DevOps

Dr. Dobb's Journal

October 2013

Continuous Delivery

The First Steps

Next

ALSO INSIDE

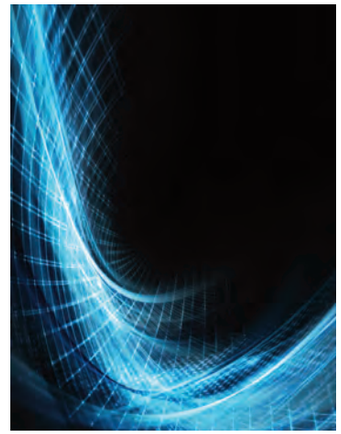
UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

October 2013



More on DrDobbs.com

Putting Absolutely Everything in Version Control
The key principle of continuous delivery — everything goes into the SCM — solves some big problems, but creates others.
http://www.drdobbs.com/240160762

Building RESTful APIs with Tornado
Tornado is a Python Web framework and asynchronous networking library that provides excellent scalability due to its non-blocking network I/O. It also greatly facilitates building a RESTful API quickly.
http://www.drdobbs.com/240160382

Theory Versus Practice: The Great Divide in Programming Languages
Do programs exist in order to tell computers what to do, or do computers exist in order to execute programs?
http://www.drdobbs.com/240160938

Making It In Embedded Systems
Just because a tool is old doesn't mean it won't do the job.
http://www.drdobbs.com/240160756

After Ballmer, What?
Microsoft's situation might be unique, but its inability to assess the present and the future correctly is shared by many vendors.
http://www.drdobbs.com/240160507

Guerilla Improvement: Getting Started in DevOps Without Buy-In

DevOps need not be a top-down mandate to succeed. With these steps, developers and operations staff can lead the way implementing DevOps techniques and later show the organizational benefits to management.

By Langdon White and Robyn Bergeron

As IT has become decentralized, silos have built up around various functions: UX, database, back-end, operations, requirements gathering, QE, and so on. Some of this segregation was addressed by agile processes and integrating requirements' owners directly into the development process. DevOps is the next logical step of integrating the people who deploy and care for the applications into the same process. But this, as with many things in IT, is easier said than done.

DevOps is at the point in its maturity where, for developers, it no longer needs an overview. But what the chatter around DevOps lacks is how to get started when you have no money, no buy-in, no directive, really "no anything" aside from a belief that a DevOps model will result in a better working and operating environment for you and your company.

As you might guess (judging from the companies we work for), we have a particular affinity for open source and leveraging not just open source software, but also the "open source way." This way of doing things generally starts with someone wanting to "scratch their own itch," then getting other people involved.

With DevOps, you can do the same. First, just start doing it; second, get other people to like what you are doing and participate; finally, make it integral to your organization, or as we often refer to it: "make your boss care."

Adding a community orientation to a DevOps implementation also means that there is almost always someone around to help you. While you are taking these steps, be sure to look for the wider "community of practice" in your neighborhood and on the Internet that you can join to ask questions of and learn from.

Step One: Start with Practices

Continuous improvement is a foundation of other quality processes, including kaizen, lean, and agile. It may sound cliché, but improving your organization's culture is an area where you can set an example — and others will follow.

Initiate and encourage more frequent communication within your team, and with other teams, whether through stand-up meetings, conference calls, or other methods. Work to identify the root causes of issues, and practice blameless failure: Don't blame the person, but fix the process so that it won't happen again. Foster a "release early, release often" attitude; this mantra is often used in open source communities as a method for transparency — everyone knows what is going on. The practice, however, has side benefits useful to any development group: Smaller code commits result in code that is easier to debug; early releases get you customer feedback as soon as possible.

"Don't keep your continuous improvement to yourself; share your methods with others. Building a community of practice in your organization is key to spreading your success"

At a more pragmatic level, we suggest:

- Start small, invite the Ops team members to your Dev meetings or vice versa, and ask to be invited to the other team's meetings.
- Pick one special snowflake and document it. A "special snowflake" is one of those things (processes, servers, etc.) that has some unique aspect that must be remembered.
- Use Puppet (or Chef, Ansible, Salt, etc.) on all your non-production machines and commit to modifying them only through configuration management.
- Create a kanban or similar notification/logistics board with all the things you (or your team) are working on, make sure everyone can see it, and just keep it up-to-date.
- "If it moves, measure it." Why? Continuous improvement doesn't happen in one step — it is a constant evolution. Logs are your friend, and keeping a visual dashboard can help you see bottlenecks or issues that might otherwise be buried in the numbers. Make sure to track your team's outstanding requests, your server downtime, and the number of applied patches. As you fix processes, see if you can pinpoint areas where your team's statistics improve as a proof point that can show everyone the light.
- Create a kanban or similar notification/logistics board with all the things you (or your team) are working on, make sure everyone can see it, and just keep it up-to-date.

Step Two: Gather Your Friends (and Make New Ones)

Don't keep your continuous improvement to yourself; share your methods with others. Building a community of practice in your organization is key to spreading your success.

- Have a weekly lunch meet-up to talk about DevOps concepts and practices, and show off not just your own successes, but

share what you've heard about other external practices and successes (and failures!) as well.

- Learn what others are doing. Check for a local DevOps meet-up or attend a DevOps Days event in your region. Join the mailing list of some of your favorite tools or languages, and listen up or even lend a hand. Obviously, the Internet has a host of DevOps-specific blogs, forums, and sites, all of which can be incredibly helpful for DevOps noobs and experts alike. There are several good books. We recommend starting with *The Phoenix Project* (http://is.gd/mH4yCl), a novel centering around the evolution of an IT organization from traditional, siloed teams to a high-functioning, well-oiled machine. Other great reads are T*he Lean Startup* (http://is.gd/HWRuAT) and *Continuous Delivery* (http://is.gd/Uj6Cao).
- Regularly attend a DevOps meet up and join some mailing lists.
- Start a book club geared towards DevOps

Step Three: Make Your Boss Care

Reaching DevOps zen ultimately requires fundamental, organizational changes: Breaking down the silos of departments to function more closely together, restructuring processes for accountability/escalation, and reconsidering where time is spent and what the priorities are. Unless you're in charge of your organization, you're going to have to sell the concept, and the best way to do that is by showing evidence of improvement in the areas that the company cares about and by bringing your internal community of practice to voice support.

