Reality Premedia Services

Whitepaper - EPiServer

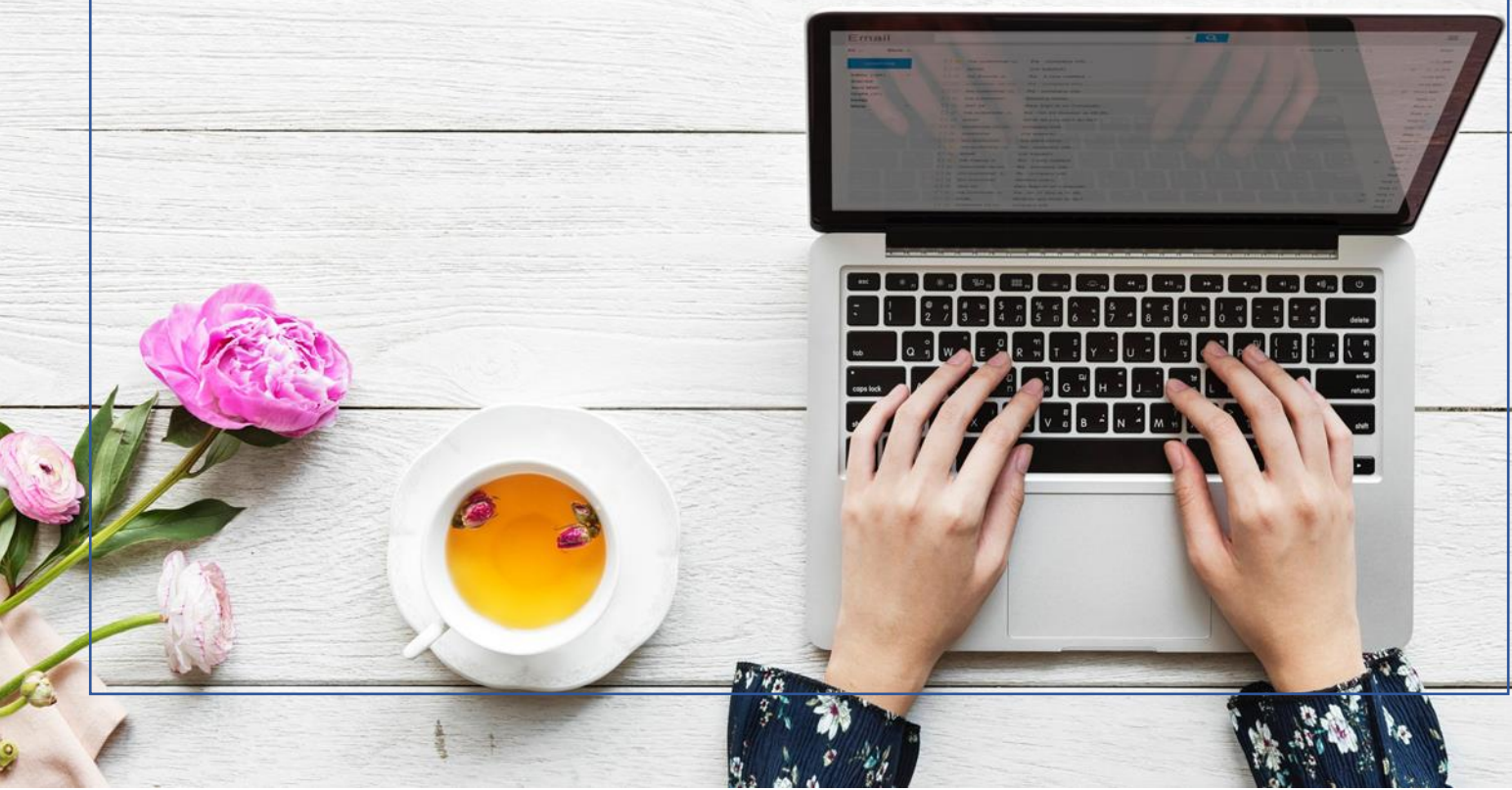
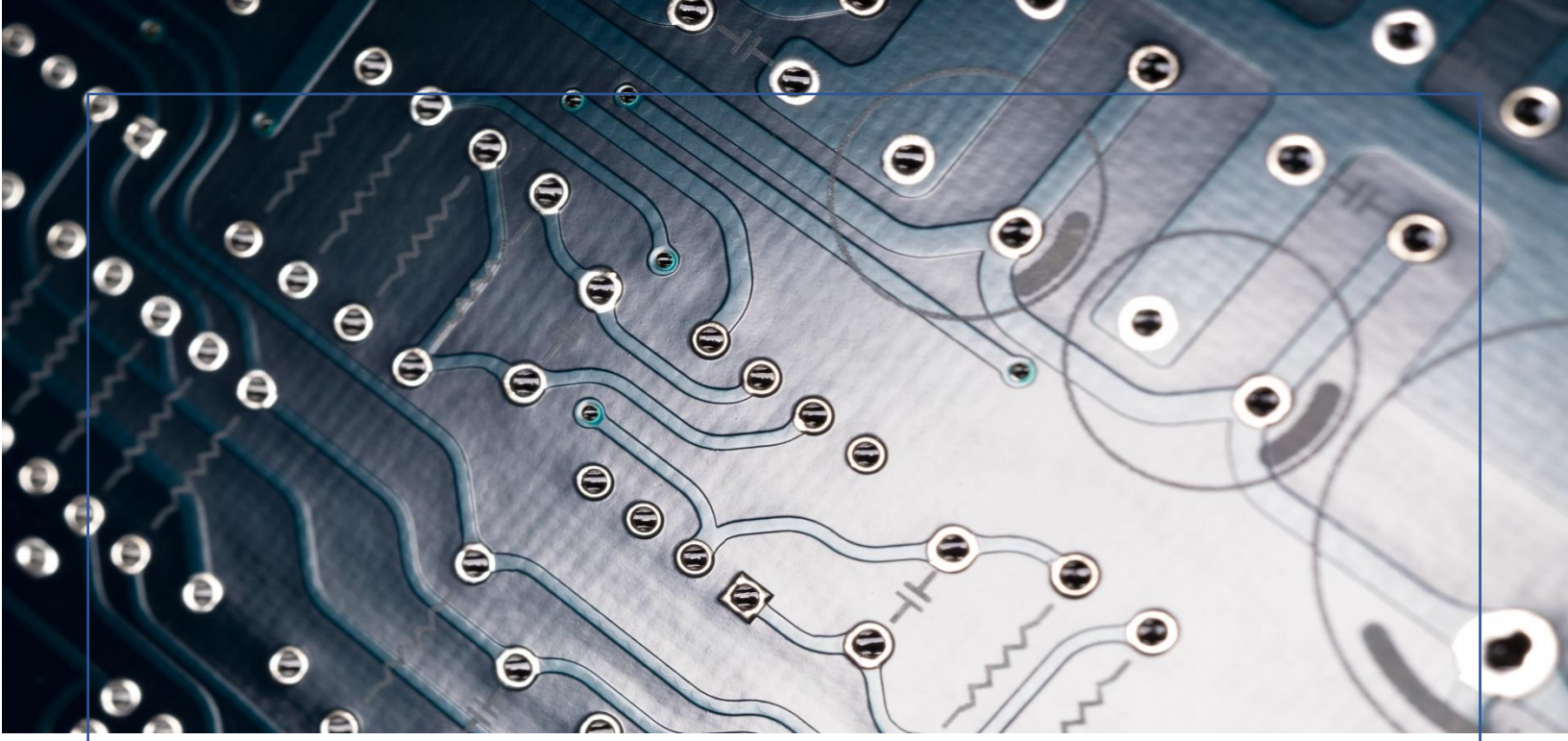


