Source [2014-01-28]:
http://www.aviransplace.com/2013/03/16/the-roard-to-continues-delivery-part-1/

The Road To Continuous Delivery - Part 1

The following series of posts are coming from my experience as the head of back-end engineering at Wix.com. I will try to tell the story of Wix and how we see and practice continuous delivery, hoping it will help you make the switch too.

So you decided that your development process is too slow and thinking to go to continuous delivery methodology instead of the "not so agile" Scrum. I assume you did some research, talked to a couple of companies and attended some lectures about the subject and want to practice continuous deployment too, but many companies asking me how to start and what to do?
In this series of articles I will try to describe some strategies to make the switch to Continuous delivery (CD).

Continuous Delivery is the last step in a long process. If you are just starting you should not expect that you can do this within a few weeks or even within few months. It might take you almost a year to actually make several deployments a day.
One important thing to know, it takes full commitment from the management. Real CD is going to change the whole development methodologies and affect everyone in the R&D.

Phase 1 – Test Driven Development
In order to do a successful CD you need to change the development methodology to be Test Driven Development. There are many books and online resources about how to do TDD. I will not write about it here but I will share our experience and the things we did in order to do TDD. One of the best books I recommend is "Growing Object Oriented Guided by tests"

A key concept of CD is that everything should be tested automatically. Like most companies we had a manual QA department which was one of the reasons the release process is taking so long. With every new version of the product regression tests takes longer.

Usually when you'll suggest moving to TDD and eventually to CI/CD the QA department will start having concerns that they are going to be expandable and be fired, but we did not do such thing. What we did is that we sent our entire QA department to learn Java. Up to that point our QA personnel were not developers and did not know how to write code. Our initial thought was that the QA department is going to write tests, but not Unit tests, they are going to write Integration and End to End Tests.

Since we had a lot of legacy code that was not tested at all, the best way to test it, is by integration tests because IT is similar to what manual QA is doing, testing the system from outside. We needed the man power to help the developers so training the QA personal was a good choice.

Now as for the development department, we started to teach the developers how to write tests. Of course the first tests we wrote were pretty bad ones but as time passes, like any skill, knowing how to write good test is also a skill, so it improves in time.
In order to succeed in moving to CD it is critical to get support from the management because before you see results there is a lot of investments to be done and development velocity is going to sink

even further as you start training and building the infrastructure to support CD.

We were lucky to get such support. We identified that our legacy code is unmaintainable and we decided we need a complete re-write. However this is not always possible, especially with large legacy systems so you may want to make the switch only for new products.
So what we did is we stopped all the development of new features and started to progress in several fronts. First we selected our new build system and CI server. There are many options to choose from, we chose Git, Maven, Team City and Artifactory. Then we started to build our new framework in TDD so we could have a good foundation for our products. Note that we did not anything that relates to deployment (yet).

Building our framework we set few requirements for ourselves. When a developer checks our code from Git he should be able to run unit test and integration tests on his own laptop WITHOUT any network connections. This is very important because if you depend on fixtures such as remote database to run your IT, you don't write good integration tests, limit to work only from the office, your tests will probably run slower and you will probably get into trouble running multiple tests on the same database because tests will contaminate the DB.

The Road To Continuous Delivery - Part 2 – Visibility

A key point for a successful continuous delivery is to make the production matrix available to the developers. At the heart of continuous delivery methodology is to empower the developer and make the developers responsible for deployment and successful operations of the production environment. In order for the developers to do that you should make all the information about the applications running in production easily available.

Although we give our developers root (sudo) access for the production servers, we do not want our developers to look at the logs in order to understand how the application behaves in production and to solve problems when they occur. Instead we developed a framework that every application at Wix is built on, which takes care of this concern.

Every application built with our framework automatically exposes a web dashboard that shows the application state and statistics. The dashboard shows the following (partial list):

- Server configuration
- All the RPC endpoints
- Resource Pools statistics
- Self test status (will be explained in future post)
- The list of artifacts (dependencies) and their version deployed with this version
- Feature toggles and their values
- Recent log entries (can be filtered by severity)
- A/B tests
- And most importantly we collect statistics about methods (timings, exceptions, number of calls and historical graphs).

We use code instrumentation to automatically expose statistics on every controller and service end-point. Also developers can annotate methods they feel is important to monitor. For every method we can see the historical performance data, exception counters and also the last 10 exceptions for each method.

We have 2 categories for exceptions: Business exceptions and System exceptions.
Business exception is everything that has to do with application business logic. You will always have these kinds of exceptions like validation exceptions. The important thing to monitor on this kind of exception is to watch for sudden increase of these exceptions, especially after deployment.

The other type of exception is System exception. System exception is something like: "Cannot get JDBC connection", or "HTTP connection timeout". A perfect system should have zero System exceptions.

For each exception we also have 4 severity levels from Recoverable to Fatal, which also help to set fine grain monitoring (you should have zero fatal exception)
Using this dashboard makes understanding what is going on with the server easy without the need to look at the logs (in most cases).

One more benefit to this dashboard is that the method statistics are also exposed in JSON format which is being monitored by Nagios. We can set Nagios to monitor overall exceptions and also per method exceptions and performance. If the number of exceptions increases or if we have performance derogation we can get alerts about the offending server.

The App dashboard is just one method we expose our production server to the developer. However the app dashboard only shows one server at a time. For an overview look of our production we also use an external monitoring service. There are several services you can use like AppDynamics, Newrelic etc'.
Every monitoring service has its own pros and cons; you should try them out and pick whatever works best for you (we currently use Newrelic).

Every server is deployed with a Newrelic agent. We installed a large screen in every office which shows the Newrelic graphs of our production servers. This way the developers are always exposed to the status of the production system and if something bad is going on we immediately see it in the graph, even before the alert threshold is crossed. Also having the production matrix exposed, the developers see how the production system behaves in all hours of the day. It is not a rare case where a developer looks at the production graphs and decides that we can improve the server performance, and so we do. We saw time and again that every time we improve our servers performance we increase the conversion rate of our users.

