Best Practices of Web Development

Table of Contents

Who Needs This?

The short answer: basically anyone who writes code. Best practices are not just for software developers who work in collaborative team environments. Even those working in their solitary

silos can benefit from establishing consistency in their code and workflow. Being able to showcase a portfolio of consistent coding style and explain with confidence each stage of the software development process to a prospective client or hiring manager will certainly evoke an image of efficiency and professionalism. Not having to "figure things out" again when starting each new project is a huge boost to overall productivity.

Inconsistent coding styles and naming conventions, lack of documentation and established processes are the hallmark of amateurs. Most of the topics in this guide have been debated for decades by software engineers of the highest caliber. If they invested the time to contemplate such matters, then you can rest assured these are not trivial issues. Just ask yourself these questions:

- Have you ever had to go back and modify your code a year later only to find yourself trying to understand what you wrote or why you wrote it?
- Do you ever wonder why your co-developers don't name variables or classes in a way that makes sense to you?
- Did you ever discover after spending hours or days writing some really clever code, that someone else had already written it, and their code was really terse and easier to read?
- When you sit down to code some new feature, do you plan it out in tasks and document the procedural flow along with the data structures that will be needed? Or do you just start writing code and let it grow organically?
- Is your code well organized and maintainable?
- Do you write unit tests for your new methods before you write the methods?
- Do you have automated tests for your software?

It should be obvious from these questions what the right answers are. Unfortunately, all of us at one time or another are guilty of not taking the time to write tests or document our code. The goal of this document is to give you the basics for improving your focus on planning your code writing, consistency in how you write your code, and increasing your code readability, which should ease the burden of future code maintenance for both you and your co-developers.

Character Encoding

One **common myth** about character encoding is centered around *files*. The notion is, that a *file* must have some sort of character encoding by default. This is wrong. Files are simply containers. They have whatever you put in them, be it binary data, or a special kind of binary data, that we have come to call *text:*

> There's no general way to tell if a file is encoded with a specific encoding. Remember that an encoding is nothing more but an "agreement" how the bits in a file should be mapped to characters. [14]
>
> — Kleiba, responding on Stack Overflow to a question about Windows encoded (windows-1252) files needing to be converted to UTF-8

> In theory, I believe any file is a valid Windows-1252 file, as it maps every possible byte to a character. Now there are certainly characteristics which would strongly suggest that it's UTF-8 — if it starts with the UTF-8 BOM, for example — but they wouldn't be definitive. [22]
>
> — Jon Skeet

Typically, character encoding is of little concern to developers until strange character groups like â€™ or â€œ start appearing on web pages. Leo Notenboom, with "an 18 year career as a programmer at Microsoft," knows this problem all too well. Using the first example above, `â€™`, he explains the process of how this happens, and identifies the two most likely culprits responsible for such a mess: 1) email clients, and 2) Microsoft Word. [17] He is correct in saying that the *right single quote*, a three-byte character in UTF-8 (Hex: E2 80 99) when misinterpreted as another (non UTF-8, single byte) encoding scheme, will be interpreted as three separate characters: "0xE28099 breaks down as 0xE2 (â), 0x80 (€) and 0x99 (™)," however, a contributor with the username of "bob," points out that Notenboom's reference to ISO-8859-1 is not quite on the mark:

> â€™ is not the result of interpreting UTF-8 as ISO-8859-1, but as Windows-1252. Just because € doesn't exist in iso-8859-1 charset! And the charset where 0x80 = € and 0x99 = ™ is windows-1252. [17]
>
> — Notenboom

Indeed, ISO-8859-1 has no mappings at all between Hex 7F and 9F, however, **Windows**-1252 maps perfectly to 0x80 (€) and 0x99 (™) in this triplet.

