more dynamic and interactive user interface, as the different micro-frontends can

communicate with each other and update in real-time.

You can integrate your micro-frontend in a myriad of ways, but we will focus on two of them in this chapter: Javascript bundles, and Web components.

However, runtime integration also has some limitations, such as increased complexity in managing the client-side code, and potential issues with SEO and performance if not implemented correctly. It's also worth noting that runtime integration requires a more powerful client-side, while server-side integration is more SEO friendly and can work with less powerful clients.

Javascript bundles

With this approach, each micro-frontend is loaded as a separate JavaScript bundle and runs independently in the browser. The micro-frontends communicate with each other through a shared event bus or a message-passing mechanism. The micro-frontends can also be loaded and unloaded dynamically, depending on the user's interactions with the application.

There should be a container Javascript runtime, which will be responsible for fetching the correct Javascript bundle when needed. This allows for islands of interactivity, and the loading and unloading of bundles depending on the user interactions with the web page. This is a similar approach as the island patterns that frameworks like Astro use.

This approach has several benefits:

- Dynamic updates: Micro-frontends can communicate with each other and update the user interface in real-time, providing a more dynamic and interactive user experience.
- Flexibility: Micro-frontends can be developed, deployed and scaled independently, providing more flexibility for teams working on different parts of the application.
- Improved performance: Since the majority of the code is loaded on demand, only the necessary micro-frontends are loaded, reducing the initial load time and improving the performance of the application.

However, this approach is the heaviest on the client, since it requires executing a lot of Javascript. If you need to have a good SEO, this might not be the right approach either, since a lot of additional work will be required to have an initial HTML placeholder, while the Javascript downloads and execute. Moreover, you need to carefully manage your dependencies. If each of your micro frontends depends on React to function, you don't want to download this dependency on the browser everytime a micro-frontends loads.

Web Components

Web components are a set of technologies that allow developers to create custom, reusable elements for use in HTML pages. These custom elements can be treated just like any other HTML element, and can be used to encapsulate the functionality of a micro-frontend.

When using web components for runtime integration, each micro-frontend is implemented as a custom element. These custom elements can be loaded and unloaded dynamically, allowing for a more flexible and dynamic user interface. The micro-frontends can also communicate with each other through a shared event bus or a message-passing mechanism, allowing for real-time updates

to the user interface.

- Reusability: Web components are reusable and can be used across different parts of the application, reducing code duplication and making it easier to maintain the application.
- Isolation: Web components provide a high level of isolation between the different micro-frontends, ensuring that changes to one micro-frontend do not affect the others.
- Interoperability: Web components are built on standard web technologies and are compatible with other web technologies and frameworks, making it easy to integrate them into existing applications.
- Browser support: Web components are supported by modern browsers, so this approach can be used in most web environments.

However, web components do require a more powerful client-side and may have a steeper learning curve for developers not familiar with the technology. Additionally, web components have a slightly different way of handling browser events and styling compared to traditional HTML, and may require additional setup and configuration.

Conclusion

Micro-frontend are a growing force in the web development industry, allowing for organizations to adopt a product-oriented approach for their team for the entirety of the stack, including the frontend, backend and operation layer. They bring similar advantages and drawbacks as with a microservice architecture, meaning that most teams and organizations that had the time to experiment with a microservice architecture probably found their way to gain as much value from them, while mitigating the drawbacks. They might use this knowledge to migrate towards a micro-frontend architecture, if they find their frontend to be too hard to maintain and to work on collaboratively.

However, just like with the microservice architecture, it is important to recognize that it is not a one-fit-all solution, and to adopting them means that you have carefully considered all the other options. If you don’t have the necessity to switch to a micro-frontend architecture, I would advise against it. This will just add the burden of maintaining this kind of architecture without benefiting from it, which is obviously a bad idea.

I would argue that this is the case for Polycode. Our current organization allows us to proceed in a product-oriented manner, even in the frontend, although it is still a "monolithic frontend". However, we don’t have enough teams and members working on the project to really gain from micro-frontend. I would suggest continuing handling the UI of Polycode the way we do it currently.