The fact that we expose all this information does not mean we do not use logs. We do try to keep the developers out of the log files, but logs have information that can be useful for post mortem forensics or when an information is missing from the dashboards.

To summarize, you should expose all the information about your production environment to the developers in an easy to use interface, which includes not only the application statistics but also system information like routing tables, reverse proxy settings, deployed servers, server configurations and everything else you may think of that can help you understand better the production system.

Continuous Delivery - Part 3 - Feature Toggles

One of the key elements in Continuous Delivery is the fact that you stop working with feature branches in your VCS repository; everybody works on the MASTER branch. During our transition to Continuous Deployment we switched from SVN to Git, which handles code merges much better, and has some other advantages over SVN; however SVN and basically every other VCS will work just fine.

For people who are just getting to know this methodology it sounds a bit crazy because they think developers cannot check-in their code until it's completed and all the tests pass. But this is definitely not the case. Working in Continuous Deployment we tell developers to check-in their code as often as possible, at least once a day. So how can this work? Developers cannot finish their task in one day? Well there are few strategies to support this mode of development.

Feature toggles
Telling your developers they must check-in their code at least once a day will get you the reaction of something like "But my code is not finished yet, I cannot check it in". The way to overcome this "problem" is with feature toggles.

Feature Toggle is a technique in software development that attempts to provide an alternative to maintaining multiple source code branches, called feature branches.
Continuous release and continuous deployment enables you to have quick feedback about your coding. This requires you to integrate your changes as early as possible. Feature branches introduce a by-pass to this process. Feature toggles brings you back to the track, but the execution paths of your feature is still "dead" and "untested", if a toggle is "off". But the effort is low to enable the new execution paths just by setting a toggle to "on".

So what is really a feature toggle?
Feature toggle is basically an "if" statement in your code that is part of your standard code flow. If the toggle is "on" (the "if" statement == true) then the code is executed, and if the toggle is off then the code is not executed.
Every new feature you add to your system has to be wrapped in a feature toggle. This way developers can check-in unfinished code, as long as it compiles, that will never get executed until you change the toggle to "on". If you design your code correctly you will see that in most cases you will only have ONE spot in your code for a specific feature toggle "if" statement.

Feature toggles do not have to be Booleans. You can have feature toggles that are Enumerations, Strings, integers or any other value, as long as it fits your flow. For instance let's say you want to migrate from one database to another. During the migration process the feature toggle can have 4 values:
1 - read from the old database;
2 - read from old database and fallback to new database;
3 - read from new database and fallback to old database;
4 - use new database only.

Using feature toggles for quick feedback
There is another great advantage to Feature toggles. Not only do they use to check-in unfinished code, they also used for getting fast feedback.

Once a developer thinks a feature is finished, or even half-finished you can use the feature toggle to expose the new feature for internal testing or for getting quick feedback from product managers. For instance you can have in your "if" statement that checks if the user who is logged-in has certain privileges; or works at your company, or even equals to a specific user id, then the new feature is open and operational. This way product managers can experience the new feature on production, on a

live system very soon, even if it is not completed, and give the developer quick feedback about the product. While the product manager sees and experience the new feature, other users are not affected by this, because for them the feature toggle is "closed".

Internal Testing
When a developer declares that a new feature is finished we at Wix first open new features internally to all the employees in the company for internal testing. This way we use our own employees for QA, which gives us hundreds of QA users that act as just regular users of our system. After few hours or few days that we test new features internally, we open it to the general public (with or without A/B test, depends on the use case).

Refactoring using Feature Toggles
One other very useful use case for feature toggles is refactoring. Since all the developers are working on the MASTER branch, refactoring can be a challenge. But using feature toggles can ease the pain.

Usually when refactoring you change one or more method implementation, replace classes and even change an entire workflow. Now when using continuous delivery and your code has to be checked-in at least once a day, this affects how you refactor.

When doing a large refactoring job you DO NOT change any of the existing code. What you do is write a new (and improved) code that replaces the old code. Using feature toggles you control when to use the new and refactored code that lives alongside with the old and "not so good" code. Once you are done with the refactoring job and opened the feature toggle to use the new code, you can safely delete the old code from your system.

Using this method has a very important benefit, that as long as you do not delete the old code you can always go back to it (flipping the toggle) if you discover a bug in the new code on production.

Feature toggle manager and BI
When you use feature toggle it is important to know which feature toggles are active, see what their values are, how long these are active and have the ability to make modifications to the toggles in real time.

For this job we at Wix built a library called Feature Toggle Manager (soon to be opened as open source). Feature toggle manager has a list of feature toggles and their values. It is backed by a database where we can modify the values via our back-office.

We use the feature toggle manager to provide the "if" statement in the code what is the value of a feature. The Feature Toggle Manager has a set of strategies that it acts upon to determine the toggle value. Some of the strategies we use are: User credentials, Wix employees, GEO location, Browser user-agent, and percentage. You can build your own strategies to determine which value the Feature Toggle Manager will return given a use case. It is important to have a default value for a feature, which defines the default behavior of your system. The default value can also change in time after a feature in mature.

Since every flow uses the Feature Toggle Manager to get the toggle value we can also report to our BI systems on every event which features (and their values) where active and passed in the current user flow.

Cleaning feature toggles
Feature Toggles should not exist forever in your code. After a feature is opened and you decide there is no need to change the toggle back to inactive, developers should go back to the code and remove the "if" statement from the code, cleaning it from unnecessary if statements and unused toggles. A

Feature toggle manager can help you with that since it shows you all the features and how long they are active. Also you can do a quick search in the code to find the places feature toggles are used.

Testing with feature toggles
Testing features can be a challenge. Remember, doing Continuous Delivery means that you have to work using Test Driven Development. Usually with unit test you would not have any problems, because you test the actual methods, and in most cases bypassing the feature toggle flow management. However when you write your integration tests you may have a problem, since you are testing end to end flows and have to pass through the Feature Toggle "if" statement, when it is both "closed", to test regression, and "open" to test the new feature flow.