Don't be surprised if you initially feel some resistance, since "the Devil you know versus the Devil you don't" is a safe place for many managers to sit. Look to understand your boss's pain points: Is it budget? Is it headcount? The skyrocketing number of requests?

"For full DevOps buy-in, bring your boss what he or she wants to hear, and follow up with all your other proof points"

Look back at your statistics and try to find the places where you're solving your boss's problems, or better yet, your department's or company's problems. Translate fewer outages into dollars saved, or show shorter times to deploy code to production.

For full DevOps buy-in, bring your boss what he or she wants to hear, and follow up with all your other proof points of improvement. Arm your boss to fight on your behalf based on what your internal community of practice already knows is working.

— Langdon White (http://twitter.com/1angdon) is a Red Hat Enterprise Linux developer advocate at Red Hat. Robyn Bergeron (http://twitter.com/robynbergeron) is the Fedora Project leader at Red Hat.

Comment

News Briefs

By Adrian Bridgwater

Syncfusion Reaches Out To JavaScript Developers

Syncfusion has released an update to Orubase, its framework for developing cross-platform hybrid mobile applications. The .NET component vendor says that at the forefront of the update are several JavaScript controls for visualizing data, and support for HTML and JavaScript-based web applications. With this latest update then we see that various editors, gauges, and charts are available as JavaScript controls. Developers have the choice to work with ASP.NET MVC or JavaScript when designing hybrid mobile apps, a choice unmatched (according to Syncfusion) in any other mobile application framework. Other new features in Orubase include Windows 8 and Windows Phone 8 support, the Orubase Project Wizard enhancement to create HTML and JavaScript-based hybrid apps, plus signature capture support to record handwriting through touch gestures on Android, iOS, and Windows devices.

http://www.drdobbs.com/240160571

Give Developers A Day Off From Data Management

The cross-platform data management app market expands this week as a result of dBase and its eponymously named dBaseAPPs. Designed specifically for non-IT professionals, dBaseAPPs allows users to manage their data (regardless of their platform) and without having to rely on IT. The firm is making dBaseAPPs available on both the Apple and Microsoft platforms — the product is targeted at small to medium sized business users. The first app in the dBaseAPPs series, dTransfer, enables users to copy data tables between databases via a user interface. By treating database tables like files in a folder, dTransfer copies tables between PostgreSQL, MySQL, and SQLite databases.

http://www.drdobbs.com/240160353

Scriptless Test Automation For iOS and Android

Automation Anywhere has launched Testing Anywhere MOBILE. This scriptless multi-platform test automation product for Android and iOS presents a set of test automation tools so developers can write a single test case and use it anywhere. Testing can be done on the devices themselves — instead of emulators — thereby (in theory) improving accuracy by eliminating environment-related errors. Key features of Testing Anywhere MOBILE include visual automation technology that automatically documents test cases as users create and edit, presenting images in a storyboard view. Testing Anywhere SMART automation object-based recording technology automatically translates what's being recorded into a code-level summary, allowing QA to use one test case across any device, browser, or platform.

http://www.drdobbs.com/240160937

Python Bite Is Without Venom

Software application development testing company Coverity has released a report indicating that Python has a markedly low "defect density" when compared to the industry average defect density for good quality software and types of defects identified. Coverity gauges that since 2006, Python has achieved a defect density of .005 (or .005 defects per 1,000 lines of code) and has eliminated all high-risk defects in its codebase. The Coverity Scan Project found an average defect density of .69 for open source software projects that happen to leverage the firm's own code scan service — this is compared to the accepted industry standard defect density for good quality software of 1.0.

http://www.drdobbs.com/240160744

Continuous Delivery: The First Steps

Continuous delivery integrates many practices that in their totality might seem daunting. But starting with a few basic steps brings immediate benefits. Here's how.

By Dave Jabs

Software development groups that achieve high performance in development and delivery provide a strategic advantage to their business. However, many organizations struggle with delivering software in a timely manner. The set of practices called "continuous delivery" is gaining favor as an important part of the work of delivering new software on time. Continuous delivery defines a set of practices that aim to eliminate mechanical impediments and deliver software with greater velocity to respond to market needs.

Continuous Delivery as a Pipeline

Let's start by outlining what continuous delivery is. One of my favorite definitions comes from Jez Humble of Thoughtworks, whose book is the seminal text on the discipline (http://is.gd/Et5gw6). He says, "The essence of my philosophy to software delivery is to build software so that it is always in a state where it could be put into production. We call this Continuous Delivery because we are continuously running a deployment pipeline that tests if this software is in a state to be delivered."

Continuous delivery is not defined as continuous deployment, continuous release, or something that only cloud applications need worry about. Continuous delivery uses a set of key principles that involve staging the software development and delivery process. Each stage in the process possesses a distinct set of criteria that must be validated before software can progress to the next stage. These processes leverage automation for testing, software deployment, and promotion. The end result of the stage-based process is the formation of a delivery pipeline through which new software can continuously flow, and as it flows, it is validated to progressively higher levels of quality.

Because continuous delivery is not a tool or product, achieving it is difficult. The approach to a successful continuous delivery process is a

holistic one, where software development organizations must align technology, process, people, and values. It means a mindset shift for teams unused to it and, frequently, changes to processes and tools. For continuous delivery to be done properly, it also requires organizational changes. Achieving excellence in continuous delivery isn't easy, but the reward is enormous: Development teams will be able to move at velocities not previously thought possible.

Steps to Continuous Delivery

The overarching concept of continuous delivery is not new; in fact, the first agile principle states, "the highest priority is to satisfy the customer through early and continuous delivery of valuable software." Despite this familiarity, many companies — even those committed to agile processes — struggle to just get started.

The roadmap to achieve continuous delivery begins with modeling the development and delivery pipeline. Many of today's organizations are formed from disconnected teams. If you want continuous delivery, you need continuous flow from beginning to end.

Achieving continuous flow requires a multi-step approach whereby organizations:

- Model and measure the cycle time in your existing processes
- Identify delays in the process
- Develop action plans to minimize current delays and eliminate potential future delays

Following this process allows organizations to break down the silos between development and operations in order to produce a single, unified pipeline ranging from concept all the way through delivered software.

Continuous delivery is a journey, not something that is achievable overnight, because development organizations must make significant changes, often to technology, process, culture, and people, to implement the full pipeline — and these changes take time, often more than a year.