Even the best configured server will not be able to turn garbled data from a web client into good data in the back end. This type of user-initiated problem will have to be tackled in code in two different situations:

- **Damaged already occurred** to due to **Windows**-1252 mis-decoding and stored incorrectly in the database. If only a few tables exhibit the problem, the following set of `UPDATE` statements in this example demonstrate how the text column(s) can be easily repaired:

```
UPDATE rfi_response SET response = REPLACE(response, 'â€œ', '"');
UPDATE rfi_response SET response = REPLACE(response, 'â€  ', '"');
UPDATE rfi_response SET response = REPLACE(response, 'â€™', '`');
UPDATE rfi_response SET response = REPLACE(response, 'â€˜', "'");
UPDATE rfi_response SET response = REPLACE(response, 'â€"', '-');
UPDATE rfi_response SET response = REPLACE(response, 'â€“', '—');
UPDATE rfi_response SET response = REPLACE(response, 'â€¢', '•');
UPDATE rfi_response SET response = REPLACE(response, 'â€¦', '…');
```

- **Preventative Measures**, such as filtering all text data coming in from client browsers with our `prepPreDB` method defined in the `View` class will keep such mis-decoded mishaps out of the database:

```
public function prepPreDB($string)
```

This becomes especially important for processing form data sent via `POST` from input types like `<textarea>` where users are most likely to copy from Microsoft Office products like Outlook or Word and paste the text into the form.

UTF-8 is the Answer

> Universality is the first and most compelling reason to choose UTF-8. It can handle pretty much every script in use on the planet today.
>
> The real kicker is that by design, UTF-8 is a much more robust and easily interpretable format than any other text encoding designed before or since. First, unlike UTF-16, UTF-8 has no endianness issues. Big-endian and little-endian UTF-8 are identical, because UTF-8 is defined in terms of 8-bit bytes rather than 16-bit words. UTF-8 has no ambiguity about byte order that must be resolved with a byte order mark or other heuristics.
>
> An even more important characteristic of UTF-8 is statelessness. Each byte of a UTF-8 stream or sequence is unambiguous. In UTF-8, you always know where you are — that is, given a single byte you can immediately determine whether it's a single-byte character, the first byte of a two-byte character, the second byte of a two-byte character, or the second or third or fourth byte of a three- or four-byte character. [8]
>
> — Elliotte Rusty Harold, IBM Developer

Database Design

- Database Table names are *singular* (not plural) and are all lowercase with underscores, for example, `acl_attribute_set`
- The first column of each database table should be named `id` and be defined as `unsigned NOT NULL AUTO_INCREMENT`. Additionally, it should be defined as a `PRIMARY KEY`.
- Create new tables with `Storage Engine: InnoDB` and `Collation: utf8_unicode_ci`.
- Create *every* table with a `PRIMARY KEY`. Without it, our in-house PHP Framework does not work as well. Having a `PRIMARY KEY` just makes life easier when debugging.
- When creating `INT, SMALLINT` or `TINYINT` columns, change `Attributes` to `UNSIGNED`.
- If the maximum value of an `INT` column will never exceed 65,000, change the MySQL column type to `SMALLINT`. If it will never exceed 255, then change it to `TINYINT`.
- Design columns functioning as `FOREIGN KEYS` (usually type `INT`) to be `NULL` if there is a possibility that some rows in the table will not have a match on any `PRIMARY KEY` column value of the

foreign (joined) table, since `FOREIGN KEY` constraints will never allow a value of 0 (the default value when inserting a new row with no value specified for a column of type `INT` and `NOT NULL`).

- For columns functioning as a flag for "either/or" cases like `yes/no`, `true/false`, `active/inactive`, `open/closed`, etc., define them as `TINYINT` and use 1 or 0 instead.
- For short lookup tables, consider not creating a separate table at all if there is little possibility of the contents ever changing *and* the lookup values will *not* be used by any other tables. If these criteria match, then set them up as a column of type `ENUM`, for example, `"Open","Investigating","Resolved","Workaround Found","Resolution Not Possible","Closed"`.

Document Your Code!

API Documentation

- All code should be thoroughly documented *before* it is written. The standard commenting style recognized by **Doxygen** should be used.