This is another case where Feature Toggle Manager can help you. In your tests during setup you can set the values in the Feature Toggle Manager to whatever flow you are testing. Since Feature Toggles always have default values (that are usually off) all the old tests should work as before, since you do not change the old behavior. However when you write integration tests for the new feature you need to set the feature toggle to "on" during your test setup and thus enable the flow go through the new feature and test that too.

Continuous Delivery - Part 4 - A/B Testing

From Wikipedia: In web development and marketing, as well as in more traditional forms of advertising, A/B testing or split testing is an experimental approach to web design (especially user experience design), which aims to identify changes to web pages that increase or maximize an outcome of interest (e.g., click-through rate for a banner advertisement). As the name implies, two versions (A and B) are compared, which are identical except for one variation that might impact a user's behavior. Version A might be the currently used version, while Version B is modified in some respect. For instance, on an e-commerce website the purchase funnel is typically a good candidate for A/B testing, as even marginal improvements in drop-off rates can represent a significant gain in sales. Significant improvements can be seen through testing elements like copy text, layouts, images and colors.

Although it sounds similar to feature toggles, there is a conceptual difference between A/B testing and feature toggles. With A/B test you measure an outcome for of a completed feature or flow, which hopefully does not have bugs. A/B testing is a mechanism to expose a finished feature to your users and test their reaction to it. While with feature toggle you would like to test that the code behaves properly, as expected and without bugs. In many cases feature toggles are used on the back-end where the users don't not really experience changes in flow, while A/B tests are used on the front-end that exposes the new flow or UI to users.

Consistent user experience.
One important point to notice in A/B testing is consistent user experience. For instance you cannot display a new menu option one time, not show the same option a second time the user returns to your site or if the user refreshes the browser. So depending on the strategy you're A/B test works to determine if a user is in group A or in group B , it should be consistent. If a user comes back to your application they should always "fall" to the same test group.

To achieve consistent user experience in a web application is tricky. Most web applications have two types of users: Anonymous user – a user that is not signed in to your web-app; And a signed-in user – a user that have a valid session and is authenticated on your site.

For authenticated users achieving consistency is easy. The algorithm you choose to assign the user to a specific test group should work on the user ID. A simple algorithm is modulus on the user ID. This will ensure that whenever the user returns to the site, regardless of the computer or browser the users logs-in from he will always get the same value.

For anonymous user this is more complex. Since you don't know who the user is you cannot guarantee a consistent behavior. We can mitigate the problem by storing a permanent cookie on the user's browser with the value of the A/B test group the user is assigned to. This will ensure the next time the user returns to the site he will get the same group assignment (you should only assign a user to a test group once). However this method has a flaw because of how the web works. If the user surfs to the site from a different browser, different computer or if they clean the browser cookies you cannot know that the user was assigned to a specific test group in the past and you would probably assign the user again to a test group (but he may be assigned to a different group).

A/B testing strategies.
The most common strategy to assign a test group is percentage. You can define what percentage of your users will get A and what percentage will get B.
Like feature toggles you can have multiple strategies to determine a test group to users. Such strategies we use at Wix are language, GEO location, user type, registration date and more. Of course you can also combine strategies, for instance: "50% of UK users would get B and all the rest would get A".

A very important rule is that bots like google **bot will always get the "A" version**, otherwise it may index pages that are under experiment and might fail.

Reporting and analysis.
Since the whole point of A/B testing is to determine if a new feature improves your goals or not all the users who were assign to a test group should be tracked and once you decide you have a large enough test participants, analyze your data and decide if the new feature is good or not. If the test is successful you would probably increase your test group or even stop the test and expose the new feature to all your users. On the other hand if the test was not successful you would stop the test and revert.
If a test was not successful, but you want to try and improve and restart the test you could pause the test in order to keep the user experience consistent, do not assign more users to get the new feature group, but whomever got assigned to see the new feature will keep see it. When you make the necessary improvement you should resume the test and resume assigning users to test groups.

Now you may ask yourself what does A/B test has to do with continuous deployment? Well since the whole point of continuous deployment is to get a quick feedback from users, A/B testing is a great mechanism to get this feedback.

Continuous Delivery - Part 5 - Startup - Self Test

So far we discussed Feature Toggle and A/B testing. These two methods enable safe guards that your code does not harm your system. Feature toggles enable to gradually use new features and gradually expose it to users, while monitoring that the system behaves as expected. A/B testing on the other hand let you test how your users react to new features. There is one more crucial test you as a developer should write that will protect your system from bad deployments (and also be a key to gradual deployment implementation).

Self-test sometimes called **startup test** or **post deployment test** is a test where your system checks that it can actually work properly. A working program does not only consist of the code that the developer write. In order for an application to work it needs configuration values, external resources and dependencies such as databases and external services it depends on to work properly.

When an application loads and starts up its first operation should be Self-test. Your application should refuse to process any operation if the self-test did not pass successfully.

Self-test have two main use cases:
1. Post deployment test – A post deployment test is a test that aims to verify two things. It should verify that the artifact is deployed with the correct configuration and that it can work properly.
2. Restart test – Verify that all the first level dependencies work and your application could potentially perform its task.

Post Deployment Test
In Continuous Delivery a deployment is being done into a live system and to keep operate with zero downtime. Having said that a self-test should check the following:
- Check that you have a connection to the database.
- Verify that the database schema is what you would expect.
- All the resources your application needs are accessible and loaded properly.
- If your architecture is SOA, check that all the services you depend on are reachable.
- All the external services have the operations you expect them to have.
- Any other test your system need to do in order to declare itself operational

Restart test
Unlike post deployment test, when you restart a service it is usually due to a problem, and not due to a deployment. Because of that you need to only test your first level dependencies and not external services because you cannot be sure that these services actually operational. If you do check external services you might not be able to reload your system. For example if service A calls a method on service B, and service B calls a method on service A, after restart of both services they would fail the test because they have a circular dependency. So restart test should only check the following:
- Check that you have a connection to the database.
- Verify that the database schema is what you would expect.
- All the local resources your application need are accessible and loaded properly.

