blueguava

MULTI-TENANT
DIGITAL PRODUCT DEVELOPMENT

Implementing SDK and Frontend Application Separately

Contents

Introduction

As demand for high-quality applications is at an all-time high, so is the pressure on software development companies to deliver those digital products as quickly as possible.

To keep pace with emerging, disruptive technologies, developers must always be at the top of their game—which, frankly, is quite taxing on them, especially if there is a time crunch hanging above their heads.

> We found multi-tenancy to be the most efficient concept for providing a balance between a short development cycle coupled with quick releases and maintaining the quality of the product.

Through multi-tenancy, a single software in a cloud environment can serve multiple customers (tenants) simultaneously, allowing them to use the same computing resources. In other words, they all share the same software application and a single database. Yet, despite the shared nature of use, customers are entirely unaware of each other; a single tenant's data is isolated and invisible to all the other tenants.

> Multi-tenancy also provided us the means for more affordable development costs and generally better use of resources. We maximized the use of computing power and other resources in case of a single instance between multiple tenants, which led to lower costs as well—imagine wasting the resources and dedicated infrastructure of an entire instance for every individual user.

Of course, the multi-tenant approach is not the be-all and end-all solution in software development. However, multi-tenancy was the logical choice for us and the purpose we had when developing this specific application, as you will see.

Still, developing a multi-tenant application is no small feat. What can a development team do to remain functional and organized throughout the product's SDLC?

We found one particular methodology to be highly effective for this purpose: The separation of development into two parts, a networking part to the backend API or other APIs and an app logic part using an SDK, as well as the frontend or UI part.

The concept sounds simple enough, yet it is hard to master. However, once this practice is integrated into a development team's operations, it can immensely improve progress, regardless of code complexity.

> The core idea of this methodology is to ease development and shorten the SDLC in a technology-independent way. We have used this particular approach in a previous project to develop a multi-tenant application that could potentially be used by millions of users worldwide. Fortunately, sticking to this strategy ensured the successful, on-time delivery of the product.

We hope that you will find plenty of utility as well as valuable practices in the following sections!