TABLE OF CONTENTS



INTRODUCTION

ABSTRACT

Reality Premedia Services has helped develop and deploy a CMS solution for a client in the Travel and Hospitality domain. The CMS used for development was EPiServer, which is a .NET based CMS.

The two biggest criteria for this CMS were performance and flexibility. We created a huge number of blocks and page types to allow the content editors to mix and match content blocks while building out content pages.

The solution was hosted on the EPiServer DXC (Digital Experience Cloud) and by the end hosted around 6 different sites. This was across three different servers based on the type (B2C, B2B etc.).

Other than generic content, this site also had to serve a lot of cards that were “booking cards” where different hotels or hotel + flights were advertised and could be filtered and searched. This data came from an external system. This needed to be synchronized periodically with the external service so that we could serve data quicker. There was also a “booking” block that could be added to different pages that allows the end user to book a deal once it had been selected.

The entire site was accessible by the WCAG guidelines and could be used completely using just eh keyboard as well. It was thoroughly tested using the Voiceover, NVDA, JAWS etc.

We started with 2 resources working with the US team full time and over a period of two years, scaled up-to 30 resources engaged with the client across three teams.

BACKGROUND

The direct client is the technology wing of the client, which deals in the Travel and Hospitality domain. They work with different resorts and help them with the branding, infrastructure, maintenance etc. They also handle their whole online presence. They also hold training programs for travel agents to educate them on the different resorts and features. They also own their own brand that is an aggregator of different resorts and packages as well as handle packages for different external groups like Funjet Vacations, Southwest vacations etc.

The client solutions approached us via our solution partner, Infarsight. They were working with an EPiServer vendor and wanted to change as they felt that development was progressing too slowly and the turnaround time for different projects was very high.

The “booking engine” was built by a separate team based out of Milwaukee and exposed APIs for integration. This was sold as a separate product to different companies, as well as needed to be integrated on the sites that we were building. Most of the brands in question already had websites built on different platforms, from Hermes, to AEM. The objective was to consolidate all these websites into a single system.

OBJECTIVES

- The solution needed to be highly customizable.
- The solution needed to be simple to use for content editors
- The solution needed to be Accessible
- The solution needed to be highly performant as the visitor count to these websites were very high
- The solution needed to be Compliant with Web standards
- The solution needed to work across multiple DXC instances

SOLUTION

INPUTS

When we were onboarded, the client had already built out their Agent Management websites. These had been done with the previous vendor. We needed to build the tools required for content editors to build websites for Vacation brands like Apple Vacations and Cheap Caribbean.

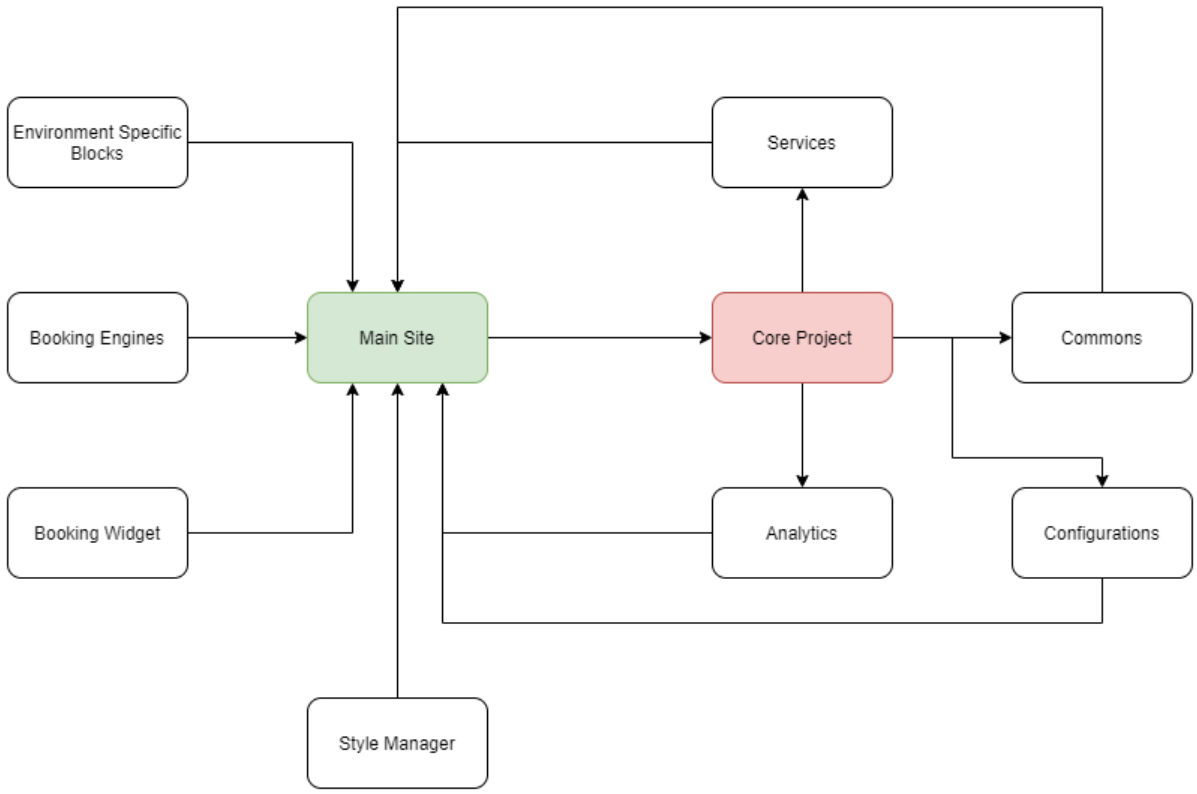
There was a design team that was in place that ensure that the site was designed according to the brand guidelines and we were provided the wireframes for this. We used these to determine the different block types we needed to construct to allow content editors to build out the website correctly.