Now since restart test does not guarantee that external services are operational at the time of the restart you should have a retry policy for external resources you may load.

At Wix we built a framework that once a service loads it will return "Service Temporary Unavailable" error until the self test has passes successfully. We have a flag (basically this is a file on the file system) that "tells" the self-test if it is running a post-deployment test or post-restart test. During deployment the deployment script deletes the "lock file" thus ensuring the full post-deployment test in run. At the end of the self-test the application writes a "lock file" so consecutive tests are just startup-tests and not post-deployment tests.

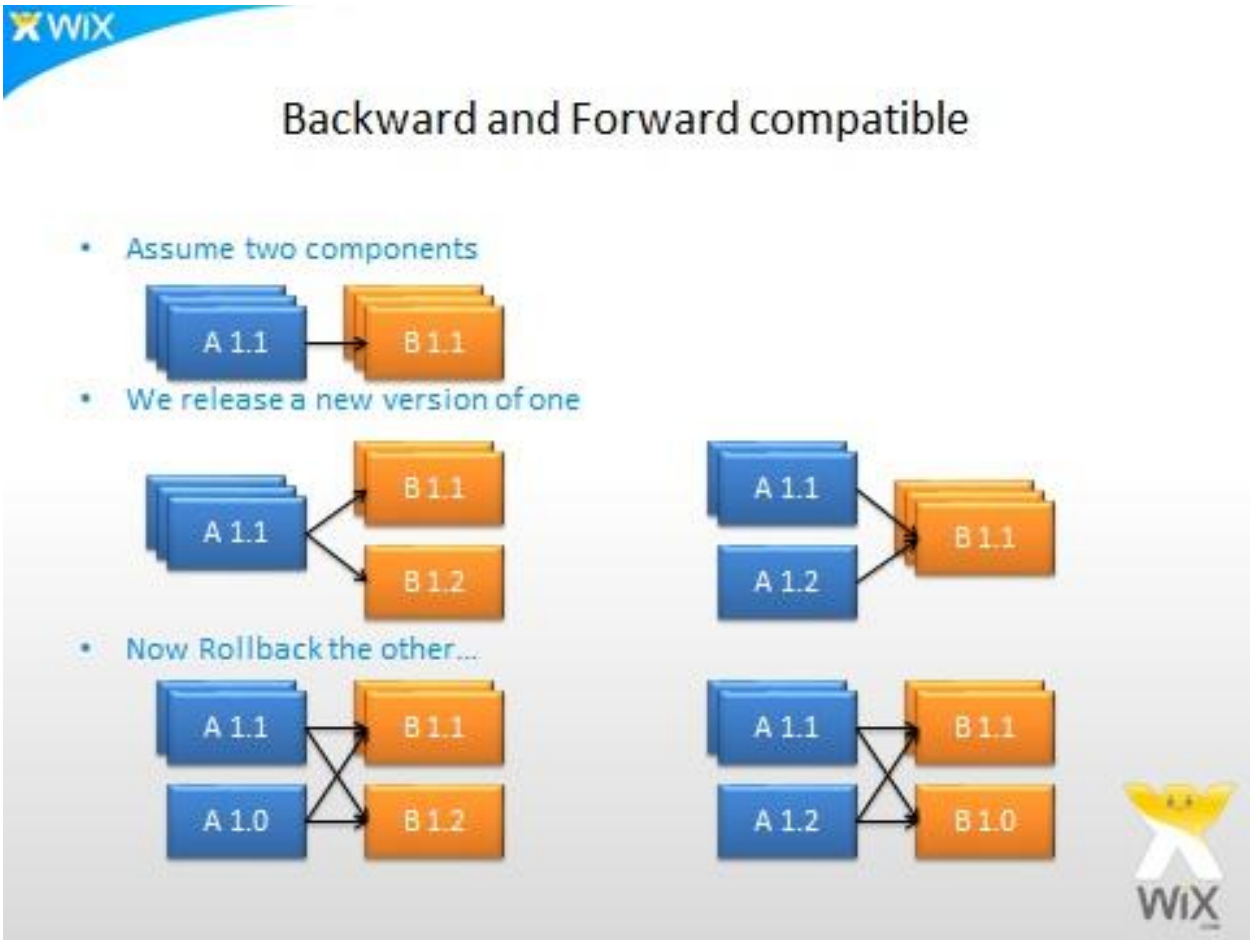
Gradual Deployment
Self-test is not only important to check the health of the deployment, it is also crucial when you do gradual deployment. Since you don’t deploy all the servers at once you need to deploy servers gradually but only after the post-deployment test passed you can deploy the next server.

Continuous Delivery - Part 6 - Backward & Forward

One very important mind set developers will have to adopt and practice is backward and forward compatibility.

Most production system do not consist on just one server, but a cluster of servers. When deploying new piece of code, you do not deploy it to all the servers at once because part of Continuous deployment strategy is zero downtime during deployment. If you deploy to all the servers at once and the deployment require a server restart then all the servers will restart at the same time causing a downtime.

Now think of the following scenario. You write a new code that requires a new field in a DTO and it is written to a database. Now if you deploy your servers gradually you will have a period of time that some servers will have the new code that uses the new field and some will not. The servers that have the new code will send the new field in the DTO and the servers that not yet deployed will not have the new field and will not recognize it.



One more important concept is to avoid deployment dependencies where you have to deploy one set of services before you deploy the other set. If we'll use our previous example this will even make things worse. Let's say you work with SOA architecture and you have now clients that send the new field and some clients that do not. Or you deploy the clients that now send the new field but you have not yet deployed the servers that can read them and might break. You might say, well I will not do that and I will first deploy the server that can read the new field and only after that I'll deploy the client that sends it. However in Continuous deployment as easily as you can deploy new code you can

also rollback you code. So even if you deploy first the server and then the client you might now roll back the server without rolling back the client, thus creating again the situation where clients send unknown fields to the server.