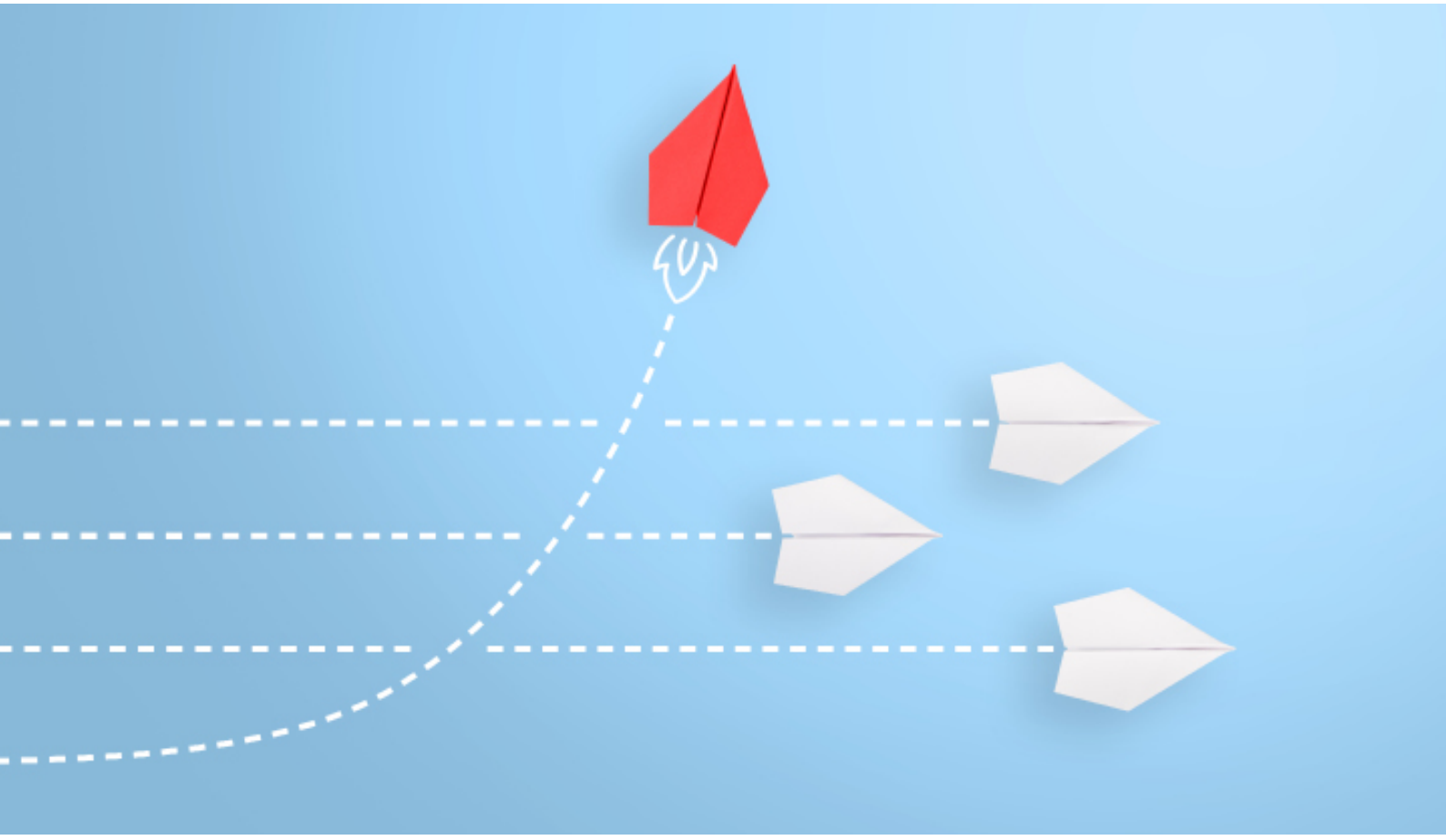
The Emerging Problem

Software development is a constantly shifting and evolving industry. The continuous, incremental advances in IT lead to incredible solutions, but managing to keep pace with these upgrades can be challenging.

Year after year, applications with only a single codebase tend to get more and more complex. As such, new feature development—or simply following and meeting new standards—becomes harder and harder to maintain, effectively turning into a never-ending chore for developers.

On the other hand, it's better not to consider changing the infrastructure or the backend behind these applications, as such feats either take enormous amounts of time or are virtually impossible.

Similarly, switching and building a new application using the old one as a foundation might not be as viable as you think. Unfortunately, existing application code and logic are rarely reusable for new projects, which means that the code cannot be packed and shared with other departments or organizations.



> On the other hand, multi-tenancy could allow developers to develop applications that simultaneously service vast amounts of customers at a fraction of a single-tenant app's development cost. But what is the best way to approach the development of a multi-tenant solution?

We've found it's best to use an SDK, complemented with the practice of separating development into logic and frontend parts. However, before we discuss why that is the best way to go for multi-tenancy, let's examine why multi-tenancy was our number one choice for developing a state-of-the-art application used by millions of customers worldwide.

WE DON'T HAVE A YEAR TO DEVELOP

Time constraints are always a hurdle in any development project. But frankly, high-quality digital product development takes a lot of time and resources—it usually doesn't pay off to push for earlier releases when the product just isn't ready.

> However, there are occasions when developers have no choice but to develop an app as quickly as possible. Still, writing an application from zero to a market-ready version is a process that consumes considerable amounts of time and money; in our experience, implementing a complex application takes a minimum of 6-12 months. Then again, that estimation does not even include the time and resources backend, infrastructure, and QA require—which are all integral parts necessary for the product's success.

We can see the problem clearly: We don't have much time or enough workforce to develop the same application for more than one platform.

Considering Multi-Tenancy

The reason for adopting a multi-tenant architecture for our application was based on several significant benefits that this approach quickly provides. These are the following:

COST REDUCTION

Multi-tenancy helps decrease the overall investment cost of the application in the long run. As mentioned earlier, the resources and databases are shared between tenants, leading to lower costs than the single-tenant architecture.

