A tale of Slicing and Imagination

NICOLÁS PAEZ, Universidad Nacional de Tres de Febrero

This is a story of a team trying to incrementally replace a legacy application. This report describes my journey with the team while adopting practices to continuously deliver small increments of valuable features. The main focus is on slicing and feature toggles.

1. INTRODUCTION

My name is Nicolás Paez, I am a Software Engineer and Professor. I teach Software Engineering at the University and I also work on my own, helping teams to enhance their software delivery capabilities.

At the beginning of 2020 a bank contacted me to join a new team in charge of developing their new home banking web application. Their current home banking application had been developed in 2008 with a monolithic architecture without automated tests or appropriate documentation. The main drivers to replace the home banking application were usability issues and long lead times, that is the time taken from the business request to production implementation. The main challenge was to provide business value in the short term. Replacing the whole application could take more than a year, which was something unacceptable to the organization.

I have been involved in several projects with this kind of "legacy replacement challenge" and that is why I joined the team in the role of XP Coach. I knew that in order to be able to provide value in the short term we had to adopt an incremental and continuous delivery strategy. Even more, we would need to slice the features as small as possible and also make those slices to be incremental slices. At the same time, to be able to enable these incremental slices we would need to adopt some technical practices like feature toggles, trunk-based development, and continuous delivery.

2. BACKGROUND

Beyond the incremental replacement challenge, the organization had some other expectations regarding this project that could be summarized in three concerns: the team, the practices, and the infrastructure.

The organization expected the project to deliver a new application and a new team, that is, an outcome of the project should be a new team capable of supporting and evolving the new application. Initially the team was assembled with people from the organization and some external people, but the intention of the organization was that the external people would leave the project at some point and that the internal people take full control of the application on their own. The external people included frontend developers, a UX designer, a tester, and me in a role of XP Coach or Lead developer. The rest of the team included: backend developers, a Scrum Master, and a Product Owner—all employees of the bank.

The organization was already familiar with Agile practices, but mainly with the "management/collaboration practices" proposed by Scrum, that is: dailies, planning, reviews, and retrospectives. In fact, the team already had a person in the role of Scrum Master who had been working on these practices for some time. Some of the developers were familiar with XP practices like TDD and Continuous Integration, but only in theory. Developers had taken some courses related to these practices; but they were not used to using them in their daily work because of the complexities of their codebase and their lack of experience. So in most cases I didn't have to convince them to use these practices but I had to guide them in how to put them in practice.

Regarding the infrastructure, the organization had recently decided to move to Kubernetes as their runtime and GitLab as their CI/CD infrastructure, so this new application had to use these technologies. Most of the team members were not familiar with these technologies, but I was.

So, to summarize the situation we could say that the goal was to deliver a new version of the home banking to replace the legacy application, using new technologies with a new team, using new practices. All this in an

Nicolás Paez, Universidad Nacional de Tre de Febrero, email: nicopaez@computer.org

incremental and sustainable way. Even when these kinds of challenges were not new to me, each organization can be a totally different world, so I knew that this was not going to be an easy journey.

3. THE STORY

The incremental replacement of the legacy application brought up two important questions: how to choose/define the increments? and how to deliver/integrate those increments so they can live side-by-side with the legacy system?

3.1 The Planning Process

Regarding how to define the increments the main thing we should keep in mind is that any increment should represent value from the business perspective. In the agile world we typically represent these valuable increments with User Stories. Each User Story is a "vertical slice" of functionality. "Vertical" means that typically a User Story includes user interface and backend logic, or in a more general sense it includes components in different architectural layers. The alternative to vertical slicing is horizontal slicing, where each slice represents an architectural layer, that is: frontend/presentation, backend, database, etc.