So here comes the next and highly important practice. **ALL code has to be backward AND forward compatible**. So what that means? In our previous example it is very simple. Old servers have to ignore unknown fields and parameters (this is the forward compatibility), your servers are resilient to API extension. Now new servers that do know how to handle the new field have to also handle old clients that do not send this new field. In our example they can simply use default value in case the clients have not been deployed yet and do not send the new field (This is backward compatibility).

In most cases backward compatibility is easier because it is new code that need to handle old clients. But forward compatibility is sometimes tricky because you have old code running that now need to handle a new use-case it was not designed to handle when it was initially written. If you reach a point where you know your old code cannot handle new data, a simple way of solving that problem is to release an intermediate version that first "teaches" the old code to gracefully handle new data it previously did not know; and after you are confident it works fine you can deploy the new code that actually uses the new data. Now if you need to roll back the new code your intermediate version could handle the pieces of data written by the rolled back code and ignore it gracefully without breaking.

At Wix we had an extreme case were due to a misconfiguration we rolled back our entire production system a whole month back. After we fixed it we realized that even though our system rolled back a whole month back, everything kept working with zero downtime because everything was backward and forward compatible and the servers handled new data written by a code that is newer. You can read more about our continuous rollback here.

Continuous Delivery - Part 7 - Cultural Change

In order for continuous delivery to work the organization has to go a cultural change and switch to Dev-Centric Culture.

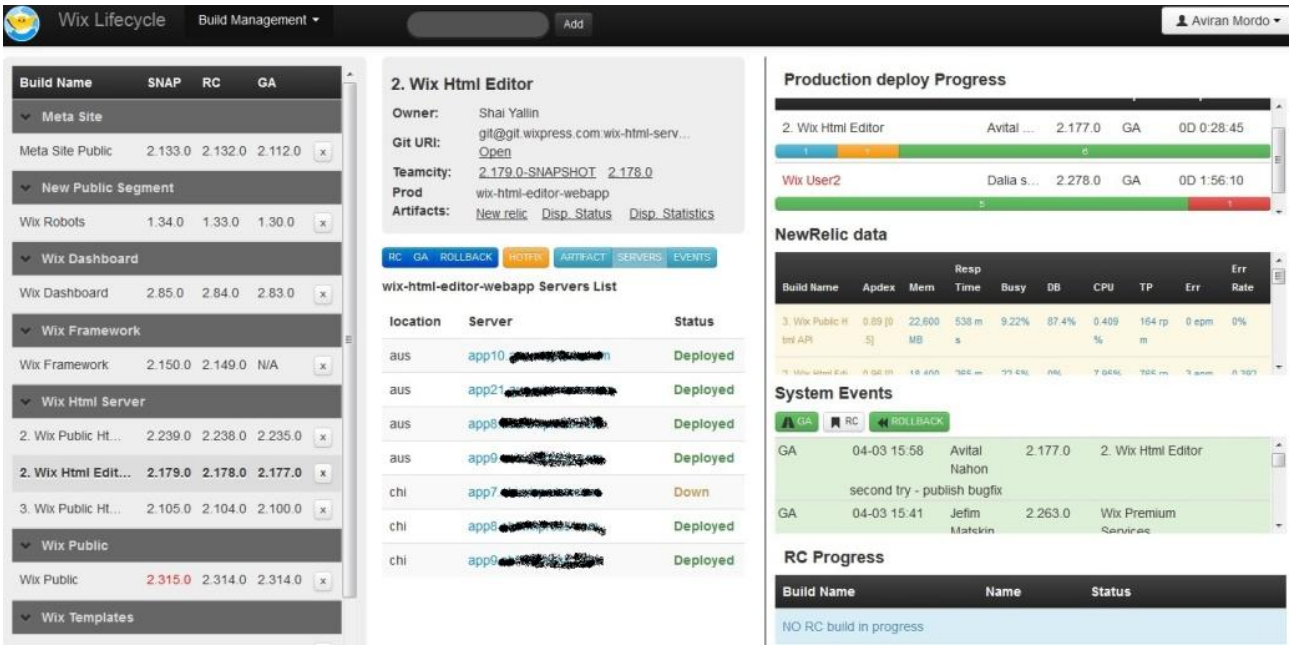
Continuous delivery gives a lot of responsibility in the hand of the developer and as such the developer need to have a sense of ownership. At Wix we say that it is the developer's responsibility to bring a product / feature to production, and he is responsible from the feature inception to the actual delivery of the product and maintaining it on the production environment.

In order to do that several things have to happen.

Know the business :
The developer has to know the business and be connected to the product he is developing. By understanding the product the developers makes better decisions and better products. Developers are pretty smart people and they don't need to have product specs in their face (nobody actually reads them). Our developers work together with the product mangers to determine what the product for the feature should be. Remember while the actual product may have several features bundled together for it to be valuable for the customer, we develop per feature and deploy it as soon as it is ready. This way both the product manager and the developer get to test and experience each feature on production (it is only exposed to them via Feature toggle) and determine if it is good enough, and may cause the direction of the product to change not according to plan, and actually develop a different next feature than the planned one.

Take ownership
Developers are taking ownership on the whole process, which means that they are the ones that need to eventually deploy to production. This statement actually changes the roles of several departments. What the company needs to do is to remove every obstacle in the developers way to deploy quickly on his own to production.

The operations department will no longer be doing the deployments. What they will do from now on is to create the automated tooling that will allow the developers to deploy on his own.
Operations together with dev will create the tooling to make the production environment visible and easy to understand to developers. Developers should not start a shell console (ssh) in order to view and know what is going on with the servers. We created web views for monitored metrics of both system and application metrics and exceptions.



Wix deployment dashboard

BI and monitoring need to no longer create an on-demand reporting. They now need to enable real time KPI monitoring available and trigger alerts on any suspicious change especially post deployment.