Users access the same application and database, and only those need to be maintained, making development and maintenance costs lower. Ultimately, it's cheaper and quicker to build a multi-tenant application at a reasonable price, as opposed to a single-tenant one.

Additionally, new tenants can be added at no extra cost either.

PERFORMANCE

Performance is perhaps the most important identifier of multi-tenancy. As shared resources and databases are disseminated between multiple tenants simultaneously, those tenants can enjoy the speed and responsivity with which the multi-tenant application handles their requests.

MAINTENANCE AND UPDATES

Due to code-sharing, no changes are happening in the data structure of a multi-tenant architecture. This means that the total cost of maintenance and updates is reduced since the same set of resources shared by the tenants must be updated.

ONBOARDING AND NEW TENANTS

Onboarding processes should always be as streamlined as possible. And that requires a design architecture that looks after the customers' needs while providing fast processing of their requests to sign up—or better yet, the application should allow users the option of self-sign-up. Multi-tenancy is an excellent choice for automating the sign-up and configuration of subdomain/domain processes. A multi-tenant application can implement the tasks of setting up default data for clients and configuring the app while also letting tenants configure it independently.

SCALABILITY

Compared with the single-tenant solution, a multi-tenancy vendor doesn't need to create a brand new, unique data center for every new tenant. Instead, tenants access one common infrastructure. Consequently, only that common infrastructure should be the target of upgrades. It will improve metrics across the whole application, offering more scalability to existing and new tenants.

RESOURCE USAGE

Suppose a tenant is simply not using up their share of allocated resources. In that case, the system will immediately optimize resource usage, redistributing a part of those resources for another tenant with higher resource consumption. This way, we can ensure that the use of all available resources is maximized, which reduces waste and optimizes costs.

Although the advantages are numerous and make the concept of developing a multi-tenant application both cost-effective and more efficient, there are a few drawbacks that developers and product owners alike should be aware of.

SECURITY RISKS AND COMPLIANCE ISSUES

Given the shared nature of a multi-tenant application, tenants who are not associated with our organization also use the same database. While it is almost impossible for them to see our data, this broader access to the same resources may reduce security control.

For instance, security issues or corrupted data from one tenant could spread to others using the same instance or machine. Fortunately, this is an infrequent problem, and if the infrastructure has been configured well, it should never occur under normal circumstances. Then again, configuring both the infrastructure and the security systems the right way are hefty investments in and of themselves.

Additionally, some regulatory requirements could potentially get in the way of storing data within shared infrastructure—regardless of how secure that infrastructure is.

MANAGEMENT/CUSTOMIZATION

Despite the added integration benefits, the ability to apply custom changes to a database is quite limited.

UPDATES AND CHANGES

Another conditional drawback, but if the application relies on integrations with SaaS products and one undergoes an update, some issues could ripple through all connecting applications.

As we highlighted in the introduction, multi-tenancy is not the be-all, end-all solution for every scenario. It can provide a series of significant advantages, but in the end, it is in no way the ultimate architecture that must be implemented at all costs.

THE "NOISY NEIGHBOR"

Typically, this should not cause issues or inconvenience users. Still, if one of the tenants uses an extreme amount of computing power, performance may significantly slow down for the other tenants. However, this is also quite unlikely to impede users, as most multi-tenant cloud environments are set up to prevent cases like this.

Why Did We Opt for Multi-Tenancy?

We have examined both the benefits and drawbacks of multi-tenancy. We can see that the positive outweighs the negative—especially if we make the necessary investments to ensure that infrastructure and security systems are configured correctly.

In our case, when we began developing an application that was to be used simultaneously by millions of customers worldwide, the choice was made for us: It had to be built with multi-tenancy. Otherwise, costs and resource usage would have gone through the roof.

On the other hand, we had another reason besides the multi-tenant approach's performance and cost optimization possibilities. It's no secret that we were developing this application for subscribing users who would consume several hours of video content almost every day. Consequently, the app itself had to contain a feature set that provided an excellent customer experience. Specifically, features such as content handling, customer management, subscription, and more were mandatory; however, the challenge was finding a way to create a multi-tenant use of these services.

