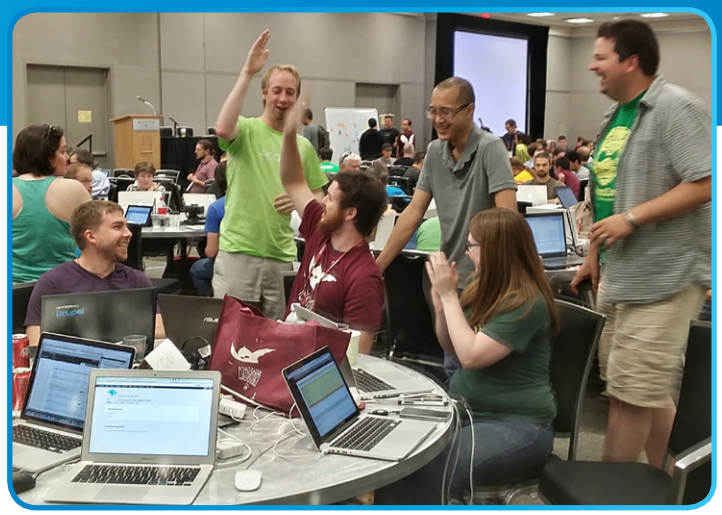
BEHAVIOR-DRIVEN DEVELOPMENT

A Large Scale Drupal Guide

LARGE SCALE **DRUPAL**

Acknowledgements



This guide represents the collaborative efforts of many individuals and organizations participating in Acquia's Large Scale Drupal (LSD) Program.[a] It is based on interviews with and input from over half a dozen LSD Members, two of our LSD Partners, and several individuals at Acquia.

Thank you all for your participation, funding, and support!

Special thanks to: Melissa Anderson at Tag1 Consulting,[b] and the lead author of this document; Steve Graham and Nate Swart at Acquia,[c] for their leadership and dedication to completing this guide and related software projects; the many LSD Members, including McKesson Specialty Health[d] and Dennis Publishing[e] for allowing us access to their teams and providing insight and feedback on this document, for their work with BDD, and for their funding and support of the LSD Program; and to our two founding LSD Partners, Appnovation[f] and Phase2,[g] for contributing to this document, and for dedicating resources and funding to build out and work on BDD software projects and tooling over the last year, in conjunction with LSD Members. (Please see the inner back cover for more information about these projects, including video demos and links to source code.)

We'd also like to thank *Drupal Watchdog*[h] for its help and support editing this document, and for printing, publishing, and distributing this guide (also available online at http://wdog.it/lsd/bdd) on behalf of Acquia's Large Scale Drupal Program.

LSD Contact Information

For more information about how your organization can benefit from and join the Large Scale Drupal Program, please visit our website LargeScaleDrupal.com or contact:

- **Michael Meyers** - VP, Large Scale Drupal
  michael.meyers@acquia.com

BDD Contact Information

For more information about, or help with adopting and leveraging Behavior Driven Development at your organization, please contact our contributing LSD Partners:

- **Adriana Zeman** - VP, Acquia, Consulting Services
  adriana.zeman@acquia.com
- **James Heise** - Sr. Director, Global BD, Appnovation
  james@appnovation.com
- **Frank Febbraro** - CTO, Phase2
  frank@phase2technology.com
- **Peta Hoyes** - COO, Tag1 Consulting / *Drupal Watchdog*
  peta@tag1consulting.com

a Acquia Large Scale Drupal: http://www.largescaledrupal.com/

b Tag1 Consulting: http://tag1consulting.com/

c Acquia: https: //www.acquia.com/

d McKesson Specialty Health: https://www.mckessonspecialtyhealth.com/

e Dennis Publishing: http://www.dennis.co.uk/

f Appnovation: http://www.appnovation.com/

g Phase2: http://www.phase2technology.com/

h *Drupal Watchdog:* http://drupalwatchdog.com/

© 2014 Michael Meyers

Table of Contents

Introduction

Acquia's Large Scale Drupal Program (LSD) is a strategic alliance that enables organizations using Drupal to collaborate on significant enhancements to Drupal – through networking, knowledge sharing, funding, development, and engagement with the Drupal community. By working together, our Members and Partners create an economy of scale, driving down the cost of building and maintaining software; through the input of many minds and perspectives, we help each other create more robust software systems and run more effective organizations.

Our LSD Members and Partners meet regularly to discuss and identify common problems and shared needs, and work together on open source solutions and resolutions.

At the end of 2012, during our Q4 LSD Technology Leadership Conference, we introduced the concepts behind Behavior Driven Development (BDD) to our Members and the Drupal Community. *BDD is a next-generation development methodology that complements and extends agile and lean systems, with a strong automated testing component.* BDD enables organizations to iterate quickly, release software faster, and help ensure that the software we create meets the needs of our stakeholders.

In 2013, through funding and resources provided by our Members and Partners, LSD helped build out and integrate the underlying BDD tooling for Drupal. We also built a comprehensive test suite for Drupal.org, both to facilitate upgrading our home on the web to Drupal 7 and to provide an example of BDD implementation that others could leverage and build on.

We've hosted several BDD webinars and presentations to help Members understand its methods and benefits, and enable them to adopt and leverage BDD concepts and tooling.

In 2014, we continue to fund and support several software projects. (Please see the "LSD BDD Software Projects and Tooling" section on the inner back cover and the Additional Resources section of this document for more information on these and other tools.)

Many of our LSD Members and Partners have adopted aspects of BDD with great success and we've highlighted some of their stories and lessons learned in this document. The goal of this guide is to introduce BDD to a broader audience, to help you better understand the methodologies, tooling, and benefits to your organization, and to enable your organization to quickly get up and running with BDD. The Drupal Community is in the process of adopting BDD as a new standard for Drupal 8 Core and contributed module development; moving forward, it will become even more important to the platform (both Drupal 7 and Drupal 8) and to the entire Drupal Community.

LSD is not a profit center for Acquia. It is a Member- and Partner-funded and driven program, managed by Acquia's Office of the CTO (OCTO). There are currently over 50 organizations around the globe that are participating in the LSD Program, representing some of the world's largest websites and most well-known brands running Drupal. This overview guide, and the related open source software projects, are funded and created by our Members and Partners, with additional support and funding provided by Acquia. To learn more about joining and supporting the LSD Program, and our projects and initiatives, please visit our website at LargeScaleDrupal.com or contact me by email. We hope you will join us in our collaborative efforts.