From the backend side, we were also given the API documentation for the booking engine that allowed us to fetch deals that needed to be shown on our website. We were also given an additional constraint that since we have different environment for different types of services, we would need to ensure that at a code level, we could mix and match services and define available services per DXC.

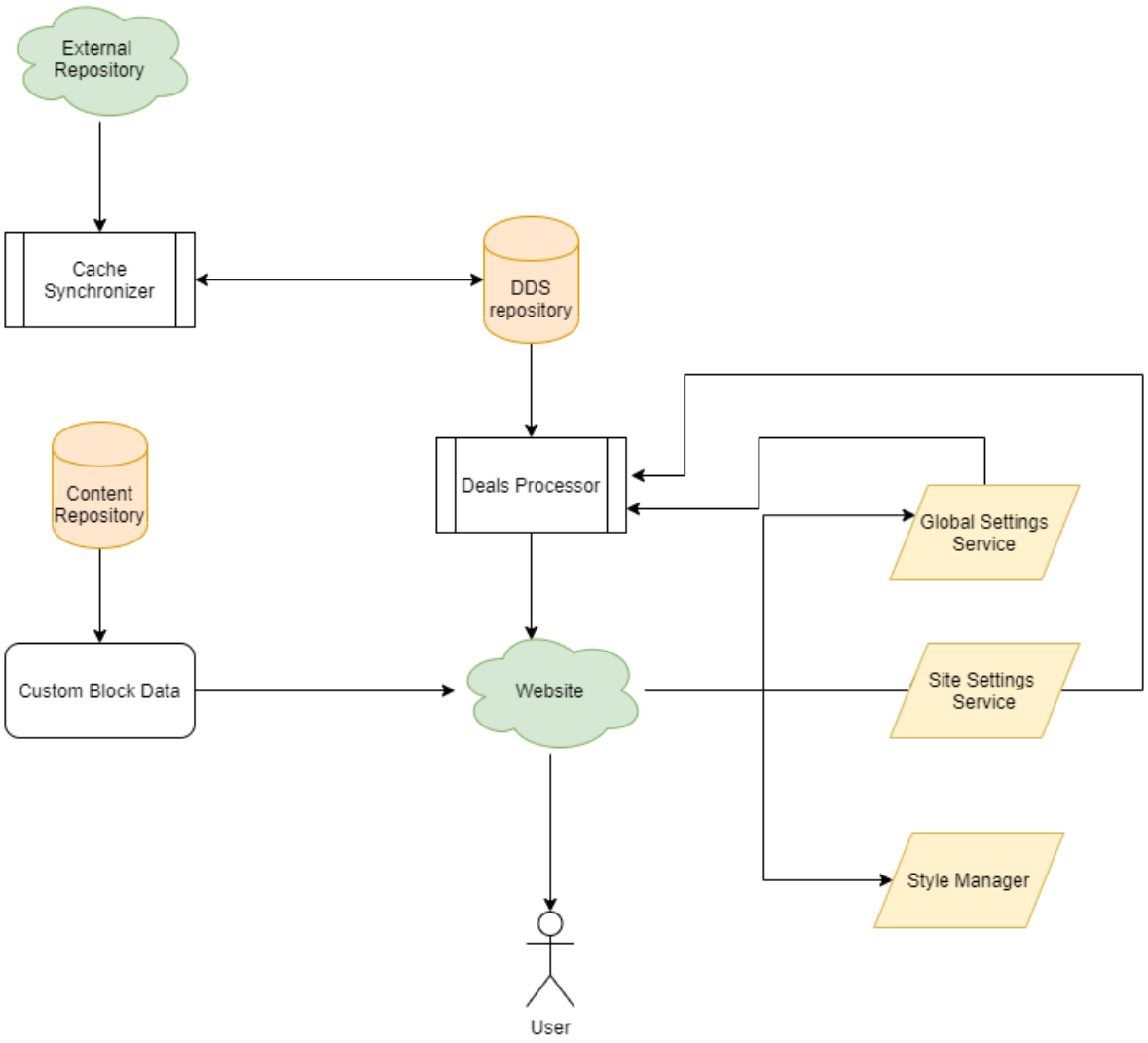
All development took place on Citrix VMs provided by the client as a security measure.