ADVANTAGES

REUSABILITY

With a multi-tenant approach, reusability becomes a significant benefit that will ultimately save us a lot of time. Better yet, reusing code can also be applied to both the SDK and Application parts.

Reusability will provide us the following benefits:

SDK

Developers will no longer have to rewrite the entire business log part of an application as they start developing another platform. Instead, they only need to rewrite parts that are related to each platform's difference. However, that is best achieved if developers have a clear idea of what logic is implemented centrally in the SDK. One way to gain that clarity is to write tests, which serve two goals. It provides documentation (looking at tests gives the developer a clearer idea of what the tested part is supposed to do) and assures us that the examined portion offers the intended functionality.

Frontend Application

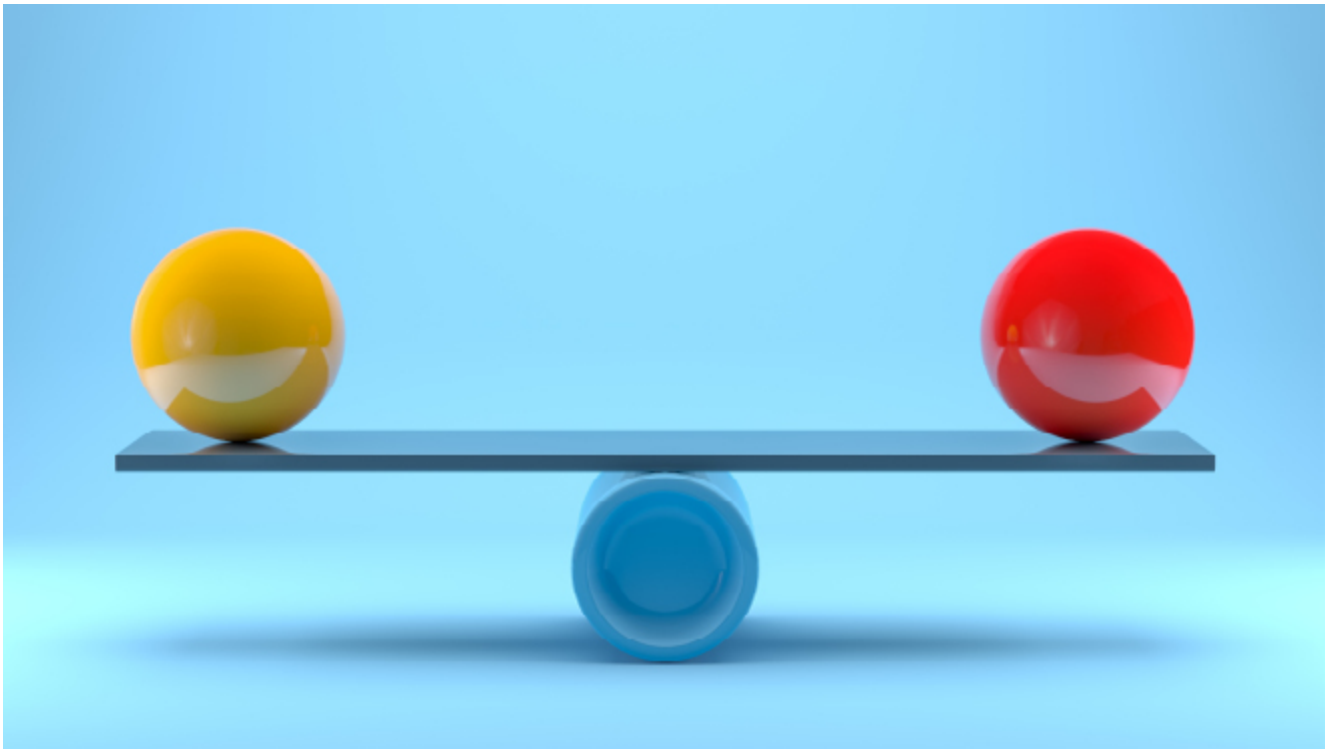
The same company will use the same UI for different platforms since it unifies their brand; everybody will know which app they are using and who the product owner company is. Although the UI should be changed somewhat based on different platforms, the code already being tested and used somewhere else will save a lot of time.