Where should software development organizations start? Here are three fundamental areas where companies should focus their initial efforts.

- *Requirements management:* Requirements must be decomposed to minimum deliverable units; development plans should be created and organized around incremental delivery.
- *Testing/Validation:* The QA process has to be empirical, structured and organized, as well as automatable. In addition, good continuous integration is a prerequisite for continuous deployment.
- *Delivery Mechanics:* Start with version control. Reproducibility of software builds is a common practice, however reproducibility of deployment is often a missing element. (For more on this topic, see "Putting Absolutely Everything in Version Control" at http://www.drdobbs.com/240160762) Then, automate delivery. Keep at it until it's a one-click frictionless process.

Barriers to Successful Continuous Delivery

Most software development organizations face what appear to be overwhelming barriers to achieving continuous delivery. They find themselves in this perceived situation because their existing legacy products have been developed without following continuous delivery principles. The good news: They can start now. Companies can begin defining their goals and objectives and put practices in place that allow them to make small gains along the way to their ideal situation.

Here are a few common areas where companies seem to have the biggest struggles with regard to implementing effective continuous delivery strategies.

Requirements Management

Companies often find themselves struggling to define small deliverable stories that are easily understood, managed, and delivered. The solution to this problem is to set goals to make each story smaller, and then focus on continually improving the story writing process until stories can be finished with very short cycle times. Feature toggles can be used to hide stories until a complete set of related stories are completed, or to allow A/B testing of the usability of a set of stories.

Focusing on the Wrong Tests

The state of software development testing today sees most organizations investing heavily in GUI testing. This practice is actually an impediment. Organizations should put the majority of test focus on unit testing, where test coverage is at its greatest. From there, focus should next be placed on integration testing, in which a small number of integration tests are used to ensure all the pieces work together. GUI testing should have the least amount of focus because GUI tests are often brittle, slow, unreliable, and the most expensive to maintain for the value they deliver. The result of this testing strategy will yield faster cycle time, higher quality, and lower test maintenance costs.

Scaling Continuous Integration

Another frequent problem is scaling continuous integration. As development scales, you get many more concurrent changes in progress, builds take longer and tests take longer, resulting in broken builds. In

the worst case, builds on a single mainline become mostly broken instead of mostly working. That is not continuous delivery. Worse yet, the unstable main line affects all developers and slows down their productively dramatically.

There are two primary techniques companies employ to achieve continuous delivery at scale. Each of these techniques is designed as a "divide and conquer" strategy, allowing for focus on specific areas which can have significant impact on continuous delivery:

- Component or Service Architecture: Companies can choose to change their software to a component or service architecture. These are components that can be individually deployed in production independently of other components. They have clean interfaces and are compatible across multiple releases of components. This structure provides scalability because you can then run separate development and delivery pipelines that don't interfere with each other.
- Multi-Team Continuous Integration: In this approach, scalability is achieved by doing development using separate teams. You then join the work from those teams as they go through integration and test processes, and eventually, out to the deployment pipeline. A key to multi-stage continuous integration is to break down the development pipeline and integration testing into stages. Organizations may define these stages by team branch, feature branch, build (nightly, weekly, etc.), and main line. Using integration testing along each of these stages provides a level of test coverage and continuity that allows code to flow up the pipeline while also identifying issues faster and closer to the developer, where the cost to fix the issue is less expensive.

An example of successful scaling continuous delivery I have seen is with an international online gaming company with 70 developers organized into 10 teams. One of the keys to their success was the change management process they have incorporated into their continuous integration practices. Due to the nature of the gaming business, inter-product dependencies cause a rate of change per product on a daily basis that is sometimes as high as 60%. Using effective continuous integration and change management techniques, this company was able to handle this rate of change, and as a result, is producing more than 3500 releasable builds per month. This company fully embodies the continuous delivery principle of always having releasable code.

Conclusion

Using continuous integration and continuous delivery effectively can dramatically improve the way your engineering team operates, as well as the business overall. Companies following these best practices will gain significant competitive advantages in software development time-to-market, agility, and quality. The key to being successful with continuous delivery is to start now. The faster your organization can begin avoiding the pitfalls of lengthy development cycles where engineering teams are stifled by manual processes, the faster it will realize the benefits of continuous delivery.

— Dave Jabs is the CTO of AccuRev and a strong proponent of continuous delivery.

Comment

Driving Continuous Integration from Git

Testing, code coverage, style enforcement are all check-in and merge requirements that can be automated and driven from Git.

By Sarah Goff-Dupont and Tim Pettersen

If you're among the rising number of Git users out there, you're in luck: You can automate pieces of your development workflow with Git hooks. Hooks are a native Git mechanism for firing off custom scripts before or after certain operations such as `commit`, `merge`, `applypatch`, and others. Think of them as henchmen for your Git repo. Pre-operation hooks act as bouncers, guarding your repo with a velvet rope. And post-operation hooks are your Man Friday, faithfully carrying out follow-up tasks on your behalf.

Installing hooks for a Git repository is fairly straightforward, and well-documented (http://is.gd/a7JnZn). In this article, we focus on using Git hooks to augment continuous integration practices, starting with an example that makes combining Git and continuous integration (CI) less painful. The code is written in Ruby. Fortunately, Ruby is a language that highly prizes readability, so even if you don't know Ruby, you can easily follow along.

Automate CI Configuration for Git Branches

One of the blessings of Git is how easy it is to branch off and develop in isolation. This means the master stays releasable, you get the freedom to experiment, and your teammates aren't derailed if code from the experimentation proves to be half-baked. One challenge of Git, however, is how many branches a team ends up with — scores of active branches, most of which live for only a few days. Who is going to take the time to set up continuous integration for all those piddly little branches? Your henchmen, that's who.