QA department will stop doing manual QA, and start writing end to end automated tests. At Wix we initially had QA helping the server developers writing IT tests, however after about a year where we had a server QA we decided that we are proficient enough and well covered in automated tests that we do not need QA for the server side. We still use QA on the client side development (i.e UI), they do both automated and manual testing.

Team leads need to give a lot of freedom to the developers and help remove obstacles from their path to deploy.

Product We trained our product managers to think lean and feature centric and mandated that every feature is A/B tested. Developers and product managers sit in the same room and discuss the product during the whole development cycle. Today product managers open and close tests and feature toggles dozens of times a day and no longer assume they know what the users want, everything is tested.

Created a CI team that will develop the CI tooling and automate the staging environment creation. Today we can create staging environment with any version and install what ever services we choose with just a click of a button.

The most important change is actually going to Test Driven Development. No feature is considered production ready unless it is fully tested . Product managers, developers and management understand the importance of automated testing and we take that under consideration in our time estimates. We invest a lot in testing and always improving our testing framework.

Continuous Delivery - Part 8 - Deploying To Production

It is about time we talk about the actual continuous delivery process works, application lifecycle and how the code reaches production once development is done.

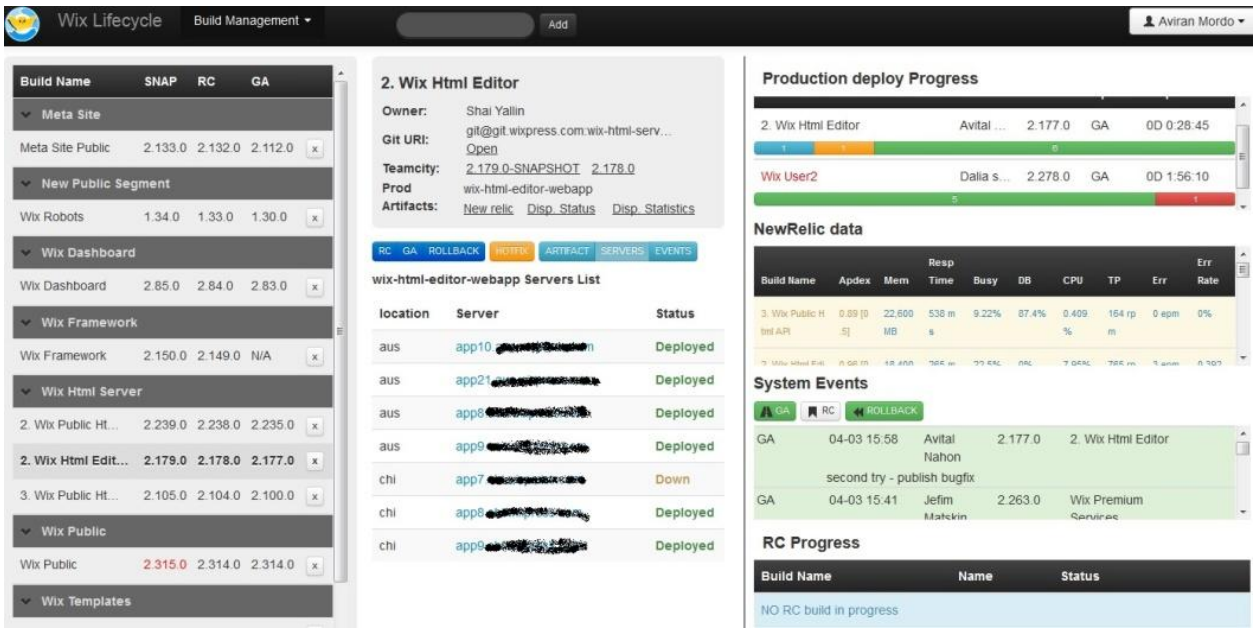
Phase 1: Git – Developers push the completed code (and the tests) to a Git repository.

Phase 2: Build – Team city (our CI build server) is triggered - checks out the code from Git; Runs the build, unit tests and integration tests. Once the build is passed a SNAPSHOT artifact is created and pushed to Artifactory.

So far the process was automatically. At this point we decided to go to a manual process in order to do the actual deployments. However this process should be as easy as pressing a button (which it actually is). We are now in the process of also automating deployment to staging making staging continuous deployment environment and production continuous delivery.

Phase 3: Deploy to staging (optional) – Once a snapshot artifact is in Artifactory, it is ready for deployment to staging servers. To manage and trigger a deployment we built our own command and control dashboard called "Wix Lifecycle" it connects to all our backend systems: Team City, Git, Noah, Chef, New Relic and our custom built reverse proxy/load balancers (called "Dispatcher).

Phase 4: Release candidate – To make a release candidate "Lifecycle" triggers Team City to do a special build using our custom build maven release plugin. The maven plugin copies the artifact from Snapshot to a separate repository while advancing the version number in the POM



Phase 5: Deploying to production (GA) – Using "Lifecycle" we trigger a production deployment. Lifecycle updates Noah with the artifact version we want to deploy. Chef agents are installed on all the production machines and every 15 minutes comparing the artifact version installed on the machine against the version that is configured in Noah (by lifecycle). If the versions do not match, Chef installs the configured version.

Phase 6: Installation – Since our services are always clustered the installation does not occur on all the cluster at once. Chef selects one server out of the cluster and installs the new artifact. Then Chef calls a special controller that exists on all the services built with our framework called "SelfTest". This controller triggers a self test (or startup test) on the service. Only after the self test passes, Chef continues to install the next server. If self test fails then the deployment is stopped until a human intervene, which he can force the next server deployment or roll back the current deployment. Services do not get traffic until their "is_alive" controller returns "true", and that will not happen unless Self test is passed.

Since during deployment we stop the service the load balancer will identify the instance is down and will not send traffic its way. In order for the instance to go back to the pool of services the Dispatcher is checking the service's "is_alive" controller which will return "false" until the "Self Test" is passed successfully.

Phase 7: Monitoring – After the deployment the person who pushed the deploy button is responsible to look at the monitoring systems for any unusual behavior. If we identify problem with the deployment a click of a button triggers a rollback (which is simply deploying the previous version.