Michael Meyers
VP, Large Scale Drupal
michael.meyers@acquia.com

CC IMAGE COURTESY OF GÁBOR HOJTSY ON FLICKR

Background

Imagine the perfect project: the organization knows what it needs, what it wants, and the difference between the two; the milestones are reasonable; the work is interesting, and communication is prompt, clear, and effective; attention is given to design and user experience, with enough time and budget for adequate discovery, iteration, testing, and training; and last, but not least, technical resources and personnel are available and reliably allocated.

As we know, rarely do all these elements converge in the same project. Usually, limitations in one or more area nudge a project off-track, sometimes disastrously. Even under the best of circumstances, projects grow and change during their lifespan in a variety ways – some expected, some that really should have been expected, and others that no one could have foreseen.

Finding a way to mitigate the risks of software development and ensure that the final product or service delivers needed business value is at the heart of Behavior-Driven Development (BDD).

BDD has a rich and specific history, but it comes down to three core principles:

- Teams need **conversations** throughout the software development process to focus everyone on discovering and delivering value.
- Everyone on the team should describe application features using a **common language** to reduce ambiguity and facilitate understanding.
- Every feature should have an **identified, verifiable value** to the business or organization.

CC IMAGE COURTESY OF GÁBOR HOJTSY ON FLICKR • CC IMAGE COURTESY OF GÁBOR HOJTSY ON FLICKR

Discovery and Development

Conversations

Initial conversations in software development are typically rife with the unknown. While the business analysts and marketers are generally (but not universally) able to articulate what they want and what business problems they're trying to solve, they may know little about what's possible either with software in general, or with the particular technology in question. They may not know what questions to ask, and they probably have little feel for the comparative effort involved to implement one feature over another. They may present problems that developers haven't solved before, and they sometimes assume that developers understand the business needs.

Meanwhile, the technical team may assume that the business side has a fundamental understanding of web sites or the application they're trying to build. As stakeholders learn what's possible, they can make better connections and decisions. As everyone's understanding grows, the initial requests change.

BDD encourages the conversations required to navigate complexity, uncertainty, and growth throughout the development cycle. It specifically asks everyone involved to focus attention on the behavior of the application as it will be experienced by the user, and it provides a formalized, accessible language that everyone – business analysts, stakeholders, project managers, developers, trainers, technical writers, quality control engineers – can understand and use in order to have those conversations.

"The Three Amigos" is a great way to ensure that conversations happen. The Three Amigos approach can be more[1] or less[2] formalized, but essentially, it means having three key stakeholders collaborate on features before moving into development:

- a business analyst or product owner
- a developer
- a tester

Common Language

Common language helps the whole team understand what's expected from the application. While a stakeholder may describe in elaborate detail the pains they've experienced meeting HIPPA and FERPA requirements, and the developer grapples with the security implications of storing confidential data (in terms of encryption, password strength, and access control), the actual users, who may not be represented early or at all in the process, get left in the cold. So whose language is right? Although different team members need to work with the nomenclature of their domain, the key is to always keep in mind what's important: What the user sees and does.

PUBLIC DOMAIN IMAGE SPACEFLIGHT NASA.GOV: STS-124_LAUNCH_FROM_A_DISTANCE

To do this, BDD uses Gherkin – a plain language template, not a pickle – to express how the feature will behave under various circumstances.

Features

A scenario takes the form of a user story, something many software teams already produce as part of their requirements-gathering. It looks like this:

```
Feature: Title
  As someone who will use the software
  I need or want some functionality
  So I can achieve something of value
```

Sometimes, you'll see documentation of a slightly different structure, which puts the emphasis on the value rather than the user:

```
Feature: Title
  In order to achieve something of value
  As someone who will use the software
  I need or want some functionality
```

Either syntax is fine. The structure is intended to ensure you've:

- provided a quick name for the user activity;
- articulated the business value;
- identified the primary target user;
- described what that user needs or wants to do.

To illustrate with a concrete example, let's look at a relatively straightforward request: staff blogs.

Example:

```
Feature: Post blog entry
  In order to increase the visibility of
    my department's work
  As a staff member
  I need to be able to post blog entries
```

Once the feature is written, the Three Amigos expand on the story using clear relevant scenarios to illustrate the expectations.

Probing for Different Conditions: Scenarios

Scenarios are BDD's powerful way of reducing the ambiguity that comes when the business or marketing stakeholders describe one thing and project managers or developers envision something else. Scenarios line up the entire team behind the application user's electronic device.

The first scenario typically begins by describing what the user will experience if everything is done as expected. For the main scenario – for the staff blog feature – the user will log in, obtain the right access, be in the right place to see the "Create" link, fill out all the required fields appropriately, and publish. *Voilà!*

Example:

```
Feature: Post blog entry
  In order to increase the visibility of
    my department's work
  As a staff member
  I need to be able to post blog entries
Scenario: Successfully publish blog entry
  Given I am logged in as a staff member
  And I am on my dashboard
  When I create a blog
  And I publish it
  Then the teaser should appear on the
    homepage
  And teaser should link to the full post
```

This first scenario, like all other scenarios, consists of three parts:

Set the scene: the Givens

Givens set the scene and establish where to begin. The "Given" part of the staff blog describes the right user with the right role in the right place:

```
  Given I am logged in as a staff member
  And I am on my dashboard
```

Change the scene: the When

With the context established, the "When" part of a scenario describes what the user might do:

```
When I create a blog
And I publish it
```

Describe the response: the Then

The "Then" section describes one or more responses the application should perform in reaction to the user's "When" entries.

```
Then the teaser should appear on
  the homepage
And the teaser should link to the
  full post
```

The goal of the feature is to "increase the visibility of my department's work." There's nothing inherent in that goal that prescribes exactly where or how that should happen, so it's good to prod a little at this point in the conversation: Is there anything else that the user should observe? Does information about the blog entry appear on other pages? Should it be highlighted on a list of recent entries? Included or excluded from site search? Archived at some point? Available via RSS? Should it go through a company approval process?

By asking these kinds of questions, we can draw out various assumptions: Don't all blogs have a Recent posts lists on every page? Don't they automatically let you tag an entry with a category and then find related entries? Don't they have a tag cloud – those things where the words get bigger when there are more entries in a category? Don't users need approval before they can publish? (Maybe, maybe not.)