To automatically apply CI to new development branches, you'll use the "post-receive" hook type. These are server-side hooks, triggered af-

ter pushes to the repository complete. In such cases, you can use the post-receive hook to fire off a script that clones a master's CI configs and applies them to new branches using the CI server's exposed API. It might look something like this, when using the open-source and hugely popular Jenkins (http://jenkins-ci.org/) CI server:

```
#!/usr/bin/env ruby

 # Ref update hook for creating new Jenkins job
 # configurations for newly pushed branches.
 #
 # requires Ruby 1.9.3+

 require 'yaml'
 require 'net/https'
 require 'uri'
 require 'rexml/document'
 include REXML

 # load ci-config.yml from hook directory
 def load_config
   hookDir = File.expand_path File.dirname(__FILE__)
   configPath = hookDir + "/ci-config.yml"
   puts configPath
   raise "No ci-config.yml found." unless File.exists?
     configPath
   YAML.load_file(configPath)
 end

 # Grab the configured Jenkins server
 config = load_config
 raise "ci-config.yml file is incomplete: missing
   jenkins_server" unless
   config["jenkins_server"]
 server = config["jenkins_server"]
 raise "ci-config.yml file is incomplete: username,
     password, url and
   default_job are required for jenkins_server" unless
     server['url'] and server['username'] and
     server['password'] and server['default_job']

 # iterate through updated refs looking for new branches
 ARGF.readlines.each { |line|
   args = line.split
   oldVal = args[0]
   newVal = args[1]
   ref = args[2]

   if /^0{40}$/.match(oldVal) and
       ref.start_with?("refs/heads/")
     # new branch!
     # retrieve the jenkins job config
     # TODO only need to do this once!
     uri = URI.parse(
      "#{server['url']}/job/#{server['default_job']}/
         config.xml")
     req = Net::HTTP::Get.new(uri.to_s)
     req.basic_auth server['username'], server['password']
```

```
    http = Net::HTTP.new(uri.host, uri.port)
    http.verify_mode = OpenSSL::SSL::VERIFY_NONE
    http.use_ssl = uri.scheme.eql?("https")

    # execute the request
    response = http.start {|http| http.request(req)}

    raise "Bad response from jenkins, is your ci-config.
      yml correct?"
      unless response.is_a? Net::HTTPOK

    # parse the config.xml from the response
    doc = Document.new response.body
    doc.root.get_elements(
      "//branches/hudson.plugins.git.BranchSpec/name").
         each {
        # overwrite branch to be our new ref
        |elem| elem.text = ref
    }

    # create a new request to upload the modified config.xml
    newJob = ""
    doc.write newJob

    newJobName = ref["refs/heads/".length..-1].
     gsub("/", "-")
    uri = URI.parse("#{server['url']}/
    createItem?name=#{newJobName}")
    req = Net::HTTP::Post.new(uri.to_s,
    initheader = {'Content-Type' => 'application/xml'})

    req.basic_auth server['username'], server['password']
    req.body = newJob

    # upload the new job
    response = http.start {|http| http.request(req)}
    raise "Failed to post new job to jenkins" unless
      response.is_a? Net::HTTPOK
  end
}
```

With this hook in place, you need only push a dev branch to the repo, and it will automatically be put under test. (It's possible to run CI builds against branches using a `build` parameter to represent the target branch, but that muddles the build history. The cloning approach provides a clean, clear history.) Applying every last facet of the CI scheme to branches isn't necessary — for example, running each and every branch through the load test gamut might be overkill. But even if you skip the load and UI tests, and run just unit and API- or integration-level tests, these are huge wins.

The risk of introducing defects into master is greatly reduced by testing on the branch before merging. Developers can also work more efficiently and confidently because of the frequent feedback on changes (instead of the old merge-then-pray technique). And for teams who include testing as part of their definition of "done," managers and scrum master types catch a break. With the Git hook automatically putting branch code under test, the team's practices and values are being enforced without the need for nag-mails or raised eyebrows during stand-up.

Vet Merges to Master

Two hallmarks of coding craftsmanship are an affinity for automated tests, and adherence to stylistic rules (such as avoiding empty `try/catch` blocks or duplicated code). Despite best intentions, everyone neglects best practices from time to time. That's where Git hooks come in. Pre-receive hooks living in the central repository qualify incoming pushes, making sure they're good enough to get past the velvet rope. Let's look at three hooks designed to protect master from slip-ups made on development branches.

Require Passing Branch Builds

The whole point of working on a development branch is to isolate yourself and create a space to experiment (read: "break stuff"). So it's natural to see failing tests on the branch while development is in progress. When it's time to merge to master, however, things had better be tidied up. This can be enforced programmatically with a hook that checks to see whether the incoming push is a merge to master, and if so, verify that all tests are passing on the branch before processing the merge.

If you happen to be using Bamboo, you can cleanly fetch test results for a given commit. If you use Jenkins or its predecessor, Hudson (http://www.hudson-ci.org/), you can fetch a set of recent build results then parse through them to see which builds ran against the commit in question. (This hook, and those that follow are implemented for the Bamboo CI server, but they can be implemented in more or less the same way on all CI servers.)

```
#!/usr/bin/env ruby

 # Ref update hook for verifying the build status of
 # a topic branch being merged into
 # a protected branch (e.g. master) from a Bamboo server.
 #
 # requires Ruby 1.9.3+

 require_relative 'ci-util'
 require 'json'

 # parse args supplied by git: <ref_name>
   <old_sha> <new_sha>
 ref = simple_branch_name ARGV[0]
 prevCommit = ARGV[1]
 newCommit = ARGV[2]

 # test if the updated ref is one we want to enforce green
 # builds for exit_if_not_protected_ref(ref)

 # get the tip of the most recently merged branch
 tip_of_merged_branch =
    find_newest_non_merge_commit(prevCommit, newCommit)

 # parse our Bamboo server config
 bamboo = read_config("bamboo", ["url", "username",
  "password"])

 # query Bamboo for build results
 response = httpGet(
    bamboo,

"/rest/api/latest/result/byChangeset/#{tip_of_merged_branch
  }.json")
 body = JSON.parse(response.body)
 # tally the results
 failed = successful = in_progress = 0
 body['results']['result'].collect { |result|
   case result['state']
   when "Failed"
     failed += 1
   when "Successful"
     successful += 1
   when "Unknown"
     if result['lifeCycleState'] == "InProgress"
     in_progress += 1
     end
   end
 }

 # display a short message describing the build status for
 #the merged branch and abort if necessary
 if failed > 0
   # at least one red build - block the branch update
   abort "#{shortSha(tip_of_merged_branch)} has #{failed}
   red #{pluralize(failed, 'build', 'builds')}."
 elsif in_progress > 0
   # at least one incomplete build - block the branch update
     abort "#{shortSha(tip_of_merged_branch)} has
```

```
  #{in_progress}
  #{pluralize(in_progress, 'build', 'builds')} that have
    not
  completed yet."
else
  # all green builds - allow the branch update
  puts "#{shortSha(tip_of_merged_branch)} has
        #{successful}
    green #{pluralize(successful, 'build', 'builds')}."
end
```

Enforce Code Coverage Requirements

Along with successful test runs, you want to make sure that new code added on development branches is tested as thoroughly as code already on master. This ensures that the overall test coverage level of the project doesn't drop when a development branch is merged back in. This, too, can be checked with Git hooks.