In our case, our team was composed of frontend (angular) developers and backend (c#) developers. So the "horizontal slicing" sounded reasonable for most of our team members and also from a task-division perspective but not from the user/business point of view. So my first move when we started the project was to propose to the team to switch our perspective and think (and build) our backlog in terms of User Stories that represented vertical slices. Initially this idea generated some debate mainly with developers because some of them were not familiar with the whole stack and they could not be able to complete the whole item on their own. However, the Product Owner loved this idea and encouraged by the Scrum Master we decided to give it a try. To support the development of vertical slices in a smooth way I knew that beyond the area of specialization (frontend/ backend) of each team member, everyone should develop all the required skills to complete a User Story (vertical slice) from end-to-end. This is usually called "T-Shaped" people: you have an area of specialization (the vertical pipe of the T) but you also have enough knowledge to do tasks outside your area of expertise (the horizontal pipe of the T). Given that we were a new team with the intention of being T-Shaped people, I suggested to work in a Mob-Programming style. Curiously, the mobbing idea generated some debate among developers. They were worried about going slower, but there were no objections from the Product Owner (in some of my previous experiences the pairing/mobbing ideas had generated more resistance from the Product Owner/Managers than from developers).

In terms of the process we followed a typical Agile/Scrum process: 2-week iterations, starting with a planning meeting and ending with Review and Retrospective meetings, and a Daily Standup meeting.

Given that we were a new team, the first planning meetings were a bit chaotic, in particular because the different estimation approaches each team member was used to. By the third planning meeting we agreed on a simplified estimation/planning strategy. We agreed on using a relative estimation with 3 possible values: 1, 2 and 3. In case we suspected a User Story was bigger than 3, we split it. To split the stories we follow the heuristic proposed by Richard Lawrence and Peter Green [1]. This heuristic suggests a three-steps process: 1) Evaluate the story to confirm its size does not exceed the ⅙ of the team capacity, 2) Apply a set of patterns to split the story, 3) Evaluate the split.

Even after agreeing on the estimation strategy, the planning meetings were taking much more time than expected. In most cases this was because we detected missing information during the planning meetings. So to mitigate this situation we decided to add a "backlog refinement meeting" to be held some days prior to the planning meeting. The idea of this meeting was to prepare the candidate features and get all the required information ready for the planning meeting to flow smoothly and faster.

After a couple of months working together and delivering incremental slices we consolidated a slicing/planning strategy that could be described in the following way:

1. The business required some feature so the Product Owner started to collect information to design the feature from a functional point of view. This work used to include an analysis of existing systems/APIs that would provide the required backend logic/data. The Scrum Master and a UX designer would also collaborate here on the high level design of the feature. In some specific cases a developer was also part of this process.
2. In the refinement meeting the Product Owner presented the feature as it had been designed. At that point the feature was sliced. It is interesting to note that at this point the feature was not yet a User Story. The User Stories were the result of slicing the feature. For the slicing process we consider the

recommendations made by Gojko Adjic [2]. The slicing process could generate multiple versions of a User story, for example: typically the first version would represent a simplified "happy path" without any validation, then subsequent versions could include validations, notifications and some other improvements. We realized that this strategy generated dependent stories (i.e. to implement the validation you first need the happy path working), which can be controversial given that many people recommend having independent stories. So at the end of the refining meeting we had some candidate User Stories without any estimation because their acceptance criteria were not clear enough yet.

3. In the strategic planning meeting the candidate User Stories were reviewed and their acceptance criteria was agreed on. With this information the development team was able to make the forecast of Stories to be included in the iteration. For this, the developers did an informal (gut feeling) estimation without assigning size/points to the Stories.
4. Finally, in the tactical planning meeting the developers would break down Stories into tasks. Sometimes this process implied the partitioning some User Stories to ensure their size.

3.2 The Architecture

Our application was one of the channels that the bank provided to its individual customers (there was another application provided for enterprise customers). This was an interesting challenge because almost all features provided by our application depended on some other application/service. All the operations related to the customer accounts depended on the "core system" that was a legacy AS400 application that was accessible via an Enterprise Service Bus. At the same time all the operations/information related to credit cards depended on a set of microservices that was in active development at that time. So in some cases our features depended on features of other systems that were already developed. And in other cases our application depended on features of other systems/applications that were not completely developed yet. Figure 1 shows a high level view of the described architecture.