Polycode TAD: Conclusion

In this paper, we've looked at the most important aspects of microservices, and most of the requirements and constraints that comes with them. From designing our microservice architecture, to how to operate it, monitor and secure it, while talking about data architectures but also specific aspects for Polycode, like mobile application and runner architecture.

I hope that, throughout this paper, I have provided a great starting point for further deepening of each of the aspects I've touched on, as well as some of the tools needed to reflect on the decisions made with architecture that you are dealing with daily.

To me, this paper opens up insightful discussion with the Polycode team about why and how we should evolve the project over time. Some of the suggestions I've made through this paper looks quite obvious to me, some are more opinionated and can be debated. Either way, I'm looking for feedback from the whole team about the decisions I've made, and I'm also looking forwards to exploring others' solutions that considered aspects and caveats that I was not able to identify or to explore.

As a way to close this paper, I want to sum up the most important points that were discussed.

Summary

Before diving into the world of microservices, we started by identifying the Polycode domain, dividing it in multiple bounded contexts, obeying to the Domain Driven Design pattern. I've identified 4 main contexts, and with a careful and iterative process of architecture design, I ended up suggesting the following architecture:

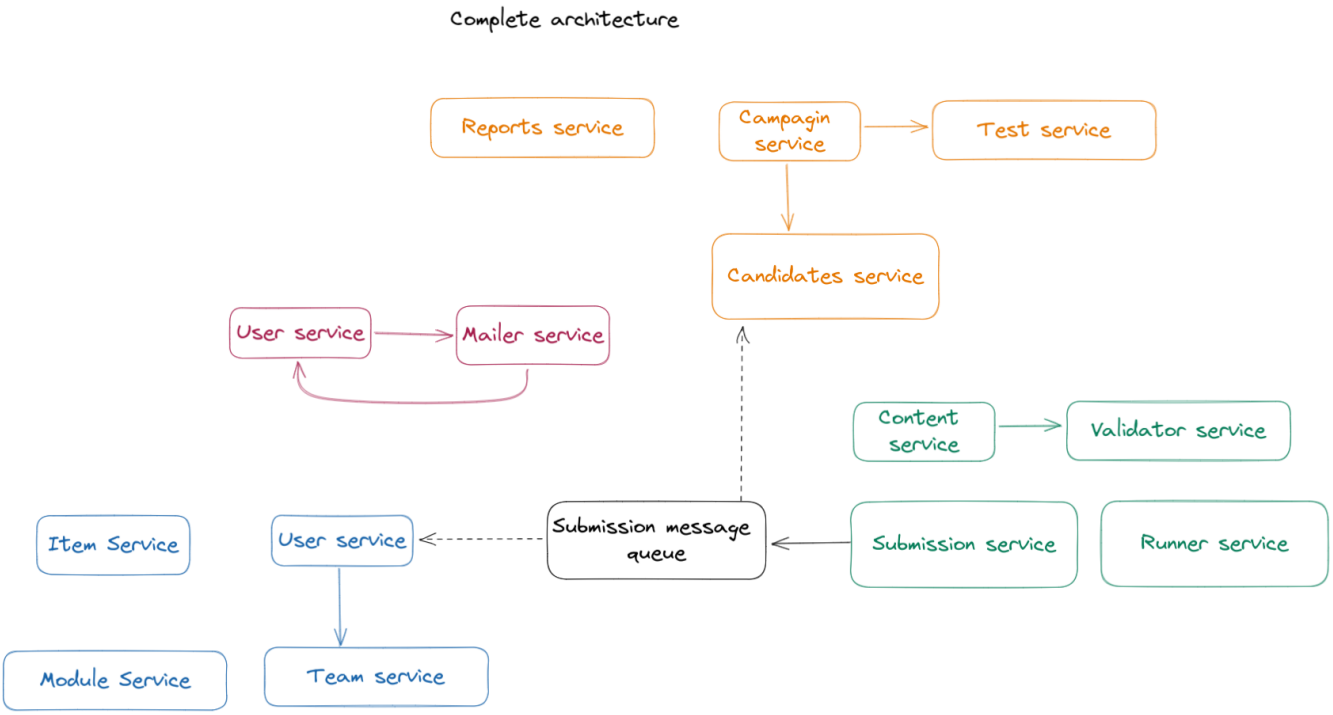


Figure 47. Section 1: Microservice architecture suggestion

Having this architecture in mind, we went forward with our next concern: how to handle authentication and authorization. After explaining the basics, I went deeper in the OIDC standard, since the IdP of choice would be keycloak. I've then showed you how I would handle our authentication flow, providing a proof of concept that used the mentioned techniques.

As stated in the conclusion of this section, an important point to remember, is that our future keycloak deployment should not be seen as a component of the Polycode architecture, but as a dependency of the Polycode architecture. Having this mindset highlights the fact that this keycloak instance can be used as a way to authenticate users across multiple of our applications, being Polycode or other projects.