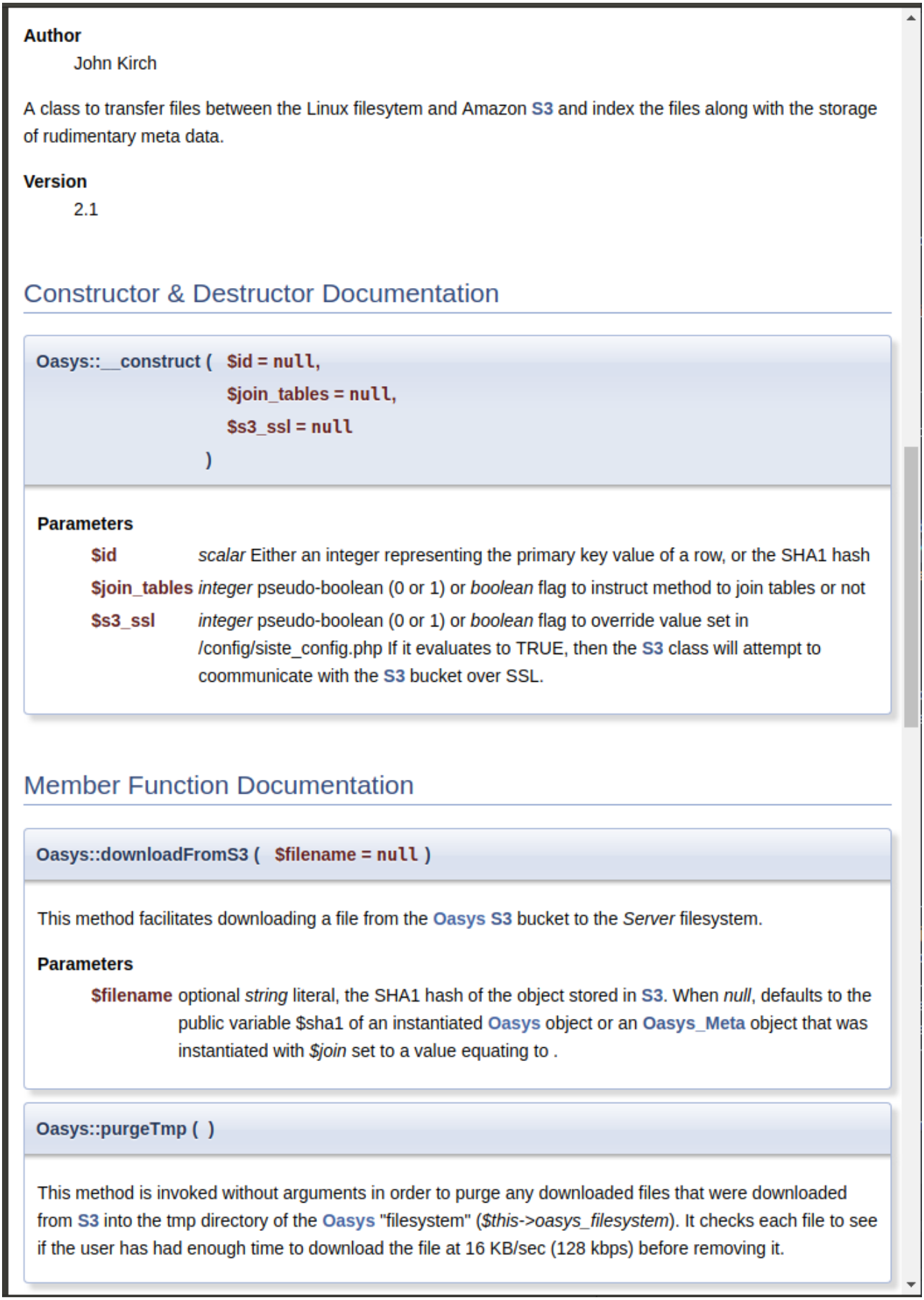
```
require_once 'Solr.php';
// $Id$
/**
 * @brief OASYS = Our Archival System
 * @author John Kirch
 * @details A class to transfer files between the Linux file system and Amazon S3 and
index the files
 * along with the storage of rudimentary meta data.
 * @version 2.1
 */
// $Log$

class Oasys extends Core {
    public $id;
    public $fs;
    public $s3;
    public $sha1;
    public $size;
    protected $child_class = 'Oasys_Meta';
    protected $upgraded_column_names;
    protected $column_aliases;
    protected $s3_obj;
/**
 * @details \b $joined is an array of predefined structure used by Core->byId or Core-
>count() to determine which tables
 * are to be joined, the join relationships and conditions, and which columns from the
foreign tables to be included
 * as "local" columns/public variables in the current class/object.
 * \li Each \b key of a \b$join array is the name of the class representing a foreign
table to be joined. In this case
 * the first join is defined by referencing class name \em Oasys_Filetype
```

```
 * \li Each \b key in the array references a subarray of 4 key/value pairs. The keys
are string value constants:
 * \li \b type defines which type of join to use.  In this case it will be a \b LEFT
\b JOIN
 * \li \b class is \b NULL if the join is between \em this class/table (represented by
the current class, \em Oasys)
 * and the table to be joined.  Notice that the second table to be joined (second key
of this array, \em Oasys_Filetype_Icon)
 * has \b class set to another class/table, i.e. \em Oasys_Filetype because our
class/table \em Oasys has no foreign key
 * column for joining the two tables, thus we must tell the framework which previously
joined class/table can be used for
 * joining, thus the value of 'Oasys_Filetype'
 * \li \b fk_column the column/public variable in \em this class/table that represents
the foreign key on the
 * \b primary \b key \b column of the table to be joined.
 * \li \b columns is set to an \em array of \em string \em literals which represent
the columns of the joined table
 * to be included as public variables when the object is instantiated via the \em
Core->byId method.
 */
    protected $joined = array(
        'Oasys_Filetype' => array(
            'type' => 'LEFT',
            'class' => null,
            'fk_column' => 'filetype_id',
            'columns' => array('ext','mime','filetype'),
        ),
        'Oasys_Filetype_Icon' => array(
            'type' => 'LEFT',
            'class' => 'Oasys_Filetype',
            'fk_column' => 'icon',
            'columns' => array('icon_filename'),
        ),
    );

/**
 * @param $id \em scalar Either an integer representing the primary key value of a
row, or the SHA1 hash
 * @param $join_tables \em integer pseudo- (0 or 1) or \em Boolean flag to instruct
method to join tables or not
 * @param $s3_ssl integer pseudo-Boolean (0 or 1) or Boolean flag to override value
set in
 *         /config/siste_config.php
 *         If it evaluates to TRUE, then the S3 class will attempt to communicate with
the S3 bucket over SSL.
 */
    function __construct($id=null,$join_tables=null,$s3_ssl=null) {
        parent::__construct();
        $join_tables = $this->setDefault($join_tables,$this->join_tables);
        $join_tables = $this->boolval($join_tables);
```