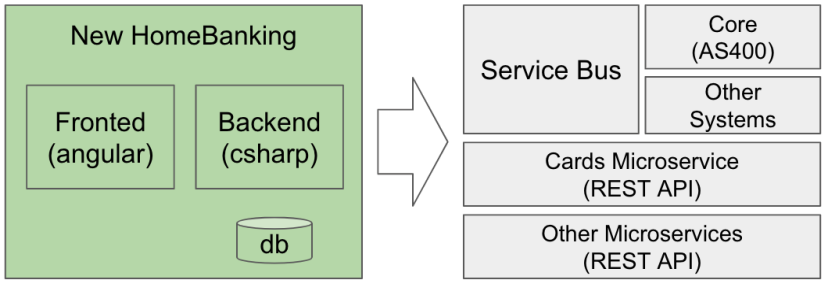

Figure 1. High Level Architecture

In the long-term the new home banking application should provide/cover several products/services provided by the bank including: accounts, credit cards, loans, and inversions among others. To avoid building a new monolith, the idea of the bank was to build the new home banking based on "micro-apps". The term micro-app has been used by some authors to refer to a different idea [3], but in this context each micro-app was a cohesive, independently deployable unit. Internally each micro-app was developed by a team and was composed of a frontend and backend created exclusively to serve the frontend, so it is sometimes called "backend for frontend." The idea of micro-frontends [4] is analogous to the idea of microservices but applied to the fronted—that is small, isolated and independently deployable units but with the additional challenge that the frontend is one conceptual unit from the user perspective. Figure 2 shows the long-term vision of the new home banking internal architecture. It is not the goal of this report to explain microservice and micro-frontends, there is enough information about these techniques on the web.

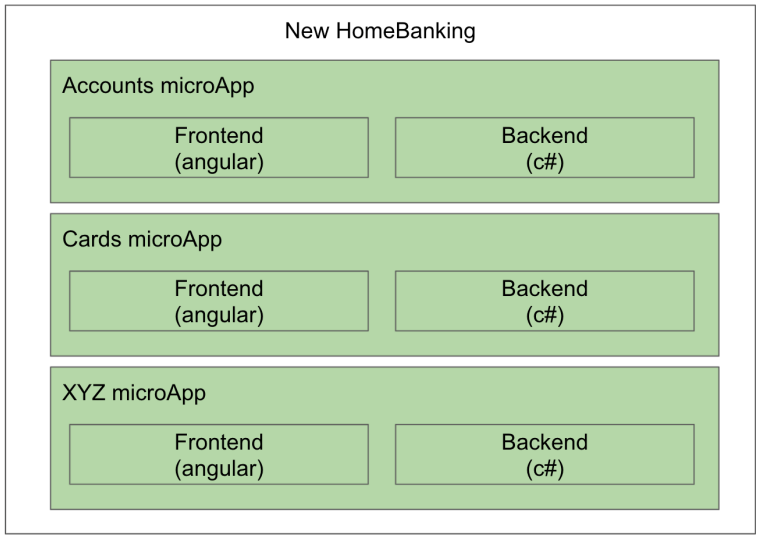

Figure 2. New home banking long-term internal architecture

This micro-apps strategy was a way of slicing the home banking application.

Given this long-term vision, we started working on the Accounts micro-app. After five months we split the team in two: one continued working on the accounts micro-app and the other started working on the cards micro-app. Shortly after that, a third team was assembled to work on the loans micro-app. The experience described in this report is focused on the development of the Accounts and Cards micro-apps.

3.3 Slices and Releases

After a month of working (2 iterations) we were able to deploy a first version of our application with some minimal functionality into the production environment. The goal of this first slice was to implement a walking skeleton guided by the development of a minimal feature that allows us to go through all the architectural layers, integrate with the authentication mechanism of the legacy system, and exercise the deployment pipeline along with the formal procedures to reach the production environment. From a functional point of view this first version included exclusively "read only features." It allowed the customers to query their balance. But even when this feature was already available in the legacy system in this case we implemented it following the usability regulations that the legacy system was ignoring. So even when this was a minimal feature, we provided some new value. We picked this strategy because we considered it a good balance between value, complexity and risk. Implementing a "write transaction" would be more complex and risky and would not add more value.