SAVING TIME AND EFFORT FOR QA

As time grows short and there's still a lot of development work left, it's usually QA that is not given enough time to test properly, forcing them into an ever-intensifying time crunch.

But with multi-tenant applications, we can also save them a lot of time, providing some breathing room to focus on testing the most critical aspects of the app. This is because QA does not need to rewrite the entire test documentation and all the test cases, thanks to the reusable code.

Testers who have tested one part of the application will already be aware of the weak points. They should be able to probe another aspect where the code was reused with an idea of pitfalls to look out for. Moreover, tests can be started much earlier into the SDLC. They don't even need to be separated into different phases; testing can become a continuous process that leads to faster feedback with less effort.



BACKEND/INFRASTRUCTURE AS A CODE

Another significant advantage is that we can now write well-balanced code that is much more accessible, leading to clearer and better functionalities in the multi-tenant approach.

Once we have well-balanced code on our hands, it becomes significantly easier and quicker to determine which parts should be implemented in the backend and which features should be implemented in the application part.

With well-balanced code enabled due to our multi-tenant development approach, we can also leverage microservices for creating a micro-frontend, which saves us a lot of money and time on the backend side.

Building a Multi-Tenant Application

This is where purpose—a multi-tenant application meant to service millions of customers consuming content in exchange for their subscriptions—meets methodology. We had to find an easily applicable method that would allow us to develop the application rapidly.

We decided that using an SDK would give us one of the best, most cost-effective solutions available in the market. However, as you will see in the following few sections, the SDK's lack of customizability is somewhat concerning. Given how vital the UI would be in an application meant to browse thousands of video content pieces, we needed a much more flexible solution to handle the frontend side of the application.

> What was our solution? We separated the application into two parts; a Software Development Kit, which contained the business logic (application logic and networking part to the backend API), and an App part where the UI and smaller logics were implemented (the frontend). The SDK can also be separated into more single modules that remain functional by themselves.

However, there is one essential rule to this solution. Minimal (preferably zero) logic should be implemented in the application part. As we experienced firsthand, dragging the logic into the app part overcomplicates development and leaves developers trying to figure out which functionality is implemented where and why. When implementing the separation this way, it is essential to carefully sort all functionalities into their respective parts before committing to writing the code on functionalities required centrally—that is, in the SDK.

> If we follow these steps, we will also have an easier time maintaining the system in functional changes. This is because each part can be managed separately if there is a need for improvements. In contrast, developing and maintaining a multi-module application is more challenging and would take more time and effort. However, despite its generally easier, "ready-to-go" nature, Framework/SDK development has its own set of difficulties developers need to be prepared for.

Another benefit of this approach is that we can modernize the application without touching the underlying business logic or the interfaces. Even if the code has to be rewritten or modernized, it can be done so quickly without changing its interface.

Moreover, the SDK has more uses than just building a single application. However, to get the most out of it, every part should be documented as thoroughly as possible.

SDK Advantages vs. Disadvantages

Ultimately, an SDK is excellent for providing high-level, complete solutions to a set of specific problems, without the need to be bothered with the implementation of explicit technologies.

> In essence, developers receive these tools after both design and testing have been completed and are proven to work as intended. This eases development considerably, as developers are able to focus their attention on making the application even better thanks to these preconfigured solutions.

For example, by applying SDK in development, we can avoid situations where three entire teams work on debugging for weeks while trying to solve the issue of content not updating in the application. Instead, the integrator simply receives an event from the SDK about the successful content update, and then it will only have to display it.

> On the other hand, the high-level nature of the SDK is what makes it less ideal when having to develop less generic solutions. If developers only need to provide a turnkey solution to one specific issue, then the SDK will support them in these tasks flawlessly. However, the more customized the solution is, built directly to serve multiple purposes, the more complicated development becomes, and the less likely it is that the SDK can cover that custom nature and high complexity.

In the table below, you can see some of the advantages and disadvantages of a good SDK in general.