A simple Git hook can verify that coverage on the branch meets the minimum threshold. To enforce this, a hook can be created to compare the coverage rate on master with that of the branch, and reject the merge if the branch's coverage is inferior.

Most CI servers don't expose code coverage data through their remote APIs. But there's an easy work-around: pulling down the code coverage report. To do this, the build must be configured to publish the report as a shared artifact, both on master and on the branch build. (Notice how automatically cloning build configs for development branches comes in handy here: Set it up for master, and get it on the branch for free!) Once published, you can get the latest coverage report from master by a call to the CI server. For branch coverage, you can fetch the coverage report either from the latest build, or for builds related to the reference (commit) being merged, as shown here for the code coverage tool Clover.

```
#!/usr/bin/env ruby

# Ref update hook for asserting the code coverage of a
# topic branch being merged into a
# protected branch (e.g. master) is the same or better
#
# requires Ruby 1.9.3+

require_relative 'ci-util'
require 'rexml/document'

include REXML

# Determine the code coverage for a particular commit by
# parsing Clover artifacts
def find_coverage(bamboo, commit)
  # grab the clover.xml artifact from the build.
  # This (assumes a shared artifact named
  # 'clover' with 'clover.xml' at the root).
  # Change this for your coverage tool's report name.
  clover_xml = shared_artifact_for_commit(bamboo, commit,
    bamboo["coverage_key"], "clover/clover.xml")
  doc = Document.new clover_xml

  # parse out the project metrics element from the response
  metrics = XPath.first(doc, "coverage/project/metrics")

  # Use algorithm similar to Clover
  # (https://confluence.atlassian.com/x/LoHEB) for
  # determining coverage percentage
  covered_elements =
    metrics.attribute("coveredconditionals").value.to_i
  covered_elements +=
    metrics.attribute("coveredmethods").value.to_i
  covered_elements +=
    metrics.attribute("coveredstatements").value.to_i

  elements = metrics.attribute("conditionals").value.to_i
  elements += metrics.attribute("methods").value.to_i
  elements += metrics.attribute("statements").value.to_i
```

```
    coverage = 0
    if (elements > 0)
      coverage = covered_elements / elements
    end
    coverage
  end

  # parse args supplied by git: <ref_name> <old_sha>
<new_sha>
  ref = simple_branch_name ARGV[0]
  prevCommit = ARGV[1]
  newCommit = ARGV[2]

  # test if the updated ref is one we want to enforce
  # green builds for  exit_if_not_protected_ref(ref)

  # get the tip of the most recently merged branch
  tip_of_merged_branch =
    find_newest_non_merge_commit(prevCommit, newCommit)

  # parse our bamboo server config
  bamboo = read_config("bamboo",
    ["url", "username", "password", "coverage_key"])

  # calculate code coverage for the old and new commits
  prev_coverage = find_coverage(bamboo, prevCommit)
  new_coverage = find_coverage(bamboo, tip_of_merged_branch)

  # if the coverage has dropped for the new commit,
    block the update
  if prev_coverage > new_coverage
    abort "Code coverage for
      #{shortSha(tip_of_merged_branch)} is
      only #{new_coverage}! #{ref} is currently at
      #{prev_coverage}."
  else
    # if the coverage has increased, TFCIT
    puts "Nice work! Code coverage for #{ref} has
      increased by #{new_coverage - prev_coverage}."
  end
```

Enforce Good Coding Style

Tests are something no self-respecting software project can do without, but they only tell part of the story. Open source tools such as Checkstyle (http://checkstyle.sourceforge.net/) and Findbugs (http://findbugs.sourceforge.net/) scour your codebase and provide reports on stylistic violations — anything from duplicated code to excessively long methods to the use of deprecated methods. These are hard-won guidelines, and they exist for a reason: Ignoring them can result in code being harder to understand, harder to maintain, and more vulnerable to runtime problems.

As with code coverage, each team has a different level of tolerance for unstylish code. But introducing more style violations is almost universally agreed-upon as undesirable. In this, Git hooks come to the rescue. Build artifacts come into play here as well since you can easily retrieve the violations report. (No CI server we're aware of exposes static analysis data via remote access API.) So you can create another pre-receive hook that checks violations for master and the dev branch, and rejects the push if it would introduce additional errors into the master.

```
#!/usr/bin/env ruby

 # Ref update hook for asserting that a topic branch
 # being merged into a protected
 # branch (e.g. master) does not introduce an increase in
 # checkstyle violations
 #
 # requires Ruby 1.9.3+

 require_relative 'ci-util'
 require 'rexml/document'

 include REXML
```

```
# This example
def count_checkstyle_violations(bamboo, commit)
  # grab the checkstyle.xml artifact from the
  # build (assumes a shared artifact named
  # 'checkstyle' with 'checkstyle-result.xml' at the root)
  checkstyle_xml =
    shared_artifact_for_commit(bamboo, commit,
    bamboo["checkstyle_key"],
     "checkstyle/checkstyle-result.xml")
  doc = Document.new checkstyle_xml
  # could go to town on the comparison here - but let's
    just count
  # the raw number of errors for the time being
  XPath.match(doc, "//error").length
end

# parse args supplied by git: <ref_name> <old_sha>
  <new_sha>
ref = simple_branch_name ARGV[0]
prevCommit = ARGV[1]
newCommit = ARGV[2]

# test if the updated ref is one we want to enforce green
  builds for
exit_if_not_protected_ref(ref)

# get the tip of the most recently merged branch
tip_of_merged_branch =
   find_newest_non_merge_commit(prevCommit, newCommit)

# parse our bamboo server config
bamboo = read_config("bamboo",
   ["url", "username", "password", "checkstyle_key"])

# calculate number of checkstyle violations for
#the old and new commits
prev_violations =
  count_checkstyle_violations(bamboo, prevCommit)
new_violations =
  count_checkstyle_violations(bamboo, tip_of_merged_branch)

# if the number of checkstyle violations has increased,
  block the update
if prev_violations > new_violations
  abort "#{shortSha(tip_of_merged_branch)}
     has #{new_violations} checkstyle violations! #{ref}
     currently has only #{prev_violations}."
else
  # if the number of checkstyle violations has
  # decreased, send kudos to the dev
  puts "Nice work! #{ref} has #{new_violations -
  prev_violations}
    fewer checkstyle violations than before."
end
```