The next couple of features were also read only features: query of latest account movements, and credit card status. This latter feature included information coming from different APIs so we decided to take advantage of this situation to slice the feature. We created a first slice with the credit card "static information": card type, card number, and card due date. Then we created a second slice with the "dynamic information": current balance, closing date, and status.

The first "write" feature was "stop debit" that feature was not available in the legacy system. By default the payment of the credit card was automatic, that is, at the payment due date, the bank automatically withdrew the corresponding balance from the customer saving account. The customer can disable this behavior by executing the "Stop Debit" feature. This feature was far more complex than any other feature we had implemented till that moment. It included several validations, notifications, and edge cases. We sliced this feature in three.

The first slice provided basic functionality without early validations, generic error handling, and no notifications.

The second slice included specific error handling.

The third slice included early validations that would improve the user experience. Instead of letting the user execute the transaction and then getting an error in case of any business rule violation, we executed some early validations and blocked the execution of the operation in the case of detecting the operation would fail. It

is interesting to note that when we implemented this slice, we had to modify (and even remove) some error handling logic that had been added in the implementation of the second slice.

This situation of "removing code" felt a bit strange for some team members at the beginning. But after dealing with it a couple of times, it got so common that on some occasions (depending on the particular slices) we add an explicit task in the associated user story to remember to "clean up dead code" generated in previous slices. Not removing that dead code would add unnecessary complexity to the codebase. Even more important, we could argue that not removing that code would turn into technical debt.

In our sixth iteration and because of the COVID-19 pandemic we faced the development of a new feature with a very tight deadline. This feature as a whole was complex because of the different cases/validations depending on the situation of the customer using it. This feature would allow users to extract money from an ATM without using a card. This feature had a main flow and several alternative flows depending on the status of the customer. The short time we had to release this feature led us to be creative in the release plan. We discovered that most of the users (over 80%) would fit in the "simple customer flow." Implementing the feature specifically for this kind of users would be very straightforward. At the same time, for this group of users we could simplify the process by setting fixed values and eliminating the options. That is, instead of letting the customer select the from different options for the extraction, we would set a default fixed value. So this was our first slice of the feature. We used our feature toggling capability to enable the "extraction without card" feature only for those "simple flow customers." After the initial release of this feature we continued working on it doing weekly releases. Table 1 summarizes the slices we created for this feature.

Slice	Short name	Description
1	Simple flow	It was just one screen with default value, no chance to select the target account or the amount of extraction.
2	Simple flow enhanced	We added a confirmation screen and a basic error handling considering 2 specific possible errors.
3	Account selection	We added the possibility to select from which account to extract the money.
4	Email Notification	We added logic to send an email with the receipt of the operation.
5	Detailed error handling	We add logic and the corresponding UI elements to provide specific feedback in case of different errors.
6	Amount selection	We added the possibility to select the amount of money to extract.

Table 1. Extraction of money without card slices

An interesting detail about the slices of a feature is that even when we designed them in an independent way to be able to release them individually, we didn't always do that. Sometimes we had a slice ready to go live but the business decided not to do it and wait some time to release two slices together. In this situation we deployed the completed slice to the production environment but we kept it hidden by disabling the corresponding toggle. In the case of the extraction of money without a card, we implemented it with 6 slices that were released in 4 different releases. We implemented slices 1 to 4. After that, another feature took more priority than slices 5 and 6, so we delayed the implementation of these slices.

3.4 Audience and Feature Toggles

Feature toggles [4], also called feature flags, allow us to turn on and off specific application features. By using feature toggles we can decouple the deployment from the release of features. Deployment is just putting the code of a feature in the production environment. Once the feature is installed in the production environment, we can make it available for users (release it) or not—that will depend on the feature toggles configuration. In its simplest implementation feature toggles are context insensitive, that is: they can be on or off, no matter

what context. In more advanced implementations the toggles can be context-aware, that is: its value (on or off) will depend on the context (user, time, user location, etc.).

The feature toggles allow us to "slice" our audience, or in other words, to release our application incrementally to different groups of users. Initially we sliced our audience into 2 groups: 1) The team members, and 2) everyone else. This first toggling strategy was implemented using the login event as the toggle point, that is: just after the login we evaluate the toggle and based on that, we redirect the user to our application (feature enabled) or to the legacy application (feature disabled).