ADVANTAGES	DISADVANTAGES
Ease of use (the whole team can work with it efficiently at the same time) Improved scalability	Lack of control leading to further challenges in the development process
Thorough documentation explaining how the code functions	Compromises in application experience
Provides enough functionalities to add value to other apps	Built-in restrictions and pre-made choices* such as: • Streaming model for data population from CMS • Image handling for dynamic controls such as lists • Prioritization of interface events and background events • Threading model or thread density for helper tasks
No negative impact on a mobile device's CPU, battery, or data consumption	UI stutter, placeholder images, etc. due to built-in choices and restrictions concerning image handling and threading, for example
Works well with other SDKs	Possible issues in a cross-platform environment where the user will interact with the application across multiple platform SDKs

** Developers hardwire these choices into the platform SDK to simplify the process of building applications.*

I NTEGRATING THE SDK

We have divided the project into two separate areas with different scopes and objectives. To see where we have to go from here, we must understand the exact differences between the two tasks. Reusability will provide us the following benefits:

Software Development Kit (SDK)

SDK connects to the backend and helps translate the APIs for the frontend application. We can make development easier by sticking to two essential design principles:

1. Have all functional logic on the SDK side of the project.
2. Make all functional modules as independent and interchangeable as possible.

For instance, the player and analytics features can be separate modules, whereas all 3rd party applications and integrated parts can become different modules

Frontend Application

This is the part that creates a visual and interactable connection with the user. In other words, all UI/UX-related features elements are here. Not only does it define how the app looks, but how it feels. This is where the user experience choices and implementations are made, for the most part, making it the user's layer for interacting with the product.

CHALLENGES WITH SDK

The number one objective in any product development project that separates the application into an SDK and a front-end app part is architecture. If the SDK is not architected as it should be, it will have significantly more problems than benefits.

For instance, implementing any functional/logic element outside the SDK—e.g., on the application side—will lead to more difficulties during later stages of development.

If anything, an exemplary SDK implementation should never be rushed. It might seem slow initially, and it certainly takes time to set it up properly, but it will bring back that time and more later. What's important is to be especially careful with maintenance as it can become significantly trickier and more complicated when several apps use the same SDK.

When using a platform SDK development approach, inherent challenges usually arise mid-project—up until that point, there should be no significant issues with core development tasks. However, as the existing application is signed off for design and business reviews, certain discrepancies will most likely be discovered between the original conceptual designs and the current application behavior. For instance, requests to eliminate UI stutter or populate a list with placeholders while the intended images are still loading may arise. Furthermore, differences in platform controls will also lead to altered behavior across platforms, making this another area where developers have little power to adjust the SDK's solution to the original design.

Unfortunately, after the standard application structures have been implemented according to the SDK's development paradigms, there's not much we can do from a development standpoint to significantly affect the issues that the business or design teams will potentially raise. In the end, this phase of the development comes down to a choice between either de-scoping certain features or a prolonged development to find a workaround within the SDK. If the business, design teams, or both, deem specific issues non-negotiable, developers are likely to face a lengthy development rework.

Nevertheless, thorough, up-to-date documentation can make a real difference when coping with such development issues.

How to Implement QA

FACING THE CHALLENGES OF DIFFERENT ENVIRONMENTS

WHY IS IT BETTER TO USE MULTIPLE ENVIRONMENTS?

An essential tip in software development is to provide developers with a sandbox to try everything out. But it is also important to give QA the same space to perform different phases for validation.

As we will see in the next section, various teams need different SDLC phases. In our organization, the product development and QA teams work in close cooperation, and we usually split the entire cycle into four stages: Development, Internal QA, Performance Test, and Final Validation. Notice the focus on testing and ensuring that the product will be released in its best possible state. It is an indispensable element required for the software's high-quality performance and success among customers.

Developers should perform tests, not in the local environment, but an independent one separated from all other departments during the development phase. Additionally, if there is an external (client-side) QA, the client must be provided with a stable environment that can function independently of the development environments. This way, if the developers make any changes, it will not affect the external environment, and the client can continue running tests uninterrupted.