PROPOSED ARCHITECTURE

The architecture of EPiServer DXC is standard, so we will instead describe the architecture of the customizations we made for site settings, global settings as well as integrating with the booking engine.



Flow:



Technology Stack

The technology stack used was as follows.

- EPiServer CMS / ASP.NET MVC 5
- Frontend – jQuery + AngularJS
- Datastore - Azure SQL
- Content Data Storage – Azure Blob
- CDN – Cloudflare
- Cache Synchronization – Azure Event Hub

Execution

The first step after getting onboard was to evaluate the existing system and gather the constraints of the build out. We also interacted with the previous vendor and discussed challenges etc.

Based on that, we created a list of architectural improvements that needed to go into the application and started working on building that base. It is during this period that we worked on the code modularization using NuGet package addons etc. We also noticed that even though EPiServer uses a lot of Repository and other Design patterns, most of this was ignored when building out the initial site. We reworked the site to make sure that everything fit into the patterns used by EPiServer.

Following the initial architecture overhaul, the developers were each given a module/feature that they were responsible for and they set out to develop it. All the dependency management had already been setup by the Architect, so it was quite simple to work in modules like that.

A couple of months in, we had to setup a third DXC for a different type of business. This was the first test of the modular approach to building an environment. This went off quite successfully and it took a couple of weeks to build out a new DXC. This was just the initial stitching together and configuring and not building out any new features that were required for the new DXC.

One of the bigger tasks that we had to retrofit was accessibility. While not a priority during the initial phase of development, we needed to ensure that all sites built on this platform were accessible, while minimizing the amount of work that content editors would be doing. While this did not really require and architecture overhaul, but it required us to revisit every addon and ensure that the right metadata was in place. We confirmed that the site was accessible using JAWS with IE, NVDA with Firefox/Chrome and Voiceover with Safari.

To reduce development time, we also built out a “Style Manager”. Since we had already migrated to using SCSS behind the scenes, we build out an addon that allowed content “admins” to set some conditions and variables at a site level. This would be the piece that provided the scheme and other brand guidelines to the entire website and could be changed from website to website.

We also went out to build new site settings, global settings, analytics, cache manager and other modules to ensure that provided us the flexibility to modularize the code as well as manage who had access to what.

PROCESS & TEAM

The team started with two backend developers with an additional cloud architect to begin work on the application. In a couple of months, once the main framework was decided, we started to add multiple resources across different teams. We also onboarded multiple QA resources to help test Accessibility and other things.