Shortly after that, we went further and divided our audience into 3 groups: 1) the team members, 2) friends and family, and 3) everyone else. This way we allowed some more people to access our application but at this point our application was not available to the general public yet—that was our next target.

To mitigate risks in case our new application had unexpected issues we decided to split the general public into two groups based on the device they used to access the application: desktop/laptop devices and mobile devices (in this context we are talking about customers using their mobile device to browse the web application which is totally different than using the mobile app.). This latter group represented around ~20 % of the users and for them, the new application would represent a great improvement because of the usability compliance of the new application (the legacy application was not responsive, but the new one was). So at this point our audience was divided into four groups: 1) team members, 2) friends and family, 3) mobile users, and 4) desktop/laptop users. Table 2 summarizes these audiences.

Audience	Release Order	Size
Team members	1	~10
Friends and Family	2	–300
Mobile users	3	100.000
Desktop users	4	500.000

Table 2. Audience details

Beyond this coarse-grained feature toggle that allowed us to define whether a user accessed our application or the legacy one, we implemented other more fine-grained feature toggles that controlled the access to specific and more granular features like fund transfers, credit card payment, and so on.

The initial implementation of feature toggles was home made, but shortly we replaced it with an open source library called FeatureManagement [6]. The toggles configuration was stored in the application configuration file, in case we needed to switch a toggle we needed to set the new configuration value and reload the application (this was a quick and safe operation given that we were running on Kubernetes).

An interesting point to mention is that storing the feature toggling configuration in a file may seem inappropriate for the long term because the configuration file could get too big and difficult to maintain but we should consider some relevant facts before coming to that conclusion. In the first place, even when many features are initially toggleable, in the long term all features are available to all users so the feature toggle is no longer needed. At that point we must remove the toggling configuration and also the toggling point in the code of our application to keep our codebase and configuration clean.

Figure 3 shows a snippet of feature toggling configuration that defines the following toggling rules:

- The "StopDebit" feature is enabled for everyone
- The "SupportBot" feature is disabled for everyone
- The "ExtractionWithoutCard" feature is enabled for 3 specific users
- The "FundsTransfer" feature is enabled for users belonging to 3 specific groups

```
"FeatureManagement": {
  "StopDebit": true,
  "SupportBot": false,
  "ExtractionWithoutCard": {
    "EnabledFor": [
      {
        "Name": "UserTargeting",
        "Parameters": {
          "Audience": {
            "Users": ["0800411222333","0800422333444","0800433444555"
            ]}}}]
  },
  "FundsTranfer": {
    "EnabledFor": [
      {
        "Name": "GroupTargeting",
        "Parameters": {
          "Groups": ["TeamMembers", "Friends", "MobileUsers"]
          }}]
  }}
```

Figure 3. Example of feature toggles configuration

Even though we never did this, we realized that our feature toggles infrastructure could allow us to easily implement A/B tests to experiment with different user experiences.

Once we had the feature toggles infrastructure in place, we discovered that we needed to know if a feature was "toggleable" or not before starting to code it because this had an impact on the implementation. So we decided to include this information in the description of each user story in our backlog management tool.

In retrospect we could say that our implementation of feature toggles had 3 stages.

- Stage 1: coarse-grained toggling at the login event, allowing users to access (or not) the new application
- Stage 2: fine-grained, home-made, very limited toggling support at the feature level
- Stage 3: fine-grained, library-based, robust toggling support at the feature level

4. LEARNINGS

I left the team after 10 months, long after what we had planned. The initial plan was to work with the team for 3 months, but the bank was happy with the results and I had the time available, so I stayed 7 months more. But, beyond the time I worked with the team, it was always clear that my participation in the project was temporary and that the goal of my participation was to develop delivery skills in the team.

I left the team with a gradual fade out. First I stopped coding but continued participating in the design, operations, and planning activities. Then, I gradually stopped participating in all other activities but continued attending the team ceremonies but in a "passive mode." Finally I said goodbye in December 2020.

In my opinion I left a consolidated team in both senses: collaboration and technical.