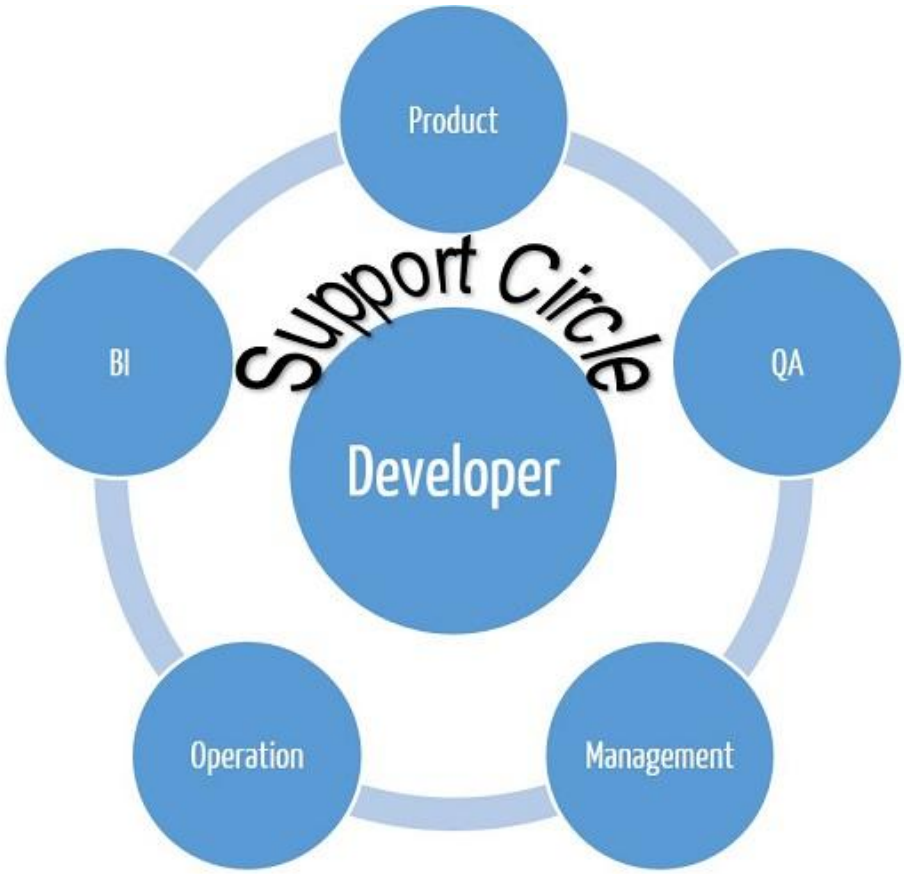
Many of the deployments do not deploy to all servers, but we also can select a "Test Bed", a single server that is deployed and runs for a while so we can compare it to the other servers which were not deployed. If everything is OK then we continue with the deployment to all the servers. Having a "test bed" server allows us to minimize problems to only one server out of the cluster.

Dev Centric Culture - Breaking Down The Walls

We have been doing Continuous Delivery at Wix for several years now and as part of that, I have been training my team at the methodology of DevOps and changing the company's culture. Last week I was trying to explain to some colleagues how we work at Wix and about our dev culture. While doing it I realized that we are doing much more than just DevOps. What we actually do is something I call "**Dev Centric**" culture.

What is Dev Centric culture?

Dev Centric culture is about putting the developer in the middle and create a support circle around him to enable the developer to create the product you need and operate it successfully.



Before I'll go into details about Dev Centric culture and why is it good let's look on the problems we have, what we are trying to solve and why it is more than just DevOps.

DevOps is about breaking the walls between developers and operations but after few years of doing DevOps and continuous delivery we realized that DevOps is not enough. The wall between the developers and the operation is just one wall out of many. If you really want to be agile (and we do) we need to break down ALL the walls.

Let's take a look at a traditional development pipeline.

- Product that "tells" engineering what they need to do.
- Software architect designs the system and "tells" the developer how to build it.
- Developers write the code (and tests in case the work in TDD)

- QA checks the developer's product and cover it with more test
- Operations deploy the artifact to production

So what we have here is a series of walls. A wall between the product and engineering, a wall between engineering and QA and of course a wall between engineering and operation. (Sometimes there is even a wall between architecture team and developers)

When a product manager hands over a spec to a developer, the developers usually don't really read the whole spec, they misinterpret the product manager's intention, they lack the product manager's visions and they are also disconnected from the client's needs.

When you have a team of architects doing all the design and tell developers what to do, you again have a wall because developers might not understand the system's architecture and will not build what the architect intended.

When developers hand over their product to QA you create another wall because now QA need to guess how the system should work and there is also a shift in responsibility where developers don't feel ownership and responsibility to the quality of their product.

Then comes the last phase where Operations get a hold of the product and they are responsible to deploy and monitor a product they know nothing about.

So when something bad happens there is no one who is actually responsible to the product and you start with a blame game.
If the product is not what the customer expected then product manager blames the developers that they did not develop the right thing. On the other hand developers blame the product managers for not defining the product clearly.
If there is a bug (got forbid) on production then developer blame QA for not finding the bug, and QA blame the developers for writing a buggy software in the first place.
If there are stability issues on production then everyone are pointing the figure to operations who do not maintain the product properly; on the other hand operation blame the developers for writing unstable software.

So as you can see there are many walls in the process that we need to break in order to have better quality.

So let's see how we can break these walls, how "Dev Centric" culture helps us with that.

Dev Centric culture means that you put the developer in the middle and create a support circle around him to enable the developer create the product you need and operate it successfully.
In a "Dev Centric" culture the developer is responsible through all the product lifecycle, from product definition to maintaining it in production.

Now if you think about a very small startup the developer is the key and is doing everything from talking to customer, define the product design the system, write the code, deploy and maintain it in production. This is also the phase where the startup is the most agile and progress very fast. Now if the startup succeeds and grow, more people join the company and slowly building walls around the different departments that eventually hurt productivity.