Even something as common and seemingly clear-cut as a blog often contains a surprising amount of ambiguity. Imagine more complex features, like departmental publishing workflows, medical records access, or online commerce. It's the conversations along the way that clarify unspoken assumptions, acknowledge that new requests will happen as the project develops and understanding grows, and allow further requests based on their value to the business.

Useful questions for imagining scenarios are:

What about...?

What if... ?

Focus on the User

Feature files maintain a focus on what the user does and how the application responds. A critical component of Gherkin style dictates that you speak with language that the target user can reasonably be expected to understand. This user-centric focus means that everyone will be drawn back from their domains of expertise to think about what the target user of the feature needs. There's a wide variety of users, too. Features intended for a site visitor may differ from those intended for site administrators, editors, or moderators. As the user varies, it's possible the language used in a scenario will vary as well.

Additional Scenarios

What if the blog author doesn't fill out a required field? What if the author is logged in but doesn't have permission to create a blog? What will that experience be like? Although these paths are less desirable than the successful outcome, the application's behavior should help the user along the correct path, and BDD is the first and primary place you define what that 'happy path' looks like. The idea behind scenarios is NOT to hunt exhaustively for every edge case, but rather to support users on their way to the desired outcome.

Keep in mind that assumptions about the application behavior can vary a lot. For example, a blogger may be technically competent, but have little time or patience to get bogged down with too many required fields. Also, it's critical that authors can easily save and find drafts, because they're not likely to finish them in one sitting. Developers, however, may assume that the title is required because it's what lets the user locate and navigate back to their work. Scenarios help specify what should happen in those cases. Here, the business value would suggest that an empty title field should save the work with a placeholder, perhaps something like "Untitled" with a link to the blog.

How Big is a Feature?

Features should be fairly granular. Each feature is elaborated on in usually no more than five to seven scenarios; more than that, and it's likely the feature needs to be broken down into smaller components. If you're used to thinking about features on a larger scale, this can take some practice. You may find that what you thought of as a feature is more like a feature *set*: several smaller features that work together to deliver complex functionality.

The discipline of creating smaller features and describing the scenarios with formalized language helps untangle complex interwoven descriptions, or vague and general requests, turning them into something that a developer can effectively focus on implementing.

Identifiable Value to the Business

Sometimes, it can be quite a challenge to focus on the behavior of the user and, at the same time, state that behavior's value to the business. Sure, it's easy enough when what the user wants coincides with what the business wants them to do, but quite often that's not the case. In a blog post[3] from 2008, Liz Keogh describes just such an at-odds situation.

Her example:

```
As a user
I want to fill in a captcha box
So that... what? No! What a waste of
   my time!
```

She suggests this is better written as:

```
In order to stop bots from spamming
  the site
As a member of the commercial team,
  I want users
  to fill in a captcha box
```

The formula in these cases look more like:

```
In order to <deliver some business
  benefit>
As a <role> I want <some other role>
To <do something, or use or be
  restricted by some feature>
```

Clearly identifying the people who are concerned about the value of a feature has a long-term benefit. When something affects how that feature works, it's explicit who needs to be involved in the conversation.

In pursuit of delivering value, if there's a consistent tension between what the user wants and what the organization wants from the user, it's worth taking a look at how to meld the two.

> *"The magic of the Internet, the work of the user experience engineer, is to find the intersection of the desire path – the motivation driving someone to visit your site – with the desired path, or any of the potential actions you hope a visitor will take to support your organizational goals."*
> http://wdog.it/lsd/1/thinkshout

When Does All This Happen?

You can begin to capture features in the very early stages of a project – long before you decide to move forward with software development – even though the language is not yet formalized; it may just be a laundry list of big-ticket items or detailed steps to support organizational processes. You may even have a variety of scenarios.

Keep in mind that the quite detailed development of scenarios is usually saved until the time when the feature has actually been prioritized for development. If you begin too early, the detail can impede learning and listening, which increases the chance of spending time developing the wrong functionality or detailing something that never gets implemented at all. At the same time, if you hear about key scenarios early on and you can capture important ones, they can help you understand the project's complexity.

Who Does What?

It's a completely reasonable BDD workflow to have something like the following.

- Business, Developer, and Tester work on features, feature prioritization, and scenarios.
- Developers implement features and may come back with new scenarios for clarification.
- Testers use the scenarios as guides to ensure that the actual implementation meets the acceptance criteria.

All of this can be done without any special tools or automation at all. An organization, by committing to the collaborative process, can save money and reduce bugs.

Am I Doing This Right?

In the early stages, the structure of the language, who exactly records it, and other minutia, take a back seat to the primary purpose: discovering what you don't know.

If your practices hinder having the conversations and learning what you need to know to deliver value, then it's time to re-evaluate. The author of Behat, Konstantin Kudryashov, has a great slide deck that helps focus on using the tools well: http://wdog.it/lsd/1/example

Summary

Driving development of your technology with BDD helps make implicit assumptions explicit. The following points help structure conversations to reduce the unknown:

- Name the business value and the features user.
- Discuss with the key stakeholders how those features behave under different circumstances.
- Consider features in a fine-grained manner at the time they're headed toward implementation.
- Use language that everyone on the team can understand.
- Focus on the value of the feature to the business.

The conversations increase the chance that valuable features are being implemented and will require less rework.

CC IMAGE COURTESY OF STEVE JURVETSON (JURVETSON) ON FLICKR

Automation

Despite BDD's emphasis on discovery and communication, and the tremendous value these provide to organizations, it's often the automation tools that get people excited about BDD.

Manually clicking through applications is tedious and error-prone; a surprising number of obvious bugs make it all the way through to production, there to be reported by end users. The promise of automation to simulate user interactions and allow routine verification that things are working as expected is a powerful lure.

Most organizations need (or want) to iterate quickly and be able to roll out frequent changes to their applications and web sites in order to meet customer demands and be competitive. The automation of tests, especially for the features which supply major business value, provides professionalism and confidence with each deployment; well-done automation means the team can focus more effectively on the new features.

CC IMAGE COURTESY OF ROBERT S. DONOVAN (BOOLEANSPLIT) ON FLICKR • CC IMAGE COURTESY OF WINDELL OSKAY (OKSAY) ON FLICKR

Tools

There are numerous software tools out there which can map Gherkin steps to code that will open a web browser and actually click the links and fill out the forms described in the scenarios, but notable practitioners warn strongly against using the structured language and automation tools before making the necessary philosophical and process adjustments to implement BDD practices at your organization.