"Checking for violations at merge time is great, pre-commit hook analyzing the changeset keeps the style police off your back entirely"

Think Globally, Hook Locally

We know that the sooner an issue is discovered, the easier (and faster and cheaper) it is to fix. That's why hooks that operate on local clones of a repository are so useful: They offer immediate feedback. Because we don't get the `cmd` prompt back until a hook completes, client-side hooks should be limited to operations that take only a few seconds, lest the development flow be interrupted. Let's look at two hooks that complete almost instantly.

Get Branch Build Status

Exposing branch build status in the terminal window with a post-checkout hook catches two fish with one worm: It provides actionable information, and eliminates the need to switch applications to get it. Upon checkout (and remember, in Git "checkout" means switching branches, not pulling down code as with SVN and Per-

force), this hook grabs the branch's head revision number from the local copy. It then queries the CI server to see whether that revision has been built, and if so, whether the build succeeded.

```
#!/usr/bin/env ruby

 # post-checkout hook for determining the build status of
the
 # checked out ref from the CI server.
 #
 # Requires Ruby 1.9.3+

 require 'yaml'
 require 'json'
 require 'net/https'
 require 'uri'

 # utility for correctly pluralizing quantities
 def pluralize count, single, multiple
   count == 1 ? single : multiple
 end

 # parse args supplied by git
 ref = ARGV[1]     # ref being checked out
 isBranch = ARGV[2]  # 0 = file checkout, 1 = branch
   checkout

 # we only care about branch checkouts
 if isBranch == "1"
   # initialise build status counts
   failed = successful = in_progress = 0

   # loop through each configured Stash server,
   retrieving build
   # statuses for the checked out commit and
    # counting the number of failed, successful and in
progress builds
   hookDir = File.expand_path File.dirname(__FILE__)
   configPath = hookDir + "/bamboo-config.yml"
   raise "No bamboo-config.yml found." unless File.exists?
     configPath
config = YAML.load_file(configPath)
raise "bamboo-config.yml file is incomplete:
  username, password & url are required" unless
  config['url'] and config['username'] and
  config['password']

# normalize base url
baseUrl = config['url']
# assume https if no scheme spcified
if not baseUrl.start_with? "http"
baseUrl = "https://#{baseUrl}"
end
# strip trailing slashes
while baseUrl.end_with? "/"
baseUrl = baseUrl[0..-2]
end

# prepare a request to hit the build status REST
  end-point
build_status_resource =
  "#{baseUrl}/rest/api/latest/result/byChangeset"
uri = URI.parse("#{build_status_resource}/#{ref}")
req = Net::HTTP::Get.new(uri.to_s, initheader =
  {'Content-Type' => 'application/json',
  'Accept' => 'application/json'})
req.basic_auth config['username'], config['password']
http = Net::HTTP.new(uri.host, uri.port)
http.verify_mode = OpenSSL::SSL::VERIFY_NONE
http.use_ssl = uri.scheme.eql?("https")

# execute the request
response = http.start {|http| http.request(req)}

if not response.is_a? Net::HTTPOK
puts 'An unknown error occurred while querying
  Bamboo for build results.'
exit
else
# if the request succeeded, count
# the statuses from the response
body = JSON.parse(response.body)
body['results']['result'].collect { |result|
```

```
    case result['state']
    when "Failed"
    failed += 1
    when "Successful"
    successful += 1
    when "Unknown"
    if result['lifeCycleState'] == "InProgress"
      in_progress += 1
    end
    end
  }
  end

  # display a short message describing the build status
  # for the checked out commit
  shortRef = ref[0..7]
  if failed > 0
  puts "Warning! #{shortRef} has #{failed}
     red #{pluralize(failed, 'build', 'builds')}
     (plus #{successful} green and #{in_progress}
     in progress).\nDetails: #{uri}"
  elsif successful == 0
    puts "#{shortRef} hasn't built yet."
  else
     puts "#{shortRef} has #{successful} green
     #{pluralize(successful, 'build', 'builds')}."
  end

end
```

If, for example, the hook tells you the head commit on the master has built successfully, then it's a "safe" commit to create a feature branch from. Or let's say the hook says the build for that revision failed, yet the team's wallboard shows a green build for that branch (or vice versa). That means the local copy is out-of-date. Whether to pull down the updates is determined on a case-by-case basis. This hook and its configuration files can be found at http://is.gd/ZDl6Sm.

Sanity-Check Code Style

Checking for violations at merge time is great, but a pre-commit hook analyzing the changeset keeps the style police off your back entirely. Start by capturing the names of files being updated or added and concatenating them. That string of file names is then passed into the Checkstyle `run` command. If violations are found, the commit is rejected.

Note that despite variations between them, all static analysis tools can be used with this approach. Findbugs, for example, must be run on the entire project because it looks at methods referenced across classes. But that's not necessarily a deal-breaker. Small and medium-sized projects can be fully analyzed quickly, especially if a generous heap space is allocated to the process.

Come As You Are

All the ideas presented here are vendor-neutral. Git hooks may not revolutionize software development the way continuous integration has, but every time a task, practice or rule is automated, it's a win.

— Tim is a developer at Atlassian, while Sarah is a former test automator and scrum master who now works in product marketing.

Comment

From the Vault

Top 10 Practices for Effective DevOps

Adopt these DevOps practices to realize the goals of effective collaboration, smoother operations, and cleaner code.

By Scott Ambler

DevOps (http://www.drdobbs.com/240009147) has become one of our industry's most popular buzzwords. Yet, surprisingly, there is little consensus as to what DevOps means beyond the high-level vision of tighter and more effective collaboration between development teams and operations teams. While DevOps might mean different things to different organizations, there is an emerging core of best practices that further its goals of enhanced collaboration to produce better software. I examine these practices here. Fair warning, though, I'm not just looking at this issue from the point of view of developers.

