TeamCity

Bica Studios Case Study

What does your company do?

BICA STUDIOS is a team that crafts the best gaming entertainment to bring enduring and inspiring new brands.

We develop great games as a service through a free-to-play approach, allowing millions of players to access new interactive experiences, exciting worlds and know charismatic characters with no initial monetary barriers. Our mission is to make a stand on the entertainment market by exploring current and alternative revenue models.

What obstacles or opportunities was your organization faced with?

From a business point of view, the company has to deal with market saturation and high UA costs. Another strong concern was the validation of proper use of strong engagement mechanics alongside meaningful and inspiring content.

As far as technology, our main concern is being able to deal with and react to change as quickly as possible.

What was your experience like before TC?

Our building pipeline is a complex one.

Having a mobile game as our main product, we obviously target all platforms: iOS, Android and Windows Phone. On the Android platform we target 3 different stores: Google Play Store, Amazon and Aptoide. Each build has a debug and store version.

So: 1 game, 3 platforms, 5 different builds, 10 different setups.

With manual specific configurations for every step, you can guess the amount of time that we take to handle this process and why we call it a ‘release day’ instead of ‘release morning’ or ‘release hour.’

Enter Unity 5 and build setup configuration through editor scripts. Several manual steps could be coded and automated on the build process, reducing a lot of human errors and saving a little bit of time. However, build time was still enormous and demanded at least one full-time dedicated individual to handle.

We decided it was time to a look into the world of Continuous Integration and try to lift the burden of building the project off our development team.

What products / options did you consider?

The first step towards effective time usage was to build our own tool to handle the setup for each specific build with the touch of a button. Feeling creative, we dubbed it 'Setup Tool.'

By using this tool we were able to save precious time when context-switching between different builds. However, each individual build had little improvement on the time it took to complete.

Some team members had previous experience with Jenkins so we decided to give it a try. Their previous experience was just using it (with a technology that wasn't Unity-based) but not setting it up. So we quickly dropped it when we saw the huge learning curve that the setup process demanded.

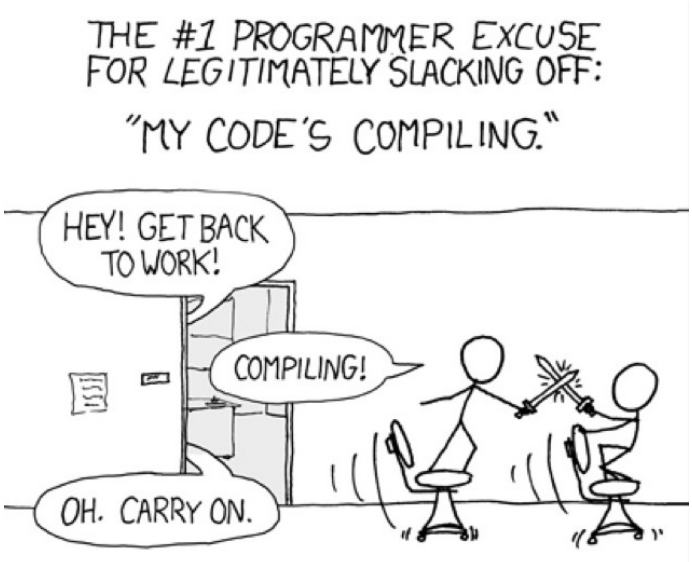
How was your evaluation performed?

When we decided to give TeamCity a spin, the first thing that caught our attention was the quickness with which we had a complex CI system fully set up: under 2 hours for 1 server and 2 agents. The fact that we found a plugin by MindCandy to help set up Unity builds greatly improved our satisfaction (though this plugin didn't target mobile builds which were our main target).

It was time to get our hands dirty! This was without a doubt the most time-consuming part of the process. That's why we're writing this article: so you don't have to go through the same trial-and-error process we did. The time it took was mainly due to lack of tutorials and proper documentation on Unity's side. This very recent feature was still being shaped while we were trying to use it: the ability to run the editor in batch mode and call static functions from an Editor Script.

Having overcome this barrier, we saw the world turning into a very happy and productive place. Despite all the initial effort and trial and error, on Day 2 we had the iOS and GooglePlay build pipeline for both our games set up, including all 4 different jobs.

The time came to try and run our first job. We went for the all-or-nothing test: build the game's iOS version, the most time-consuming task we had (about 40-60 minutes). When we successfully got a build artifact 21 minutes later, we just couldn't believe it. Evaluation done, TeamCity greenlit and no more swordplay!



Kidding! Instead of crossing swords, we kept ourselves busy measuring the duration of all the jobs. This is roughly what happened:

BUILD TYPE	TIME BEFORE TEAMCITY	TIME AFTER TEAMCITY
Smashtime iOS	45-60 min	18-25 min
Smashtime GooglePlay	15-25 min	5-13 min
SliceIN iOS	30-45 min	18-22 min
SliceIN GooglePlay	4-6 min	1-2 min

Bear in mind that besides cutting our build times in half (sometimes even more!), we managed to stop having the dreaded issue of “forgetting to disable that debug flag” or “enabling that optimization thing.”