Once the commitments to new ways are made and adjustments are in place, there's an incentive to consider automation. Within the Drupal community, Behat[4] and Mink[5] are obvious tool choices. Because they're written in PHP, Drupal developers don't need to context-switch to a different programming language in order to automate steps. The Drupal community has embraced them with an extension to Behat itself[6] (funded and supported in part by LSD Members) that allows the community to share code supporting common Drupal-specific behaviors, as well as other modules to assist in using Behat with Drupal.[7] As of Drupal 8, BDD tools will be incorporated into Drupal core,[8] distributed with the platform, and BDD practices will become a standardized aspect of the Drupal community development process. There's a lot of value in collaborating on the underlying toolsets, and LSD Members have funded multiple projects to share upfront R&D costs as well as long term support. So many elements of automation are reusable. By sharing work, we can create resources that solve our common needs and focus independent efforts on what is unique to an application.

Kinds of Testing

Because there is a lack of consistency in the way different types of testing are defined, let's place these on a continuum:

- *Unit tests* are written by programmers for programmers. They test at the function or method level, and are intended to encourage loosely-coupled, modular code. They ensure the independence of code segments in a way that integration tests do not.
- *Integration tests* are also written by programmers for programmers. They check the interaction of functions or methods working together for the same reason as unit tests: to verify that code works as expected, especially when changes are made elsewhere.
- *Acceptance tests* are written as a collaborative effort by the team to ensure that the application meets the business objectives and continues to deliver the business value when changes are made elsewhere in the system (such as after a security update or the introduction of a new feature). Because they describe user behavior, they can either be performed manually or automated.
- *Functional tests* check the application's functionality with interface-driven tests that are independent of the underlying code. This includes the automation of acceptance tests.
- *Regression tests* include unit, integration, and functional tests. Automated or manual, they are intended to identify when things stop working the way they used to.

When to Automate

Automation increases the number of times that code can be tested, which allows bugs to be detected and resolved earlier in the cycle. The more valuable or stable a feature is, or the more often it is used, the better a candidate for automation it becomes.

Automation is equally important to reduce cognitive load on the team by minimizing the amount of information, interaction, and context switching required when members are involved in complex learning and creation.

Automation should:

- **Help developers focus as they develop.**

Front-end developers work on the theming for a page that depends on complex user actions. They need to click through the application to see that the combination of javascript, ajax, and CSS is coming together as expected. Describing the steps once in Gherkin – and executing them automatically – allows these developers to keep their focus on the code they're writing; it doesn't matter what programming languages they know.

A back-end developer may have the same need. More than that: with complex tasks like E-commerce or editorial workflow, it can be tempting to make a one-line required code change and then skip manual verification that the complete process is working. With a quality automated test in place, we've found developers are more likely to check their work and pass along higher-quality code.

- **Provide living documentation and organizational resilience.**

When automated functional tests are run at every deployment, a change in requirements that causes a test to break will be identified; when there's an organizational commitment to automation, the test will be updated to reflect the change, so that a new arrival can discover the latest intended functionality. Automation provides incentive to keep features and scenarios up-to-date, increasing organizational resiliency. A new team member has access to current information about what a feature is expected to do.

- **Increase the likelihood that developers will pass along better tested code.**

The context switch between writing code and testing the user interface is significant. It's much easier for a developer to test only a small part of what might be affected by his or her changes or to verify behavior as a user with elevated permissions. Running the automated tests before passing along code can expose problems and allow them to be addressed long before QA looks at the new feature, eliminating delay.

- **Produce better-constructed sites.**

Code that is hard to test is often difficult to maintain. Just as unit testing encourages good design at the code level, functional tests can promote more consistent and more coherent HTML output. The same practices that facilitate easy CSS markup and the use of automated screen readers also support functional tests. The performance issues that become apparent when running functional tests affect users too, so focusing on clean, fast-loading code in pursuit of business value helps the overall app.

- **Increase frequency and quality of deployments.**

The point has been made, but it's worth repeating: A well-automated test suite ensures that areas of high business value are tested with every deployment. This allows the human testers to focus on new areas, as well as maintain, refactor, and expand automated testing.

Functional Tests and Continuous Integration

There's value to having developers and testers run tests locally, but most often, organizations will incorporate tests into a Continuous Integration (CI) system. Many variations on CI exist, but a professional development workflow looks something like this:

- Developers work in their local environment, using a version control system.
- When they're ready, they push their code to a shared server (often called something like "development").
- Before that code is accepted on the development server, unit and integration tests are automatically run to ensure the code is functional, preventing a developer from pushing a regression that will be picked up by the rest of the team. Carefully selected functional tests can be added as well.
- Code is pushed to a quality assurance server, where manual verification is done. Sometimes acceptance demos are performed there.
- Prior to a release, code is staged and the full suite of tests – unit, integration, functional, and manual – is run.
- Code is deployed to one or more production servers.
- Post-production verification is done, including manual and less-intrusive automated functional tests.

Functional tests can also be integrated into the deployment process, but this should be carefully done, with special attention to:

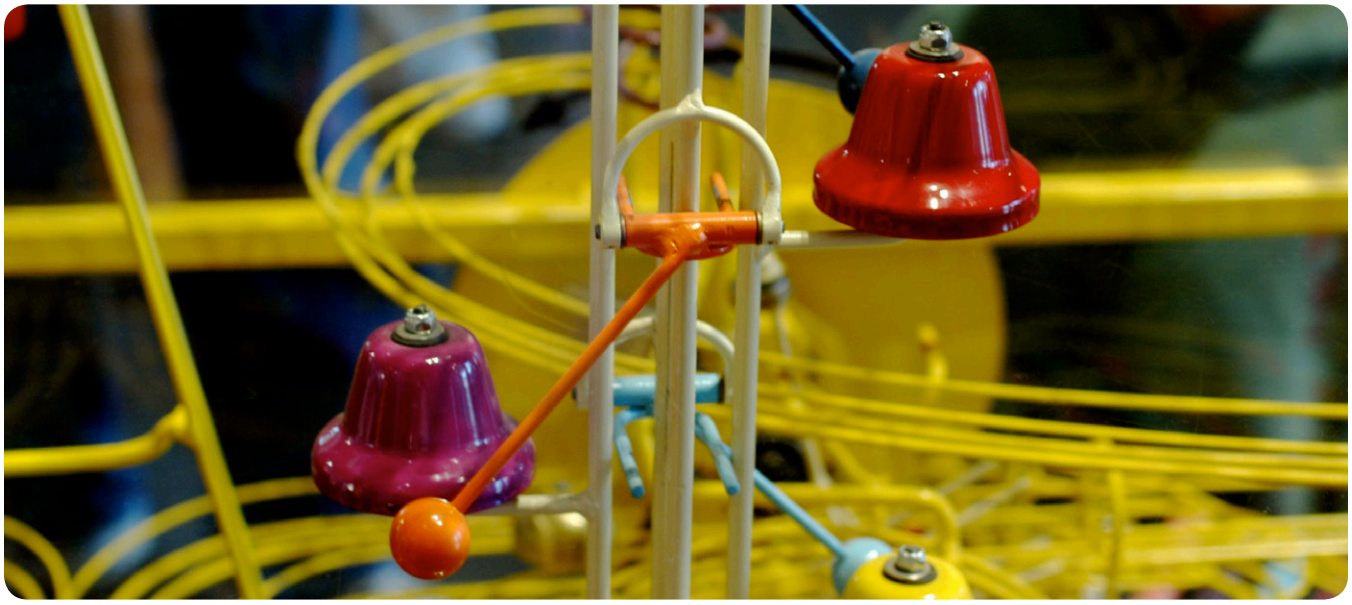
- **Execution time** Functional tests are much slower than unit tests because they involve actual simulation of user actions. It's unwise to run every functional test each time a developer pushes code. Full test runs are more appropriate for pushing to a staging environment prior to release. Wise tagging can either include tests that should be run or exclude tests that need not run at this time.
- **Robustness** Nothing will discourage the use of functional tests like writing brittle automation that causes failure when everything is actually just fine. A common example of this is writing a test that relies on seeing text on a page that is highly subject to change. Using the blog example, if someone writes a test that checks for a blog title on a "Recent Posts" list, that title is definitely going to change in the next week or two, causing a false failure. Unreliable services can cause this, too. If there are known performance issues with other services that cause excessively slow-loading pages, tests will fail. Don't put anything but the most valuable and robust tests between a developer and pushing code. Before a deployment, maybe, but not when change and collaboration is the most critical activity.

Workflow

Organizations looking into Behavior-Driven Development are often unclear about who should do what and when, which makes sense. There is no single recipe. It is a highly contextual question and one that can be really contentious in environments where everyone is already running at full capacity. It's hard to learn new things without adequate time, and the development of the tests can be just as difficult as the development of the product, especially if you bypass the communication.

That said, here's one way it might proceed:

- Business, marketing, and/or sales will be discovering customer needs. They'll be looking for areas of the unknown (as well as the known) and striving to find out what they need to initiate the project. They may begin user-stories themselves or that may happen later, but they're beginning to discover requirements and can benefit from understanding the Gherkin format, even if they're not writing them.
- Project managers and business analysts will review features and begin working at the scenario level. They are likely NOT to have the click-by-click level of detail needed to use available pre-built steps at this stage.
- Developers will use the features and scenarios to implement the requirements. Where useful to that implementation, they may automate some scenarios.
- QA will collaborate on maintenance and make sure that the tests are appropriate, robust, and maintainable, implementing additional automation.
- Automated test failures can then be verified by anyone. A likely path is to have the initial failure interpreted by less-technical staff: someone who can manually see what happened and know if it's either a desired change (and the test needs updating), a poor test that needs improved quality, or a problem with the application that requires developer intervention.



CC IMAGE COURTESY OF PHIL ROEDER (TABOR-ROEDER) ON FLICKR • CC IMAGE COURTESY OF JEFF KUBINA (KUBINA) ON FLICKR

Another common workflow emerges when you have an existing site and you're looking to introduce automated regression tests. BDD doesn't guide the development of new features yet. Instead, BDD tools are being introduced to make sure the existing features are still working. Business value, discovery, and ensuring the team is delivering the *right* features, all take a backseat to automation.

The work begins at a completely different phase. It's normally undertaken because bugs are reaching the production web site with unacceptable frequency. The team is under pressure to reduce regressions, often at the same time they're under pressure to deliver features faster.

Developers or testers will then dig in to write some tests. They're following an existing interface, so they tend to use prebuilt steps that are click-by-click and implementation-specific. The actual feature descriptions become hard to write without engaged participation from the business team, and they become correspondingly difficult to read. There may also be an impulse to crank out as much automated testing as possible, as quickly as possible, in an effort to restore a sense of control. Unfortunately, there isn't always an effort to get the business team back into the loop and the major value of BDD isn't realized.

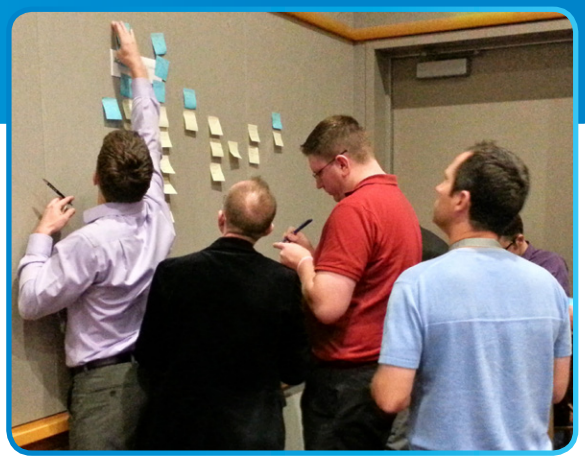
A reasonable workflow under these circumstances is still possible. It might look like this:

- Less-technical staff write the features and scenarios, doing their best to fit it into pre-existing steps. Projects like the BDD Use Case Editor are a huge boon, lowering the barrier to entry for writing and running the tests.
- Developers use the tests to check their own work, improving the quality both of code and of tests.
- QA uses the tests with an eye on their long-term value.
- The project manager, scrum master, or QA person works with the business people to better understand the value of the feature, refactoring what was written.
- Each deployment, more coverage is added, based on known vulnerabilities and the value of the feature to the business.
- As the business people become familiar with the common language, new work begins to use that language and the full benefit of BDD can be realized.

In every workflow, regardless of when Behavior-Driven Development is introduced, support for the whole development process is necessary. It needs to be an entire team effort, regardless of where you begin.