We then dove deeper into our microservice architectures, by analyzing the options we had for inter-microservice communication. Since our services are now separated into their own application, we needed to find a way for them to communicate with each other, since we can't rely on simple functions calls anymore. I settled on gRPC, that filled this gap in a very convenient way for us, despite the drawbacks it comes with. An important tipping factor in this decision is the rigidity that it comes with, that will force the team into designing more refined API designs upfront, since we need to define protobufs for any communication to happen. This rigidity will help solve some of the problems we currently have.

I've suggested adding a service mesh, more specifically Istio, as an added data plane in our operation layer. Istio would take care of managing service discovery, encryption, circuit breakers,

retries and timeouts, as well as giving us more monitoring option and resiliency features. This means that we move all the infrastructure and network concerns into the operation layer, which makes total sense since it is not related to the business logic, which is what the application should focus on.

We've identified that, in a microservice architecture, having ways to discover and monitor your system efficiently is key to maintaining a good resiliency and catching problems before they get out of control. This is what we focused on the 4th section of this paper, where I suggested using a OTLP-compatible stack to monitor our infrastructure, since it is becoming the de-facto way to implement observability in all microservice architectures. I've chose the Elastic stack, but adapted to fit these requirements.

ElasticSearch was also brought up when talking about how to integrate a search engine into Polycode, where I identified that it would be a fitting solution to this problem. However, I've later explained that, for our needs and scale, using the integrated MongoDB full-text search functionalities would be sufficient for our needs.

We then took a deep dive into the runner architecture, by making a complete overhaul of the system, that were compatible with the constraints I highlighted beforehand. I chose a solution that would allow to optimize cost, while guaranteeing scalability, by having a controller/worker architecture, where the worker pool could grow by registering new machines. Controllers would then send requests to workers, prioritizing on-premise workers, and turning towards serverless offerings of cloud providers to handle surges in demand.

In the data architecture section, I've tackled the problem of storing data in a cloud-native manner, and how to organize the data stores that would fit the microservice architecture. I settled on the database-per-service pattern, since we structured our microservices in a way to be able to handle transactional consistency at the application layer. I've then described ways to secure performance and resiliency, as well as define a schema that pictures how the cluster and databases would be organized :