Similarly, internal tests such as performance tests, load tests, regression tests, or the final validation itself shouldn't be interrupted either due to a code change. Giving these activities an independent, separate environment will solve this issue quickly.

ENVIRONMENT FOR PURPOSE

For the optimal development/QA cycles, we found that it is best to provide the following set of environments for each team:

Environment	User	Usage	Notes
DEV	Developers	Internal	Provide all rights and permissions to the developers—they need it to make changes on the fly
QA	Internal Testers/QA	Internal	Testing and checking the app before it is ready for presenting to the vendor or stakeholder
STG (Staging)	Vendor	External (client-side)	Final validation of the product for the vendor before pushing it to PROD
PROD (Production)	Customers	External (client-side)	The environment that the customers use

DO WE NEED DIFFERENT APPLICATIONS FOR ALL ENVIRONMENTS?

That will not be necessary at all, fortunately.

If the SDK side is handling it well, we can get the back-end data to change the environment via optimization. Of course, as we saw in earlier sections, this isn't the only advantage a good SDK can provide in this regard.

For instance, the application we were building required multiple environments for external use. It would be released in numerous countries across three continents, which meant we needed more than one STG and PROD environment. However, the SDK we used allowed us to change these environments on the applications side!

The key is to enable an environment selector logic in the SDK. It will make the SDLC—development and testing phases altogether—significantly more accessible, faster, and efficient.

ARE PRODUCTION ISSUES DONE FOR? – FURTHER CHALLENGES

The separation of the SDK and the frontend application can be a beneficial methodology in software development. However, it is better to be cautious when applying it to specific challenges.

For example, some of the previously mentioned environments use nearly identical configurations to the production environment. Consequently, the test results will be much more reliable than those we will experience after release in production. Naturally, these are not 100% efficient either, but they are closer to the real deal.

The challenges are rooted in several observable phenomena:

- User behavior on the application side cannot be predicted easily.
- Scaling and loading the backend part with no deterministic load can cause issues that can only be experienced in a specific production environment.
- Issues or crashes specific to different devices and platforms are also imperceptible during tests.

Some of these issues can be identified if there is a chance to test the application when the load is low; for example, when most users are asleep or in a smaller country or region where chances of receiving millions of queries every minute are low. Otherwise, it is also feasible to consider letting users beta-test the app.

Before Release – Optimizing the Application

PERFORMANCE

Today, virtually anyone can develop a relatively well-performing application that functions as intended. Scaling that up, quite a few companies can build good to extraordinary applications that are complex and much more prominent in scope. Even so, many software developer firms can create multi-tenant applications.

But only a few companies can build apps for as many platforms as needed, and virtually no company can build apps that function well on all platforms and with the same level of performance. Only a select few development companies can achieve something in the ballpark of that level of service.

We have made it into that prestigious club. Now we want to share what's needed to join it by developing great, multi-tenant apps that function on almost all available platforms with nearly the same level of high-quality performance.

ENVIRONMENTAL PROBLEM

The fundamental problem is relatively easy to grasp; different platforms will have additional hardware and software that the app should be compatible with. Some devices range from low-end to high-end (e.g., Android versions, Smart TVs, etc.).

Low-end devices can pose a whole new set of issues, some of which we have listed below (drawing from our own development experiences):

- Weak computing hardware (memory, processor).
- Weak connectivity (Wi-Fi).
- Old software/firmware which the vendor does not maintain.
- Unsupported video formats or DRM techniques.

Developers must consider reality, which is that developers and testers mostly use modern, high-tech devices in a controlled development environment with excellent connectivity. Most end-users will be using low-end or mid-range devices—and quite often not in the best network environments.

If a digital product is to succeed, this dilemma must be addressed head-on.

THE SOLUTION: RESEARCH, DECIDE, AND BE READY FOR THE WORST CONDITIONS

A great way to circumvent the high-end development/low-end reality dilemma is to optimize the code and ensure that the app is not developed purely for good "lab" conditions.

That, however, requires plenty of research. The best we can do is gather all available information on the bottlenecks (both hardware and software) of the supported devices, investigate libraries and third-party applications to find out whether our app will be compatible with all devices.