CC IMAGE COURTSEY OF ANURAG AGNIHOTRI (AGNIHOT) ON FLICKR

Advice from LSD Members



To provide perspective on implementing Behavior-Driven Development, or using the PHP tools Behat and Mink to facilitate automated functional tests, we interviewed several LSD members about their experience.

We found that organizations whose primary goal was to refine their business requirements were more satisfied with the outcome of both requirements-gathering AND automation than those who did not emphasize improved discovery and stakeholder involvement.

We also found that organizations with multiple sites based on an internal Drupal distribution were generally more satisfied with their automation efforts than organizations with a single complex site. This is partly because they were able to reuse more of their scenarios, but also because the focus on reuse forced them to write more robust tests.

What Not to Do

Don't implement with just part of the team.

For everyone on the team to benefit from BDD, everyone needs to participate. Making the process work for developers is especially key: they sit at the crucial point where they need to understand what's been asked and can make the application more testable. They know what automation helps them, and have a stake in reliable feedback.

Don't rely on pre-built steps.

Pre-built steps become hard to read, incorporate too much implementation detail, and too easily rely on data that's expected to change.

Don't strive for coverage.

When it comes to unit-testing, it may be feasible to think in terms of code coverage. When it comes to automating functional tests, they are slower to perform and there is a seemingly infinite variety of things to test. Focus on value, not coverage, and tag your tests so that you can selectively choose which groups to run in a given context.

Never rely on data from other scenarios.

Avoid having one scenario create content that subsequent scenarios depend upon, where failure will cascade when there may be no problem with the subsequent scenarios.

Don't make features too specific.

Features should be able to contain additional related scenarios as they come up and still make sense. For example, what is the value of meta tags, not just in one context, but across the site?

Don't expect BDD tasks to fit into the existing schedule.

Teams need time to learn the new tools, automate scenarios, respond to failures, and refactor for greater reuse and reliability. This needs to be accounted for in sprint schedules. The overall improvements to your development process and ability to execute faster are worth the small upfront investment!

CC IMAGE COURTESY OF KATHLEEN MURTAGH (CEARDACH) ON FLICKR • ©2014 Michael Meyers

Emerging Best Practices

The complexity of software development itself – combined with differences in business goals, team skills, resources, and more – make it difficult to identify a single best practice. Even those we list below will have exceptions, but they reflect the aggregate experience of individuals at six Large Scale Drupal Members, two LSD Partners, and Acquia.

Write the correct tests.

With appropriate unit tests and integration tests, which run much more quickly, the code base itself becomes more stable. Choose the scenarios you automate to run through the interface based on the business value of the feature. In addition, only run tests through a full browser where it's necessary

Use fixtures to set up the Given conditions for scenarios to succeed.

Many folks start out using pre-built step definitions and then either:

1. Use existing data in tests. Often such data is subject to natural and appropriate change over time. When tests fail because of expected changes in content, it doesn't reflect a problem in the code and may not reflect a problem at all. Such failures reduce overall confidence in the value of the automated tests and, depending on organizational processes, distract developers from their tasks at hand.

2. Supply needed data by clicking through the interface to create it. There are situations where there's no way around doing this, such was when you have no privileged access to the database; however, creating data this way is slow, error-prone, and quickly becomes redundant.

> *"The complexity of software development itself – combined with differences in business goals, team skills, resources, and more – make it difficult to identify a single best practice."*

The DoctineDataFixtures[9] extension to Behat allows you to both increase the isolation of tests and more efficiently set up the appropriate conditions for success. The Behat Drupal Extension also provides direct data setup via the Drupal API.

It's okay to test important database content.

It's true that not everything that is tested is encapsulated in code, but important content *is* managed in the database. When the content is critical to the organization, it's okay to check for it with automated functional tests.

Create every scenario so it can be run independently.

Never allow one scenario to depend on another. It can be very tempting – especially if you're not using fixtures to set up the Givens – but dependent scenarios prevent you from effectively tagging, as described below.

Dependent scenarios also create cascading failures: when the first scenario fails, every scenario that depends on it will fail, too, so you're not only denied feedback about the functionality the scenarios describe, you also create noise that detracts from confidence in the tests, and you introduce latency, as the tests need to be run and re-run until the whole chain has been tested successfully.

Tag tests strategically.

Tagging tests allows you to run subsets to provide the right information at the right time.

`@content` Sometimes the ability to see specific data supplied by the database has a high business value. Likewise, ensuring that certain data is NOT visible is incredibly important, but this, too might rely on settings that exist in the database and are not encapsulated in code. When the data is important and is not code-related, tagging such scenarios makes it possible to run them alone or to exclude them when there's a need to focus only on the code.

`@smoke` Different organizations use this tag differently, but the basic idea is that features and scenarios tagged `@smoke` are fast, reliable indicators that something very important isn't working.

`@featureset` Sometimes, it's enough to organize feature files in directories, but often that doesn't provide enough flexibility. Linking related features and scenarios with a common tag will allow developers to run automated scenarios locally that focus on the work at hand, reducing their cognitive load as they check their work, increasing the value of the feedback they receive, and increasing the speed with which they receive feedback.

`@partner` For scenarios that involve 3rd party services and systems.

`@wip` The work in progress (wip) tag is meant for scenarios that are not completely automated or where the feature is not completely implemented. These are typically excluded from runs since they're expected to fail. Consider differentiating between the two situations.

Stop code from moving forward until tests pass.

When test failures stop code from moving forward, a lot of collaboration takes place to solve the problem. The code and the tests, and ultimately the organization, benefit. Achieving this collaboration is another reason why buy-in from the entire team, including those who prioritize the tech team's time, is so important.

Emphasize quality and value over coverage.

When it comes to automation, false failures can undermine your entire effort. Only include your most reliable, valuable scenarios in the continuous integration process; rely on manual testing where automated success is unreliable.

Make solving test failures an entire team activity.

When tests are required to pass before code can be integrated, engineers have an incentive to help each other. In some cases, project managers will expedite getting tests for important features running, giving the tech team (developers and QA) time to work on them.

Have your test reports delivered to where your developers already are.

CI systems can be configured to report into chat programs (irc, skype, slack, etc.) and email mail lists. Developers see pass/fails instantly, and such reporting introduces the social aspect of "who broke the build" that leads to fixing bugs closer to the time they're introduced, which can foster more collaborative development.

Implement BDD Agilely.