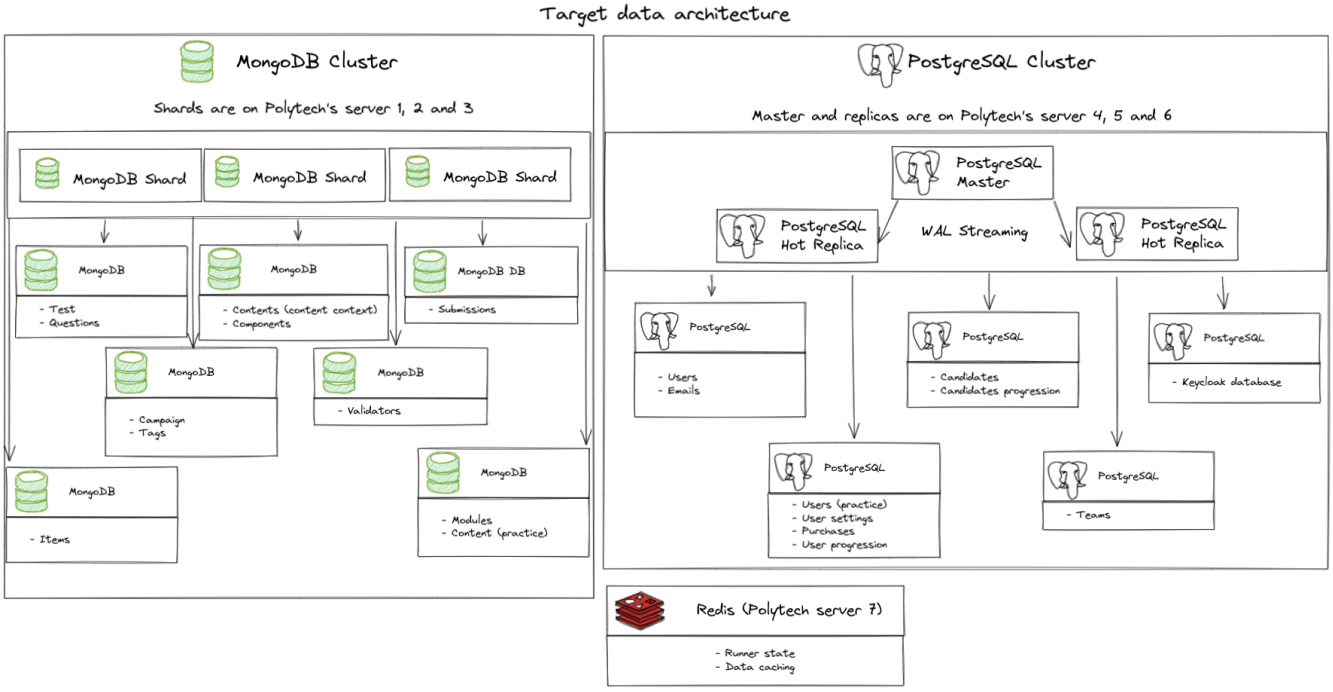


Figure 48. Section 7: Data target architecture

In the following section, we took an interesting dive into mobile applications, and what would make sense for Polycode. I've identified that there is no clear path for us to implement the desktop functionalities into a web application, and instead took the approach of defining new mobile-only features, that would fit the requirements needed for a mobile app: repetitive content and short in time.

We then turned towards security, and what are the concerns to have during our migration to microservices. I've started by highlighting the biggest security concerns a company should have, before identifying what we should do for Polycode. If you were to remember one thing for this section, it should be that security is a mindset that you need to have at every step of developing a product. Security should be an integral part of the development and deployment process for any web application.

And finally, we took a look at how to integrate our frontend with our microservices. I've explored some solutions out there, but mainly focused on the concepts and the need for this kind of solution. At the end of the day, migrating towards micro-frontends results in the same logic than migrating towards microservices. However, I hope I demonstrated that having a monolithic frontend application can be compatible with having a microservice architecture. You will need to adapt to some organizational behaviors that the micro-frontends makes mandatory, but with discipline and by extending the change of organization you adopted when switching to a microservice architecture, I would argue that it is totally feasible. I would suggest doing so for Polycode, while making sure this doesn't impact the user experience.

End note

I want to make clear that this paper is not without limitations and there is room for much further exploration. I am excited to receive feedback on this work and to continue the conversation around microservices. I encourage others to build upon this research and to explore new avenues. I am always open to discussion and collaboration, and I look forward to exchanging with the Polycode team and anybody willing to discuss this matter. I am going to continue my exploration in the microservice world, and looking forward for any occasion on participating on related projects.

Thank you for your time.