These comments above are rendered by doxygen into beautiful documentation as seen here:



Author

John Kirch

A class to transfer files between the Linux filesytem and Amazon S3 and index the files along with the storage of rudimentary meta data.

Version

2.1

Constructor & Destructor Documentation

```
Oasys::__construct (  $id = null,
                      $join_tables = null,
                      $s3_ssl = null
                    )
```

Parameters

$id	*scalar* Either an integer representing the primary key value of a row, or the SHA1 hash
$join_tables	*integer* pseudo-boolean (0 or 1) or *boolean* flag to instruct method to join tables or not
$s3_ssl	*integer* pseudo-boolean (0 or 1) or *boolean* flag to override value set in /config/siste_config.php If it evaluates to TRUE, then the S3 class will attempt to coommunicate with the S3 bucket over SSL.

Member Function Documentation

```
Oasys::downloadFromS3 (  $filename = null )
```

This method facilitates downloading a file from the Oasys S3 bucket to the *Server* filesystem.

Parameters

$filename	optional *string* literal, the SHA1 hash of the object stored in S3. When *null*, defaults to the public variable $sha1 of an instantiated Oasys object or an Oasys_Meta object that was instantiated with *$join* set to a value equating to .

```
Oasys::purgeTmp (  )
```

This method is invoked without arguments in order to purge any downloaded files that were downloaded from S3 into the tmp directory of the Oasys "filesystem" (*$this->oasys_filesystem*). It checks each file to see if the user has had enough time to download the file at 16 KB/sec (128 kbps) before removing it.