Now let's see in more details what walls do we have that we need to break down and how "Dev Centric" culture can help breaking it down and improve the quality and agility of the whole organization.

Wall #1: Product || Engineering

The Problem:

When getting a requirement to design a new feature or system, the product requirement is defined by the product manager. The product manager sits with the clients, understands their needs and interpret the needs to system requirements. The problem with that is that the product manager does not write the actual code and design the system. The one who is doing that is the developers. If developers get a design document, many times they don't understand the actual customer need, they miss-interpret the product manager thoughts (and between us, they don't usually read the whole product requirement document). This causes a waste of time because the developers might develop something completely wrong that don't meet the customer needs. If the product manager catch it on time the developers can fix the application before it gets to the customer, but it just wasted their time because they could have built is good in the first place.

In Dev Centric:

Instead of product manager tell the developer what to do, the developer sit down with the product manager and TOGETHER they define the product vision. This way the developer gets to understand the product and the customer's needs. By understanding the product, the developer will create a much better system that fits the client's requirements. If we'll add to that continuous delivery and "lean startup" concepts, the developer is a very good resource to help the product managers to define the MVP (Minimal Viable Product) and making sure that no feature is being develop that is not needed.

Wall #2: Architect || Developers

The problem:

In many organizations once the product manager finished with the product definition, the "ball" is coming to the architect's court. The architect designs the system and hands the design to the developers to develop the application. In this scenario the developers don't completely understands the architecture and may not produce what the architect intended. Another side effect for this is that we treat the developer as a coder and not as a software engineer, thus limiting their ability to cope with the complexity of designing a good product. This also create a big workload on the architect who now has to supervise the engineering work and make sure they follow the design.

In Dev Centric:

Instead of handing the designs to the developers, we need to invest in the developer's growth and to teach them how to design a good system. Architecture should be done by the developers themselves while the architect should become a consultant and a supervisor. There is a challenge here for the architects because in many cases there is not one correct architecture and the developer might choose a different architecture than the architect would have chosen. The architect should be open to accept different approach as long as it is a good one even if it is different from what he thinks. Architects should mentor the developers in how to design a system, teach them how to approach a problem, how to analyze it and guide the developers to think about the architecture concerns the developers are not aware of or missed.

By doing this you invest in your team and grow their knowledge, quality and abilities, thus creating a better products and better developers.

Do not be afraid let the developers make mistakes. As long as they learn from the mistakes you will end up with better engineers. If architect will feed the developers with the answers all the time the developers will not learn to cope with complex problems on their own and will come to the architect on every problem, thus creating a much bigger workload on the architects.

Wall #3: Developers || QA

The problem:

When developers finish their work and hand it over to QA they are being put in a state of mind that they hand over the responsibility of ensuring the quality of their work to someone else. Instead of owning the product that they have just wrote it is up to somebody else to find their bugs. This case causes developers to not invest too much in the quality of their code and not feel responsible for their

own work quality. Now when a bug is found on production the developers believe that it is the QA fault for not fining the bugs.

In Dev Centric:
The solution for this problem is really simple. You need to make your developers write tests and be responsible for their own work. The only way to do that is to start doing Test Driven Development. Developers who work in TDD methodology write better code, their designs is much better because it has to be testable, with the right abstractions and separation of concerns.
Now for developers who are not used to write test this can be a challenge, but from my experience, once they do that long enough the developers understand the benefit of TDD and never want to go back to not writing tests.

Developers should write both unit tests and integration tests. For back-end development this is usually enough, no QA is necessary unless in very rare cases where changes are very deep and might affect other system. At Wix the back-end group works without any QA.
Now you might think that if developers do their own testing why we need QA for. Well you still need QA but QA should change their role. Instead of verifying the developer's work quality they should complement the developer's tests by writing automated system tests (end-to end) and on some cases do manual QA for UI components during development until an automated test is created for them.

Wall #4: Developer || Operations
The problem:
In order to deploy the application the developers or QA hand over the artifact to operations and the operation people install or deploy the application. The problem is that the operation people know nothing about the application, what it does and how it should behave. Now we expect them to install it and also maintain the applications that they are not familiar with.

In Dev Centric:
The developer should be responsible for their own applications in production. The developer should install and maintain the applications with the support of the DevOps engineers. DevOps engineers should create the tools to help the developer install and maintain their applications on production. While developers should have ssh access to production, ops should create the tools so that developers will not have to ssh to production.
Make the production environment visible in a developer friendly way. Via monitoring system, easy access to logs (log aggregation). Developers should see how their system behaves in production at all times. This creates a better quality product because now the developers are also responsible to their work on production. Every time there is a problem on production the developer who is the owner of the rough process should be alerted and handle it on production with the help of the operation engineers.

Wall #5 Developer || BI
The Problem:
We see BI as a business concern and generate reports to marketing and management while the developers are not exposed to most of them. This creates a lack of knowledge by the developers of what is going on with his system and how it is used by the customers. If the developer does not know how his system is used he will not be able to make improvements on the parts that really matter.

In Dev Centric:
Expose the BI reports to the developers. Make the BI tools also accessible to developers so they can add also technical data and take advantage of the BI analytic tools to understand better how the system is used. By doing that the developer can have a sense of priorities and the knowledge of how the system is used in real life. Once developers understand that, they can make improvements and optimizations to the most heavily used parts of the system and by doing that help customer satisfaction.

Maintaining a Dev Centric culture is not easy. You give a lot of responsibility in the hands of the developers. You actually give all the responsibility to the developers. But if you think about it this is exactly what you should do. The developers are your company assembly line, they are the ones who actually create the product. They have a direct impact on the company's success and bottom line. By understanding that and entrusting the developers with the power to change and take critical decisions it is a win-win situation.

Remember developers CAN DO the work of a product manager, architect, QA and Ops, in fact this is exactly how a small startup succeeds, because developers actually do all of it.

What do you think is it time to go back to the future by breaking all the walls?