I've listed these items in priority order, with later practices often depending on those that come before.

Practice 1: Active Stakeholder Participation

A fundamental philosophy of DevOps is that developers, operations staff, and support people must work closely together on a regular basis. An implication is that they must see one other as important stakeholders and actively seek to work together. A common practice within the agile community is "onsite customer," adopted from Extreme Programming (XP), which motivates agile developers to work closely with the business. Disciplined agilists take this one step further with the practice of active stakeholder participation, which says that developers should work closely with all of their stakeholders, including operations and support staff--not just business stakeholders. This is a two-way street: Operations and support staff must also be willing to work closely with developers.

Practice 2: Automated Testing

Agile software developers are said to be "quality infected" because of their focus on writing quality code and their desire to test as often and early as possible. As a result, automated regression testing is a common practice adopted by agile teams, which is sometimes extended to test-first approaches such as test-driven development (TDD) and behavior-driven development (BDD). Because agile teams commonly run their automated test suites many times a day, and because they fix any problems they find right away, they enjoy higher levels of quality than teams that don't. This is good news for operations staff that insists a solution must be of sufficient quality before approving its release into production.

Practice 3: Integrated Configuration Management

With an integrated approach to configuration management (CM), development teams not only apply CM at the solution level as is customary, they also consider production configuration issues between their solution and the rest of your organization's infrastructure. This can be a major change for some developers because they're often used to thinking about CM only in terms of the solution they are currently working on. In a DevOps environment, developers need to be enterprise-aware and look at the bigger picture. How will their solution work with and take advantage of other assets in production? Will other assets leverage the solution being developed? The implication is that development teams will need to understand, and manage, the full range of dependencies for their product. Integrated configuration management enables operations staff to understand the potential impact of a new release, thereby making it easy to decide when to allow the new release to occur.

Practice 4: Integrated Change Management

From an IT perspective, change management is the act of ensuring successful and meaningful evolution of the IT infrastructure to better support the overall organization. This is tricky enough at a project-team level because many technologies, and even versions of similar technologies, will be used in the development of a single solution. Because DevOps brings the enterprise-level issues associated with operations into the mix, an integrated change management strategy can be far more complex, due to the need to consider a large number of solutions running and interacting in production simultaneously. With integrated change management, development teams must work closely with operations teams to understand the implications of any technology changes at an organization level. This approach depends on the earlier practices of active stakeholder participation, integrated configuration management, and automated testing.

Practice 5: Continuous Integration

Continuous integration (CI) is the discipline of building and validating a project, through automated regression testing and sometimes code analysis whenever updated code is checked into the version control system. CI is one of the sexier agile development practices (at least from a developer's perspective) that is typically associated with DevOps. CI enables developers to develop a high-quality working solution safely in small, regular steps by providing immediate feedback on code defects.

Practice 6: Integrated Deployment Planning

From the point of view of development teams, deployment planning has always required interaction with an organization's operations staff;

in some cases, via liaison specialists within operations typically called release engineers. Experienced development teams will do such planning continuously throughout construction with active stakeholder participation from development, operations, and support groups. When you adopt a DevOps strategy, you quickly realize the need to take a cross-team approach to deployment planning due to the need for operations staff to work with all of your development teams. This isn't news to operations staff, but it can be a surprise to development teams accustomed to working in their own siloed environments. If your team is not doing this already, you will need to start vying for release slots in the overall organizational deployment schedule. Furthermore, to support continuous deployment, release engineers will need to increase the number of release slots available to agile teams that are disciplined enough to continuously and consistently meet the quality requirements for release.

"Continuous deployment enables development teams to reduce the time between a new feature being identified and being deployed into production."

Practice 7: Continuous Deployment

Continuous deployment extends the practice of continuous integration. With continuous deployment, when your integration is successful in one sandbox, your changes are automatically promoted to the next sandbox, and integration is automatically started there. This automatic promotion continues until the point where any changes must be verified by a person, typically at the transition point between development and operations.

Continuous deployment enables development teams to reduce the time between a new feature being identified and being deployed into production. It enables the business to be more responsive. However, continuous deployment increases operational risk by increasing the potential for defects to be introduced into production when development teams aren't sufficiently disciplined. Successful continuous deployment in an enterprise environment requires all the practices described earlier.

Practice 8: Production Support

In enterprise environments, most application development teams are working on new releases of a solution that already exists in production. Not only will they be working on the new release, they will also have the responsibility of addressing serious production problems. The development team will often be referred to as "level three support" for the application because they will be the third (and last) team to be involved with fixing critical production problems. Although the need to do level three production support is common, with the exception of Kanban and Disciplined Agile Delivery (DAD), many agile methods only address this effort in passing. An important side effect of this practice is that it gives developers an appreciation of the kinds of things that occur in production, providing them with learning opportunities to improve the way that they design solutions in the first place.

Practice 9: Application Monitoring

As the name suggests, this is the operational practice of monitoring running solutions and applications once they are in production. Technology infrastructure platforms such as operating systems, application servers, and communication services often provide monitoring capabilities that can be leveraged by monitoring tools (such as Microsoft Management Console, IBM Tivoli Monitoring, and jManage). However, for monitoring application-specific functionality, such as what user interface (UI) features are being used by given types of users, instrumentation that is compliant with your organization's monitoring infrastructure will need to be built into the applications. Development teams need to be aware of this operational requirement or, better yet, have access to a framework that makes it straightforward to provide such instrumentation.

Practice 10: Automated Dashboards

The practice of using automated dashboards is business intelligence (BI) for IT. There are two aspects to this, development intelligence and operational intelligence. Development intelligence requires the use of development tools that are instrumented to generate metrics; for example, your configuration management (CM) tools already record who checked in what and when they did it. Continuous integration tools could similarly record when a build occurred, how many tests ran, how long the tests ran, whether the build was successful, how many tests we successful, and so on. This sort of raw data can then be analyzed and displayed in automated dashboards. Operational intelligence is an aspect of application monitoring discussed previously. With automated dashboards, an organization's overall metrics overhead can be dramatically reduced (although not completely eliminated because not everything can be automated). Automated dashboards provide real-time insight to an organization's governance teams.