Application Flow Documentation

- In accordance with the best practice of writing the documentation before writing a single line of code, here is an example of how that might unfold:

```
/*    Script to showcase the power of the Framework

    * Focus our demo on Change Orders with Project ID = 5926
    * Define our One-to-Many relationship as 1 Parent Project to many Change Orders
    * Get the IDs of the Change Orders belonging to Project ID 5926
    * Create a subset of Change Order Numbers we wish to display
    * Define which Change Order columns we wish to display
    * Define which Change Order Line Item columns we are interested in displaying
    * FOR EACH Change Order ID now in the list of interesting Change Order IDs
        +    Instantiate an object of class Change_Order using this Change Order ID
        +    IF this Change Order Number is in our list of interesting Change Orders
then show it:
            *    Convert the Change_Order Object into an array of keys => values akin
to the
                arrays returned by MDB2::getRow , i.e. an array of column_name =>
row_value pairs
            *    Set up the Primitive UI/Output for demo purposes
            *    Output the results using the inherited "dump" method
            *    Get the IDs of the Change Order Line Items belonging to this Change
Order
            *    FOR EACH Change_Order_Item, now that we have a list of their row IDs:
                +    Instantiate an object of class Change_Order_Items using this
Change Order Item row ID
                +    Convert the Change_Order_Item Object into an array of keys =>
values akin to the
                    arrays returned by MDB2::getRow , i.e. an array of column_name =>
row_value pairs
                +    Output the results using the inherited "dump" method
            *    END of iteration over each Change_Order_Item
            *    Now that we're finished with this Change Order and its line items,
output a separator line
        + END IF Block for our subset of interesting Change Order Numbers
    * END iteration over the returned Change Order IDs
*/
```

- Once the application flow has been documented in the form of code comments, the process of inserting the actual code becomes a trivial pursuit:

```
// Focus our demo on Change Orders with Project ID = 5926
$project = new Project(5926);

// Define our One-to-Many relationship as 1 Parent Project to many Change Orders
$project->child_class = 'Change_Order';
echo "$project->name - $project->brand [Oracle ID: $project->oracle_id]";
```

```
// Get the IDs of the Change Orders belonging to Project ID 5926
$change_orders = $project->children();

// Create a subset of Change Order Numbers we wish to display
$co_numbers = array(8,11,12,13,16);

// Define which Change Order columns we wish to display
$co_columns = array(
    'number','total_not_to_exceed','overhead_profit','permit_fee',
    'sales_tax','reason_code','po_number','cas_number','processed_date'
);
// Define which Change Order Line Item columns we are interested in displaying
$co_item_columns = array('wbs_code','category','type','units','cost','description');

// FOR EACH Change Order ID now in the list of interesting Change Order IDs
foreach ($change_orders as $co_id) {
    // Instantiate an object of class Change_Order using this Change Order ID
    // This line could also have been written as:
    // $co = new Change_Order($co_id);
    $co = new $project->child_class($co_id);

    // IF this Change Order Number is in our list of interesting Change Orders then
show it
    // See lines 52,53
    if (in_array($co->number,$co_numbers)) {

        // Convert the Change_Order Object into an array of keys => values akin to the
        // arrays returned by MDB2::getRow , i.e. an array of column_name => row_value
pairs
        $co_info = $co->valuesOf($co_columns,$formatted=1);

        // Set up the Primitive UI/Output for demo purposes
        echo "Change Order:";
        // Output the results using the inherited "dump" method
        $co->dump($co_info);

        // Get the IDs of the Change Order Line Items belonging to this Change Order
        // Unlike $project->children(), there is no need to specific a value of
        // $co->child_class because it has only 1 child: Change_Order_Item  which is
        // Pre-defined in the Change_Order class:
        // public $child_class = 'Change_Order_Item';
        $co_items = $co->children();

        // FOR EACH Change_Order_Item, now that we have a list of their row IDs:
        foreach ($co_items as $co_item_id) {
            // Instantiate an object of class Change_Order_Items using this Change
Order Item row ID
            $co_item = new $co->child_class($co_item_id);

            // Convert the Change_Order_Item Object into an array of keys => values
```

```
akin to the
            // arrays returned by MDB2::getRow , i.e. an array of column_name =>
row_value pairs
            $co_item_info = $co_item->valuesOf($co_item_columns,1);

            // Output the results using the inherited "dump" method
            echo "Change Order Line Item:";
            $co->dump($co_item_info);
        } // END of iteration over each Change_Order_Item
        // Now that we're finished with this Change Order and its line items, output a
separator line
        echo "";
    } // END IF Block for our subset of interesting Change Order Numbers
} // END iteration over the returned Change Order IDs
```

Writing Code

Once the documentation has been finished and the procedural flow of the application have been laid out in code comments, a developer might think, the only remaining task would be to focus on the mechanics of writing the PHP code. There are some higher level considerations that need attention before going any further. Questions like, “What about PHP classes functioning as general purpose libraries that are not procedural in nature?” Or, “How should I name my variables?” Also, “How should my PHP scripts be organized on the filesystem relative to each other and to the web web server’s *document root*?”