When you implement Behat, do it in an Agile way; iterate, getting feedback from all the stakeholders throughout the process.

Automating scenarios for third-party services is difficult and valuable.

When you're running functional tests that involve third-party services, you've gone beyond code. You're testing *your infrastructure*, the third-party infrastructure, their code, and the connectivity. But it can also give you information about third-party reliability. Tagging such tests will allow you to isolate them so they can be included and excluded at the appropriate times during your development cycle.

Automation can create trust.

When it once took 50 days to regression test and a failure was reported, developers would invariably ask "What else broke?" It wasn't possible to answer that question without completing the entire 50 days of manual testing. Now that those tests are automated and run in eight hours, we can tell the developers what else broke, and fixing the regression becomes the team's main focus.

CC IMAGE COURTES OF SETH ANDERSON (SWANSKALOT)ON FLICKR

Conclusion

The diversity of team structures and project needs varies massively. Comparing a project that requires 50 days of manual regression testing – and which has the staff to support that intensity of QA – with a team where the developers are responsible for both producing code *and* doing QA, it's clear there is no single formula for BDD adoption.

Likewise, there is no one-size-fits-all solution for automation. Behavior-Driven Development emerged from a recognition that no amount of up-front planning will eliminate a project's unique challenges. BDD does not try to supply an all-purpose formula or rigid approach to software projects. Rather, it provides practices that, when used to support the situation at hand, helps teams better meet those challenges. BDD fosters communication and encourages the willingness to work together through the entire cycle.

By sharing our experiences – the successes and the false starts – as well as the tools we build to help us deliver quality web applications, we create a dynamic for the unique value our work brings to our customers.

About The Lead Author

Melissa Anderson has worked in software development since 1994. In the early days of the Web, she taught high school language arts, provided software support when people still submitted tickets by fax, and was associate publisher of *Windows Tech Journal*, a print magazine for Windows programmers.

After web access became more common, she worked as a systems administrator, information architect, and web developer. She has been using Drupal since 2006 as a project manager, site builder, and trainer with a passion for quality assurance and user experience. She has been an active community member, helping to bring Git to Drupal.org and leading the initiative for automated functional testing for Drupal.org. She co-maintains the Behat Drupal Extension.

Melissa enjoys amplifying the effectiveness of others by coordinating communication and and keeping focus on producing value. She enjoys creating and maintaining systems, both social and technical, that support organizations and the people within them in their pursuit of quality. She is currently a Manager at Tag1 Consulting.

Footnotes

1 Introducing the Three Amigos: http://wdog.it/lsd/1/intro

2 The Three Amigos in Agile Teams: http://wdog.it/lsd/1/agile

3 Liz Keogh blog from 2008: http://wdog.it/lsd/1/liz

4 Behat: http://wdog.it/lsd/1/behat

5 Mink: http://wdog.it/lsd/1/mink

6 Drupal Behat extension: http://wdog.it/lsd/1/drupal

7 Other Drupal modules that assist in using Behat: http://wdog.it/lsd/1/modules

8 BDD in Drupal 8 core: http://wdog.it/lsd/1/core

9 DoctrineDataFixtures Behat extension: http://wdog.it/lsd/1/ddf

Additional Resources

So You Want To Know More...

BDD Frameworks

BDD frameworks typically consist of story or test parsers that parse through user stories and map each line to functions for execution. Upon execution, these frameworks conduct tests in real browsers. Browsers can be headless (no display) or regular. Browsers are remote controlled through APIs, or through controlling servers (drivers) that implement APIs. As the APIs are often not consistent, these tools tend to also incorporate their own abstracted APIs that understand how to perform common functions in a variety of Browsers and Browser drivers.

For teams that need a fairly full suite of browsers to test against, Saas options are available that host all popular desktop browsers, as well as iOS (iPad, iPhone, iPod) and Android webkit.

Each framework comes with a full suite of tests that can be incorporated into user stories. Each also provides accommodation for programmers to create new functions with their own language strings for user stories. Teams who will need to create their own custom test functions will need to ensure that the framework chosen is written in the programming language their team can support. Since Drupal is a PHP framework, and teams programming for Drupal will already be equipped to program in PHP, we will focus primarily on usage of Behat. Others are being included for broader options, for teams that may need or want them.

We will use and extend the Drupal community's extension for Drupal to widen options for our community. Any gains made in our project-bound extension will be made available to the Drupal extension developers for inclusion, if applicable.

Behat

Behat is an open source PHP based tool that extends and uses the Symfony PHP framework. This is the same framework that Drupal 8 is built on and will provide some familiarity for Drupal 8 developers with respect to setup and configuration, but knowledge of either Drupal 8 or Symfony are not needed to use this tool.

Behat was a fork based on Cucumber and ported to PHP. Theirs is a fairly popular Drupal community extension that is well supported, and also an active community constantly discussing it at Drupal.org.

Behat supports a long list of both regular and headless browsers and uses Selenium to drive common desktop/mobile browsers. In addition to requiring Selenium access, users who choose Behat for local testing on their own computers will also need to install special drivers for most common browsers. The only desktop/mobile browser we found that did not require its own driver was Firefox, and Firefox requires fewer tests modifications to support its DOM. Drivers are available for Webkit (Chrome/Safari), all of the supported IE's, as well as iOS, Android, Windows Phone, and Opera.

Website: http://behat.org/

JBehave

JBehave is an open source Java based tool written initially by Dan North to explore the BDD concept in a real toolset. It is actively maintained and implements the BDD concept as envisioned by North.

JBehave is recommended for teams whose testing groups or resources are familiar with Java and not proficient in PHP. The tool is well known and well supported.

Website: http://jbehave.org/

CC IMAGE COURTESY OF SAM DROEGE (USGSBMIL) ON FLICKR

Cucumber

Cucumber is an open source Ruby based tool patterned after the work completed by Dan North in his Java based jBehave tool. It accommodates the same user story patterns dictated by North. It also expands and extends this functionality with the ability to create stored tables of information within a user story. For example, a list of user names and passwords can be stored in the test to recursively test against the user story.

Cucumber is recommended for teams whose testing groups or resources are familiar with Ruby and not proficient in PHP. The tool is well known and well supported.

Website: http://cukes.info/

Behave

Behave is an open source Python based tool patterned after the work North completed for JBehave. It is actively maintained, and implements the same user stories and testing styles supported in JBehave and the other tools listed in this section.