However, it is essential to reach a sound, well-researched decision on these issues before development commences. Product owners, developers, and other stakeholders should decide early on if creating a new app for all devices is worth the time and effort or whether it's better to just cut down on the number of supported devices and models.

Over GA Launch – Hotfixes

After the app has been pushed to production, we should prepare for the onslaught of never-before-seen issues that only emerge in live environments. Fortunately, with the SDK–frontend separation methodology, we can achieve better efficiency in hotfix management.

PRIORITY AND SEVERITY

It is most expedient only to release a hotfix when the issue priority and severity are both high. Hotfixes generally focus on getting rid of the most jarring issues and providing must-have fixes that guarantee stable performance and positively affect user experience. Therefore, it is counterproductive to include anything else, even nice-to-have fixes or features—those can wait until a giant patch. Furthermore, these hotfixes need only a single sanity test; any other test types are redundant in this case.

Also, keep in mind that issue and crash numbers will differ in the DEV, PROD, and STG environments.

Additionally, post-release, it is worth checking whether infrastructure costs have not increased after a hotfix affected the application in any way. While we are there, we should also examine it from the backend to ensure that certain response times have not become significantly longer.

RELEASE FREQUENCY

Either way, it is best if application releases are as frequent as possible. That conveys user-centricity and ensures that the app remains stable and performing according to standards, keeping overall application quality high.

We have found a high release frequency (either bi-weekly or monthly) to be most feasible; this way, the number of hotfixes can be kept at a low level, only pushing them to production when necessary.

THE GOAL

With hotfix management, our objectives are simple and straightforward. To summarize, here's a list of the four primary considerations that will keep the app fresh and high-quality:

1. Release a hotfix only if the issue priority/severity is high.
2. Perform thorough monitoring on both the SDK and the application side as well—it's easy to figure out when it is needed.
3. After each release, hotfixes are only necessary if the issues affect users significantly and spoil their experience.
4. Have a fallback mechanism for each application at the ready; if there is a significant production issue, we need to make sure the means for reverting to an older but stable and functioning version is readily available.

Afterword

“

For more than 15 years, we have been developing various desktop, web, mobile, TV, console, backend, and database applications that run in multi-tenant mode. We admit it is not always possible to build the application this way. But if you need to run an app in several different modes and systems with relatively little rewriting, we highly recommend this architecture.

Our projects were about creating brand-new products that can be used on multiple platforms. The company that contracted us wanted one product: **the application**. But why can’t it be two valuable products instantly?

By separating the product into SDK (business logic) and front-end (UI) parts, we can make development significantly easier and double the gains.

The SDK part can be used and sold as a separate product. Alternatively, it can be used to develop new applications for a different brand or company. All in all, these other apps can be showcased as excellent references when negotiating with a new client.

It may seem too complicated at first, but the complexity of a project and the testability of different conditions determine the level of difficulty this development process entails.

It is important to emphasize that this is not developing a **marketing app** or a RAD (Rapid Application Development) methodology. These are product development approaches for projects where product support and sustainability are required for years.

The product development team consists of developers and people with other interests, such as BA, BI, Ops, or different levels of QA, where various issues also matter in the overall readiness and testability parts.

That’s why we consider these separations and approaches essential in the development of a **successful product**.

”

Péter Dikházi – Founder and CEO of Blue Guava Technology

About Blue Guava

Our goal is to become the best long-term partner that any of our clients could wish for. With more than 10 years of state-of-the-art software development, streaming, and testing solutions, we have helped market-leader partners increase their revenue and the efficiency of their IT operations while cutting costs and time. Simultaneously, the software products we developed for them streamlined and optimized the streaming experience for millions of their customers across more than 50 countries on three continents.

At Blue Guava, we believe in exceptional customer service. Our passion is to provide our clients with nothing but the highest quality services that are guaranteed to meet their needs and help them in their quest to produce excellent software solutions.

Our content delivery, content management software solutions, and quality assurance services will help you maximize customer engagement, ultimately empowering your business's customer adoption and retention capabilities.

Contact Us