What led you to stick with TeamCity?

Given these results, we had no other option but to adopt TeamCity as our CI platform and slowly keep adding all the other platforms to it. The effectiveness and correctness skyrocketed. Our confidence in the build results increased dramatically, and our developers wasted less and less time on the whole process. This in turn allowed us to perfect the production process even more, investing the gained time into bulletproofing, automating even more, and adding more relevant steps (that we’re going into detail in the next section).

What's your current build process with TeamCity like?

So how did we actually do it? In this section we go into each and every step with concrete explanations of what is happening, why we did it and why we feel it needed to be done, tackling both iOS and GooglePlay jobs.

iOS Build

To make an iOS build you'll generally need 3 steps:

1. Build the Xcode project in Unity
2. Build the iOS app from the Xcode project
3. Build the IPA from the iOS app

STEP 1

To have TeamCity do the Unity part, you need to create a class with static methods that run the build process. Then in the TeamCity job (build configuration), create a "Command Line" step that calls that function.

UnityBuilder.cs

```
public static string BuildPath()
{
	return Application.dataPath.Substring(0, Application.dataPath.LastIndexOf('/')) +
	"/Builds/";
}

public static void DoTheiOSBuild() {
	string outputDir = BuildPath() + "iOS/";
	//Build options that you can customize
	BuildOptions options;
	//Uncomment to set development build
	//options |= BuildOptions.Development
	//Uncomment to enable script debugging
	//options |= BuildOptions.AllowDebugging;

	//Set the correct platform
	BuildTarget target = BuildTarget.iOS
	EditorUserBuildSettings.SwitchActiveBuildTarget(target);

	// Get the editor scenes
	var scenes = new List<string>();
	foreach(var scene in EditorBuildSettings.scenes)
	{
		if(!scene.enabled) continue;
		scenes.Add(scene.path);
	}
```

```
        //Actually build
        var res = BuildPipeline.BuildPlayer(scenes.ToArray(), outputDir, target, options);
        if(!string.IsNullOrEmpty(res))
        {
                throw new Exception("BuildPlayer failure: " + res);
        }
}
```

Note: This file should be saved in the "Assets/Editor/" folder.

The code called on the command line should be something like this:

```
%UNITY_EXECUTABLE% -batchmode \
-executeMethod UnityBuilder.DoTheiOSBuild()  \
-projectPath "%teamcity.build.checkoutDir%" \
-logFile \
-quit
```

UNITY_EXECUTABLE is an environment variable defined in the buildAgent.properties file so that we can easily change Unity's executable location (or version!) without having to go through all the scripts to update it. Handy, isn't it?

This will open Unity in batch mode, execute the method, log the progress to TeamCity and then exit gracefully.

STEP 2

Now that we have the Xcode project built, we need to build the iOS App. We create a new build step "Command Line" and use these lines:

```
xcodebuild -project %teamcity.build.checkoutDir%/Builds/iOS/Unity-iPhone.xcodeproj \
-target "Unity-iPhone" \
-configuration Release clean build \
ENABLE_BITCODE=NO \
DEPLOYMENT_POSTPROCESSING=YES \
CODE_SIGN_IDENTITY="iPhone Developer: Your Name"
```

This will run the xcode building process in batch mode with Release config, clean, build and codesign with the given identity. The bitcode and postprocessing parts are our custom needs and were added here just to show you how to set up Xcode build settings vars. The result of this step will be the iOS app. Now we just have to make an IPA out of it!

STEP 3

To create the IPA, we add a new build step "Command Line" with the following code:

```
/usr/bin/xcrun -sdk iphoneos PackageApplication -v \
%teamcity.build.checkoutDir%/Builds/iOS/build/Release-iphoneos/yourtTargetName.app \
-o %teamcity.build.checkoutDir%/Builds/iOS/build/nameOfTheIPA.ipa
```

This creates the IPA file ready to be installed on your iOS device. However, there's no straightforward way of doing this without going through iTunes or sending the build to Testflight, waiting for it to be available, downloading and installing it.

Or is there?