Behave is recommended for teams whose testing groups or resources are familiar with Python and not proficient in PHP. The tool is well known and supported.

Website: http://pythonhosted.org/behave/

Browser Tooling

Selenium

Selenium is an open source Java based application that provides an abstracted API for browser access. It is run on a host server and opens ports on the host for access by other programs to interact with. It is widely used for acceptance testing in QA teams who often write their own scripts that use Selenium's API directly. Within BDD toolsets, programmers typically use the toolsets provided API instead of writing code that interacts with Selenium.

Selenium can access Firefox directly via Firefox's own API. Other browsers require "Web Drivers" which are standalone applications that provide functionality similar to what Firefox provides natively.

Website: http://docs.seleniumhq.org/

Additional Web Drivers for Selenium

- **Chrome:** http://wdog.it/lsd/1/chrome
- **Safari:** http://wdog.it/lsd/1/safari
- **Opera:** http://wdog.it/lsd/1/opera
- **Internet Explorer:** http://wdog.it/lsd/1/ie
- **Android:** http://wdog.it/lsd/1/android
- **IOS (iPhone, iPad, iPod):** http://wdog.it/lsd/1/ios
- **Windows Phone:** http://wdog.it/lsd/1/windows
- **Selenium Webdriver (used for PhantomJS browser):** http://wdog.it/lsd/1/selenium
- **Hosted Web Drivers (Saas): Sauce Labs supports all major browsers for both mobile and desktop:** http://wdog.it/lsd/1/sauce

Headless Browsers

A headless browser is a web browser without the graphical user interface, useful in test automation.

PhantomJS implements the Webkit code (also used originally by Chrome/Safari) and supports JavaScript: http://wdog.it/lsd/1/phantom

Zombie.js is a Node.js project that also requires Python and supports JavaScript: http://wdog.it/lsd/1/zombie

Goutte is a php application that can be driven by Behat and does not support JavaScript: http://wdog.it/lsd/1/goutte

Sahi

Sahi is an open source special case tool that provides a framework for creating browser based testing scripts. Scripts can be created in any browser, and executed in other browsers. It is written to be cross-browser compliant for operation.

Behat can integrate with Sahi to execute scripts.

Website: http://wdog.it/lsd/1/sahi

Task Runners and Build Tools

Task runners and build tools play a supporting role in a continuous integration and testing framework. These tools allow defining jobs or tasks that consist of a sequence of operations to perform. Some tools (like Grunt) use a programming language (JavaScript) for defining jobs, while other tools (like Phing) depend on configuration files (structured with XML). These tools provide ways to integrate with other common components of a build, testing, and deployment system, such as version control systems, code quality checking tools, asset minification, and much more.

Grunt

Grunt is an open source general purpose JavaScript-based task-runner that uses Node.js. It is supported by a large base of plugins, both officially maintained and community contributed, each of which provides support for a specific task (e.g. making a directory, minifying CSS files, or validating JavaScript with JSHint). These generic plugins are then configured with project-specific settings in the Gruntfile.js included with the project.

Website: http://wdog.it/lsd/1/grunt

Phing

Phing is an open source PHP-based build tool based on Apache Ant. Its primary focus is building, testing, and packaging/deploying code, but it is agnostic on the steps involved in these processes for any given project. It includes a core set of tasks that cover most basic build processes and a set of optional tasks that provide integration with common tools. For example, the core tasks include basic if-else conditionals and file management tasks, while the optional tasks provide Git and documentation generation integrations.

Website: http://wdog.it/lsd/1/phing

Related LSD BDD Software Projects and Tooling

Over the last year the LSD Program has worked with our Partners and Members on several exciting open source BDD projects and initiatives. For each of the projects highlighted below, you can access the code and view a quick video demo. Check out our recent BDD Prototypes & Tooling webinar* to see the following projects covered in greater detail:

BDD Use Case Editor

The use case Editor is a robust GUI that enables non-technical stakeholders to engage (write, edit, run, record, and view tests/use cases) in enterprise BDD. The tests run via a 3rd party SaaS integration with Sauce Labs; the project was led by an LSD Member in the Pharma & Life Sciences sector, and Appnovation, with participation and contribution from other Members and Partners.

Project: http://wdog.it/lsd/1/uce
Vagrant image: http://wdog.it/lsd/1/uce2
Video demo: http://wdog.it/lsd/1/uce3

BehatRunner

BehatRunner is a development tool that resolves all prerequisites for running behat, auto-discovers/finds and helps manage and run Behat tests locally on your dev machine (drush & gui), and standardizes how everyone should store Behat tests. The project is led by Acquia and funded in part by an LSDMember in the Media & Entertainment sector.

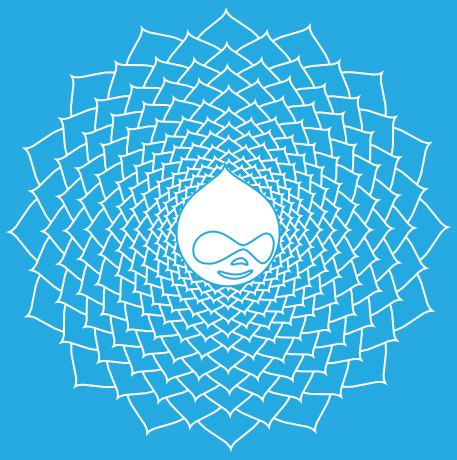
Project: http://wdog.it/lsd/1/br
Video demo: http://wdog.it/lsd/1/br2

BDD for local development environments

This is a lightweight powerful Continuous Integration (CI) toolchain for local dev machines that enables true test-driven development – Behat tests run on file save – all on your local machine. It is led by Phase2 and funded by an LSD Member in the Media & Entertainment sector.

Project: http://wdog.it/lsd/1/ci
Vagrant image: http://wdog.it/lsd/1/ci2
Video demo: http://wdog.it/lsd/1/ci3

* BDD Prototypes & Tooling webinar: http://wdog.it/lsd/1/webinar



Collaborative efforts get big results

LARGE SCALE **DRUPAL**

Benefitting Organizations Through Collaboration

Large Scale Drupal is a strategic alliance that enables organizations using Drupal to collaborate on significant enhancements to the platform through networking, knowledge sharing, funding, development, and engagement with the Drupal community.

Learn more about building success with the leading organizations running Drupal.

LargeScaleDrupal.com

Acquia POWERED,
COMMUNITY DRIVEN