We, as a team, learned a lot of things in many different dimensions: collaboration, testing, and architecture. But in my opinion the most important things we learned were about planning and slicing. The combination of slicing techniques and technical practices (like feature toggling and continuous delivery) provided flexibility in the planning and delivery strategy.

We discovered that slicing can be applied at two different levels of abstraction.

At a high level it is possible to slice the architecture like we did with our micro-apps approach. Slicing at this level may have an impact in the organizational structure because it may require coordination of teams developing the different slices. At a low-level we sliced features, which is something that is completely in the hands of each team.

In both cases, slicing requires business support because slicing strategy may impact release dates.

At the same time, slicing brings a set of benefits that in my opinion are all consequences of short development and release cycles. Instead of building the complete feature, we can develop a slice, release it and get fast feedback from real users. Small releases are less risky and in most cases are easier to develop. Of course, this does not come for free. Some investment is required. Feature toggles and continuous delivery infrastructure are key practices for supporting a flexible slicing strategy with smooth releases.

The mob programming practice was extremely useful during the first couple of iterations. It allowed us to establish a solid and unified way of coding. Later in the project we identified some specific situations where pair programming or even solo programming were more appropriate. For example, the tasks related to fine-grained UI adjustments (css work) were more appropriate for solo programming because they didn't provide much new knowledge for the team while at the same time they required several cycles of "trial and error" to make the UI element looked/behaved according to what the team had agreed on with the Product Owner.

At the same time there were some things that didn't go according to my expectations. One of them was the T-Shaped people strategy. My sense was that most of the backend developers gained some front development skills, but I felt that frontend developers were reluctant to put hands on the backend code. My expectation was that all developers in the team would be able to take any story and develop it from end to end. But maybe the fact that this didn't go as I expected was not bad. I mean, the T-Shaped people strategy was not the objective; it was a way to avoid delays because of lack of enough people with a specific skill. But during my time with the team we never suffered that kind of delays.

As for slicing, at a certain point, the Product Owner and UX Designer were so happy with this strategy that they started to create slices on their own without involving any developer in the slicing activity. This didn't go well and we found ourselves in the planning meeting dealing with slices that:

1. Were not viable because of technical restrictions, or
2. Some slices that were so technically complex that it was easier and faster to implement the whole feature at once instead of the individual slices

For example, there was a feature that implied interacting with a service developed by another team. The Product Owner and UX designer sliced the feature into several slices, each of them adding new data/fields to the operation. From the UI perspective those slices were ok but from the data model that was not possible because the external service required all that data.. The key learning here was that you need at least one developer actively involved in any slicing activity.

Another important learning was that when slicing features into stories, the resulting stories usually build one onto another, establishing a certain level of dependencies between them forcing an implementation sequence. Of course it would be great to have totally independent stories and it is ok to work toward that ideal, but in practice that may not be possible. So the dilemma may be: either define big independent stories or small, less risky, not so independent and releasable stories.

This is the end of my report and you may be wondering, "Where is the imagination part mentioned in the title?" Well, maybe "imagination" was not exactly the best term to use. What I was trying to express is that slicing may require you to think a bit out-of-the-box and to use your imagination, which can be a challenge for those of us with an engineering background. But trust me, that thinking can have a wonderful impact on the ability to incrementally deliver business value.

5. ACKNOWLEGEMENTS

First of all I want to thank all the people that were part of the team, it was a great experience working together. I also want to thank Marcelo, the manager of the bank who invited me to join this project.

Rebecca Wirfs-Brock was my shepherd to write this experience; her feedback and guidance was very helpful to complete this report. Like we say in Argentina: ¡Gracias Totales!

REFERENCES

[1] http://www.humanizingwork.com/wp-content/uploads/2020/10/HW-Story-Splitting-Flowchart.pdf
[2] https://gojko.net/2012/01/23/splitting-user-stories-the-hamburger-method/
[3] https://blog.dreamfactory.com/what-is-a-micro-app-monitoring-an-emerging-trend
[4] https://martinfowler.com/articles/micro-frontends.html
[5] https://martinfowler.com/articles/feature-toggles.html
[6] https://github.com/microsoft/FeatureManagement-Dotnet