- 9 Developers
- 3 Team Leads
- 4 QAs
- 1 Architect
- 2 Project Managers

- 8 Content Management

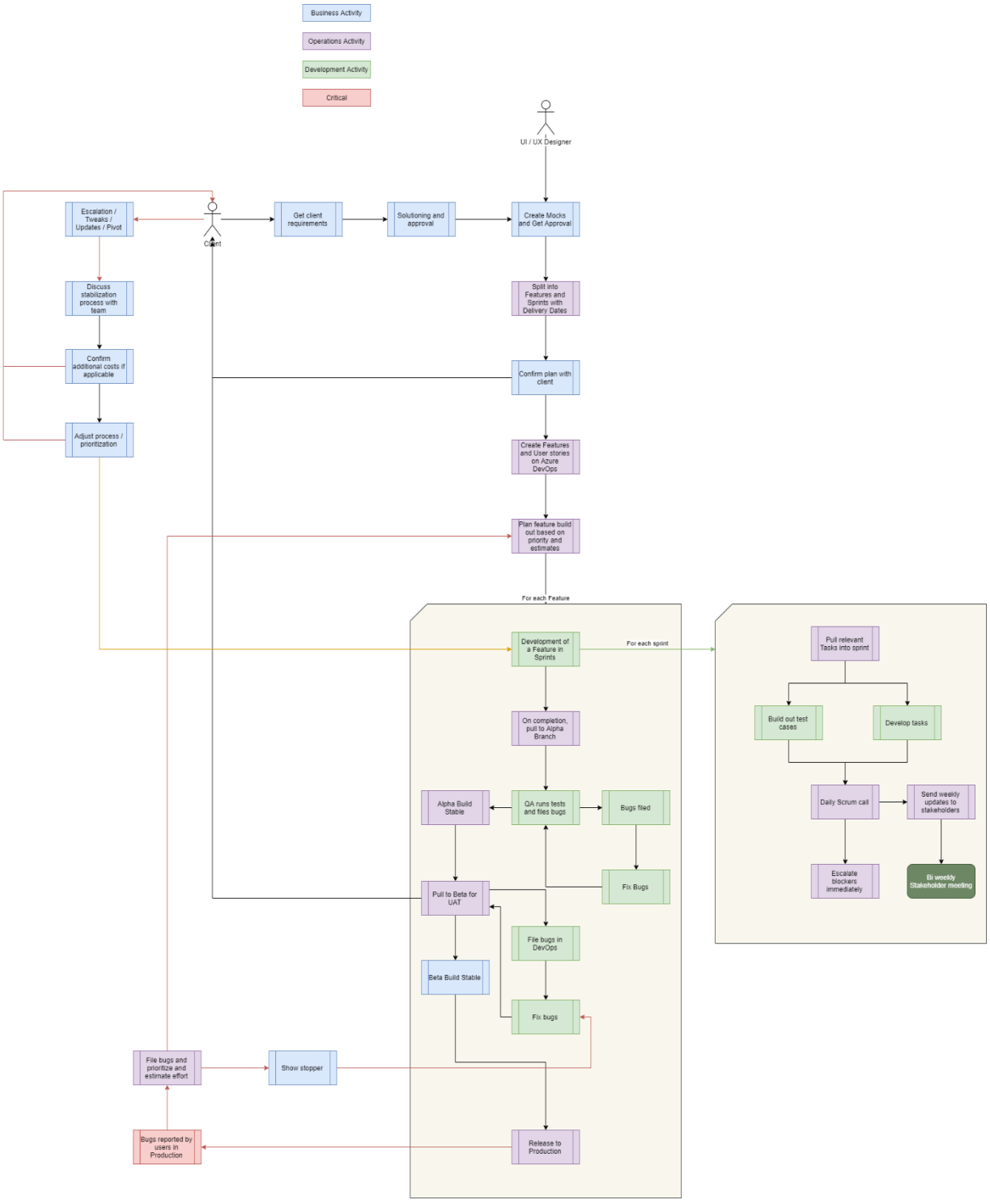
We follow Agile Scrum internally for all projects and this project is no exception. The difference here is that the in 2 of the four sub-projects, the Scrum master was from the client's side and we were using their boards on TFS instead of our boards.

We used TFS 2010 as our code repository. All user stories, features and bugs were tracked on azure boards along with all sprint management features.

All deployments and builds are completely automated using the TFS CI/CD pipelines which deploys both mobile apps as well as to the DXC cloud.

Bugs and user stories are "auto resolved" based on links created with user code check-ins and alpha pulls, instead of developers manually resolving items.

All pull requests made from a feature branch need to be code reviewed by the team lead before it is approved and integrated. Major releases required additional approvals from the dev manager and architect.