Using Dropbox and a basic HTML page, you can directly install IPA files on your provisioned devices! We just need to add this extra lines to the final step:

```
mv -f %teamcity.build.checkoutDir%/Builds/iOS/build/nameOfTheIPA.ipa \
"/The/Path/To/Your/Dropbox/iOS"
```

This will move the created IPA to your Dropbox folder of choice. Then, in that Dropbox folder add these two files:

manifest.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyL-
ist-1.0.dtd">
<plist version="1.0">
<dict>
	<key>items</key>
	<array>
		<dict>
			<key>assets</key>
			<array>
				<dict>
					<key>kind</key>
					<string>software-package</string>
					<key>url</key>

<string>http://dl.dropbox.com/path/to/your/nameOfTheIPA.ipa</string>
				</dict>
			</array>
			<key>metadata</key>
			<dict>
				<key>bundle-identifier</key>
				<string>your.bundle.identifier</string>
				<key>bundle-version</key>
				<string>1.0.0</string>
				<key>kind</key>
				<string>software</string>
				<key>title</key>
				<string>App Name</string>
			</dict>
		</dict>
	</array>
</dict>
</plist>
```

index.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Page title</title>
</head>
<body>
<h2>iOS Build</h2>
<a href="itms-services://?action=download-manifest&amp;url=https://dl.dropbox.com/url/to/
your//manifest.plist">App Name</a></body>

</body>
</html>
```

Now, every time you run the job on TeamCity, you can just open the page from Dropbox on your mobile browser and install the latest version of your app!

Note: To get the URLs for the IPA and manifest from Dropbox, right-click them and choose “Copy Link”.

Note 2: For the install process to work, you need to uninstall the previous version of your app from your iOS device.

EXTRA STEP

This setup already improves dramatically the whole process of obtaining a new build. But TeamCity made us aim for more, to waste the least amount of time possible. There was still the waiting time between uploading the build to Dropbox and having it available on your device.

At Bica Studios we use Slack for communication, so we added a Slack application using an IFTT recipe to send a notification to our #builds channel when the file upload is done. This notification can be customized with a custom URL, so we tailored the URL to be that of the HTML file stored in Dropbox. BAM! As soon as the upload is done, we can have the new build on the device by clicking on the notification link, on Slack, on the device! Lots of time saved and lots of happy developers as a result.

Android Build

To make an Android build, you’ll generally need 1 step:

1. Build the APK through Unity.

STEP 1

To have TeamCity do the Unity part, create a class with static methods that run the build process. Then in the TeamCity build configuration, create a “Command Line” step that calls that function.

On our UnityBuilder.cs file add the following function:

```
public static void DoTheAndroidBuild() {
	string outputDir = BuildPath() + “Android/”;

	//Build options that you can customize
	BuildOptions options;

	//Set the correct platform
	BuildTarget target = BuildTarget.Android
	EditorUserBuildSettings.SwitchActiveBuildTarget(target);

	// Get the editor scenes
	var scenes = new List<string>();
	foreach(var scene in EditorBuildSettings.scenes)
	{
		if(!scene.enabled) continue;
		scenes.Add(scene.path);
	}

	PlayerSettings.Android.keystoreName = Application.dataPath + “path/toYour.keystore”;
	PlayerSettings.Android.keystorePass = “yourPass”;
	PlayerSettings.Android.keyaliasName = “yourAlias”;
	PlayerSettings.Android.keyaliasPass = “yourAliasPass”;
	PlayerSettings.Android.useAPKExpansionFiles = false;
```

```
        //Actually build
        var res = BuildPipeline.BuildPlayer(scenes.ToArray(), outputDir, target, options);
        if(!string.IsNullOrEmpty(res))
        {
                throw new Exception("BuildPlayer failure: " + res);
        }
}
```

The code called on the command line step should be something like this:

```
%UNITY_EXECUTABLE% -batchmode \
-executeMethod UnityBuilder.DoTheAndroidBuild()  \
-projectPath "%teamcity.build.checkoutDir%" \
-logFile \
-quit
```

EXTRA STEP

To install a build on Android, simply open an apk through Dropbox and the install process happens. To enable this, we create a new build step "Command Line" with the following code:

```
mv -f %teamcity.build.checkoutDir%/Builds/Android/yourApkName.apk \
"/Path/to/your/Dropbox/Android"
```

The same kind of IFTT recipe can be created to allow installing the build right away from Slack on the device.

What are your key outcomes of using TeamCity? What next steps are you planning?

There are two types of results to analyze here: the short term and long term effects of using TeamCity as our CI platform. The short term results are clear: shorter building times save us more time to spend on developing the product. Also, fewer manual steps reduce the number of human errors and increase the confidence we have in the builds and their overall quality.

In the long term, the fact we have the builds available for testing with such ease allows for more testing to happen sooner and more regularly. This in turn helps us detect problems sooner and avoid small problems from turning into big ones later on.

Overall, these benefits deeply affect our product development pipeline and change it for the best.

Looking back, going with TeamCity was the right step to take and we only regret not doing it earlier. If you have a project with different and complex manual building setups, you absolutely have to give this a try.

Thanks to the team at Bica Studios for sharing their success story with us. It's certainly inspiring. If you have a similar tale to tell, we're listening!

Should you have any questions to the Bica Studios team, feel free to contact them directly at nuno.monteiro@bicastudios.com.