DevOps Is Really About Culture

After describing these critical practices which support DevOps, I feel the need to emphasize that the primary critical success factor is to build a collaborative and respectful culture across your entire IT organization. My experience is that people, and the way that they work together, are the primary determinant of success when it comes to adopting an effective DevOps strategy. Unfortunately, it is considerably more difficult to bring about cultural change in an organization than it is to adopt a handful of new practices. More on this in future articles.

Additional Information

What Exactly is DevOps? (www.drdobbs.com/240009147)explores why DevOps is important for developers.
Getting DevOps Right (www.drdobbs.com/240062639) describes some of the challenges associated with adopting DevOps strategies.
Disciplined Agile Change Management (www.drdobbs.com/240001474) discusses change management options.
Disciplined Agile Delivery (http://disciplinedagiledelivery.com/) features more information about the DAD process framework.

— *Scott Ambler is a long-time contributor to* Dr. Dobb's *and is the co-author of* Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise (http://is.gd/fc8owe). *You can follow him on Twitter @scottwambler.*

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

GETTING STARTED WITH GIT: THE FUNDAMENTALS

The distributed SCM system that's taking the world by storm has its own unique way of doing things. This tutorial explains how things work and the basic commands for getting started and checking-in changes.

http://www.drdobbs.com/240160261

GIT TUTORIAL: BRANCHES AND WORKFLOW

Pulling changes, handling merges and conflicts, and building a productive workflow are activities that Git handles in its own productive but unique way.

http://www.drdobbs.com/240160315

ADDRESSES AND NODES: TWO WAYS TO GET AROUND

Programs that avoid address arithmetic can also avoid moving data. The reason is that it is only arithmetic that really cares about where in memory an object is; in other circumstances, having a pointer to that memory is enough.

http://www.drdobbs.com/240160602

DISTRIBUTED VCS: UNDERSTANDING THE PARADIGM SHIFT

Distributed Version Control Systems use a different model of check-out, check-in, branching, and merging than traditional, centralized, repository-based solutions. Understanding these differences is the only way to enjoy the benefits of distributed version control.

http://www.drdobbs.com/240159530

ATOMIC OPERATIONS AND LOW-WAIT ALGORITHMS IN CUDA

Used correctly, atomic operations can help implement a wide range of generic data structures and algorithms in the massively threaded GPU programming environment. However, incorrect usage can turn massively parallel GPUs into poorly performing sequential processors.

http://www.drdobbs.com/240160177

DEFINING AN MPI DERIVED TYPE FOR GAME OF LIFE

What if you need to send data that is composed of two or more primitive types? Clay Breshears shows you what to do when using MPI to send data between distributed processes.

http://www.drdobbs.com/240160694

Dr.Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

CONTRIBUTING EDITORS
Scott Ambler
Mike Riley
Herb Sutter

DR. DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

INFORMATIONWEEK

Rob Preston VP and Editor In Chief, InformationWeek
rob.preston@ubm.com 516-562-5692

Chris Murphy Editor, InformationWeek
chris.murphy@ubm.com 414-906-5331

Lorna Garey Content Director, Reports, InformationWeek
lorna.garey@ubm.com 978-694-1681

Brian Gillooly, VP and Editor In Chief, Events
brian.gillooly@ubm.com

INFORMATIONWEEK.COM

Laurianne McLaughlin Editor
laurianne.mclaughlin@ubm.com 516-562-5336

Roma Nowak Senior Director, Online Operations and Production
roma.nowak@ubm.com 516-562-5274

Joy Culbertson Web Producer
joy.culbertson@ubm.com

Atif Malik Director, Web Development
atif.malik@ubm.com

MEDIA KITS

http://createyournextcustomer.techweb.com/media-kit/business-technology-audience-media-kit/

UBM TECH

AUDIENCE DEVELOPMENT
Director, Karen McAleer
(516) 562-7833, karen.mcaleer@ubm.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Sales Director, Michele Hurabiell
(415) 378-3540, michele.hurabiell@ubm.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, sandra.kupiec@ubm.com

Account Manager, Vesna Beso
(415) 947-6104, vesna.beso@ubm.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, matthew.meyer@ubm.com

MARKETING
VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, winnie.ng@ubm.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, angele.leemoll@ubm.com

Marketing Manager, Monique Luttrell
(949) 223-3609, monique.luttrell@ubm.com

Program Manager, Nicole Schwartz
516-562-7684, nicole.schwartz@ubm.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
(212) 600-3092, steven.sorhaindo@ubm.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mary.hyland@ubm.com

Account Manager, Tara Bradeen
(212) 600-3387, tara.bradeen@ubm.com

Account Manager, Jennifer Gambino
(516) 562-5651, jennifer.gambino@ubm.com

Account Manager, Elyse Cowen
(212) 600-3051, elyse.cowen@ubm.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kathleen.jurina@ubm.com

BUSINESS OFFICE
General Manager,
Marian Dujmovits
United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030

(516) 562-5000

UBM Tech

UBM TECH
Paul Miller, CEO
Robert Faletra, CEO, Channel
Kelley Damore, Chief Community Officer
Marco Pardi, President, Business Technology Events
Adrian Barrick, Chief Content Officer
David Michael, Chief Information Officer
Sandra Wallach CFO
Simon Carless, EVP, Game & App Development and Black Hat
Lenny Heymann, EVP, New Markets
Angela Scalpello, SVP, People & Culture
Andy Crow, Interim Chief of Staff

UNITED BUSINESS MEDIA LLC
Pat Nohilly Sr. VP, Strategic Development and Business Administration
Marie Myers Sr. VP, Manufacturing

UBM TECH ONLINE COMMUNITIES
Bank Systems & Tech
Dark Reading
DataSheets.com
Designlines
Dr. Dobb's
EBN
EDN
EE Times
EE Times University
Embedded
Gamasutra
GAO
Heavy Reading
InformationWeek
IW Education
IW Government
IW Healthcare
Insurance & Technology
Light Reading
Network Computing
Planet Analog
Pyramid Research
TechOnline
Wall Street & Tech

UBM TECH EVENT COMMUNITIES
4G World
App Developers Conference
ARM TechCon
Big Data Conference
Black Hat
Cloud Connect
DESIGN
DesignCon
E2
Enterprise Connect
ESC
Ethernet Expo
GDC
GDC China
GDC Europe
GDC Next
GTEC
HDI Conference
Independent Games Festival
Interop
Mobile Commerce World
Online Marketing Summit
Telco Vision
Tower & Cell Summit

http://createyournextcustomer.techweb.com