Business Activity
Operations Activity
Development Activity
Critical
UI / UX Designer
Client
Escalation / Tweaks / Updates / Pivot
Get client requirements
Solutioning and approval
Create Mocks and Get Approval
Discuss stabilization process with team
Split into Features and Sprints with Delivery Dates
Confirm additional costs if applicable
Confirm plan with client
Adjust process / prioritization
Create Features and User stories on Azure DevOps
Plan feature build out based on priority and estimates
For each Feature
Development of a Feature in Sprints
For each sprint
On completion, pull to Alpha Branch
Alpha Build Stable
QA runs tests and files bugs
Bugs filed
Fix Bugs
Pull to Beta for UAT
File bugs in DevOps
Beta Build Stable
Fix bugs
File bugs and prioritize and estimate effort
Show stopper
Bugs reported by users in Production
Release to Production
Pull relevant Tasks into sprint
Build out test cases
Develop tasks
Daily Scrum call
Send weekly updates to stakeholders
Escalate blockers immediately
Bi weekly Stakeholder meeting

CHALLENGES AND WORKAROUNDS

EPISERVER DATA REPOSITORY

We needed a place to store data fetched from API. The first choice was to user EPiServer commerce and put the deals as items there. But unfortunately, we did not have a license and a business decision was made to not purchase it. The second option was Redis, but this too was rejected as an option. So, we decided to use EPiServer's own caching mechanism that was based on HTTPCache.

When pulling data from the External API (for booking information) and caching it, we ran into an issue with the way EPiServer DXC is setup. Cache is not synchronized across multiple server instances within one DXC. The ISynchronizedObjectInstanceCache only synchronizes deletions. This means that depending on which instance is serving data, you may or may not get data.

We created a new module called "DDSCache". This was essentially a cache that was backed by DDS (Dynamic Data Store). The idea was that everything is loaded into HTTPCache, but If it needs to do a cold reload, then instead of having to make the API calls, it will use the database instead.

CODE SEGREGATION

Since we had multiple DXC with different audience types and feature sets, we wanted the feature set available to each one to be configurable without having to clone the code across those DXCs. Cloning would lead to a huge increase in maintenance. The idea was that when a developer setup an environment, they would be able to specify what modules it needed to run and then you could just spin up an instance with that feature set.

We decided to create multiple NuGet packages that were self-contained. The only dependency was the Core package that had all the interface definitions. If any addon needed to support site settings, it just declared it as a dependency and processed to use it in the standard Dependency Injection pattern. The dependent addon itself was responsible for managing the admin UI etc. for that feature. So, when we went from DXC to DXC, we were able to install the libraries needed for each on (for example, which booking engines were supported) and they were auto registered.

CONFIGURATION

By default, Episerver keeps "site settings" as a tab on the home page for content editors. This causes a couple of issues when managing permissions. This is an even bigger problem when you want modular code coming from different addons, because they would not be able to inject the required setting onto that screen.

We went out and created a brand-new service called the ISiteSettingsService. This service was backed by DDS and had a whole separate admin panel in EPiServer that could be controlled on its own with granular permissions (Permissions for Functions). Using this service, any external addon could inject its own definition of what it needed the content editor to specify at a site level and could be made completely plug and play. The same concept was extended to both the style manager as well as global settings.

TIMEFRAME

The first site went live in about a month with a team of three people. The subsequent sites took between 3-4 months each, excluding bugfixes, improvements etc. We were perpetually adding new features and capabilities to the websites as new sites came along.

Later, along with these websites, we even create a completely new addon that integrated with Widen and gave content editors the ability to embed data from the Widen DAM into their system. The entire engagement (so far) lasted 2.5 years.

CONCLUSION

Reality Premedia worked with the client to build a cutting-edge solution on top of EPiServer CMS. We added a ton of features that EPiServer does not support yet (at the time of writing) and ensure that all of them were extremely performant. The entire application was enterprise grade and not just your run of the mill website.

KEY TAKEAWAYS

- Built new addons from scratch
- Optimized underlying framework
- Build a better caching mechanism than the out of the box solution
- Supported easy maintenance across multiple DXCs
- Reduced GTM turnaround time