Categories

Some of the questions above can only be answered within the context of *code purpose*. Once we establish the various categories of PHP scripts, how we name them, where they should be placed within the server’s file system and whether or not they need special access permissions leads us to establish these basic categories:

- **CSS**
  1. Site specific themes
  2. Special purpose CSS styles, such as those specifically for internal documentation
- **HTML** files for static content or includes
- **Javascript** and/or **jQuery** scripts/plugins
- **PHP** code
  1. **Back-end libraries** comprised entirely of classes, for example, our custom PHP framework or tool-specific libraries
  2. **Front-end scripts** directly responsible for generating UX/UI, this is, any scripts devoid of of class definitions. This can also include helper scripts for processing form data send by the user, or Ajax calls. These scripts are typically highly procedural.
  3. Unit Tests

4. **Back-end *scripts*** run via `cron` for data feeds or system maintenance

These categories listed above will be referred to later on when establishing the various best practices specific to them.

Naming Conventions

- **Class names** are CamelCased with an initial uppercase letter.

```
class BadFunctionCallException extends LogicException{}
```

- Class names **defining Database Tables** mimic the table name, but the first letter of each element should be uppercase, for example, the DB table `acl_attribute_set` is represented by:

```
class Acl_Attribute_Set extends Core {
    public $id;
```

- Function names are CamelCased with an initial lowercase.

```
public function byItemIdPriceEffectiveDate($item_id,$date) {
```

- **Avoid prepending the word *get* to Method or Function names** if possible.

```
public function mostRecent($where=null) {
```

  is preferred over

```
public function getMostRecent($where=null) {
```

- **File names** containing PHP Classes following the same pattern as the class name, `MaterialOrder.php`
- Try to have only **one PHP Class per file**, unless the classes form a semantic group that are usually used frequently used together by application code or have strong dependencies on each other.
- **PHP application script filenames** are all lowercase with underscores, for instance, `finalize_bid.php`
- **PHP variable names** follow the same rules as PHP application script filenames: are all lowercase with underscores, for example,on PHP cla `$parent_id_column`
- Use lowercase forms of **reserved keywords** unless PHP documentation uses caps, for example, `true | false | null | self` however **constants** are usually in caps: `ENT_QUOTES | PREG_SET_ORDER`

File Locations

- PHP Back-end libraries: *docroot*`/lib/`
- PHP Front-end scripts: *docroot*`/projects/module/`
- PHP Unit Tests: *docroot*`/lib/unit/`

Editor/IDE Settings

Configure your editor or IDE as follows:

- **UNIX line endings**: `\n`, *not* Windows `\r\n`.
- **Indentation**: The debate over (hard) `TABS` versus *soft tabs* (4 spaces), is anything but new. Each has its pros and cons. But, to be more mainstream and "compatible" with the majority of web development teams and the languages they typically use, *soft tabs* seems to get the popular vote. Consequently, all new code should contain only *soft tabs*.
- **Display Whitespace**: In PhpStorm this can be turned on via [File] [Settings] [Editor] [General] [Appearance]. Trailing whitespace is like an unmade bed. If your IDE has setting to automatically remove trailing whitespace, make sure it is activated.
- **Encoding**: `UTF-8`, *not* `ISO-8859-1`, and most certainly never `Windows-1252`

HTML Entities

When Should One Use HTML Entities? Very rarely. The best practice is to forgo using HTML entities and use the actual UTF-8 character instead. The reasons listed are as follows:

1. UTF-8 encodings are easier to read and edit for those who understand what the character means and know how to type it.
2. UTF-8 encodings are just as unintelligible as HTML entity encodings for those who don't understand them, but they have the advantage of rendering as special characters rather than hard to understand decimal or hex encodings.

> As long as your page's encoding is properly set to UTF-8, you should use the actual character instead of an HTML entity. [5]
>
> — Brendel

Includes

First of all, read up on the differences between `include`, `require`, and `require_once`. Second, note that they are PHP statements (not functions), and you do not need to use parentheses around the filename.

- Use `require_once` for PHP Back-end library files.
- Use `include` for anything else that gets used more than once in the codebase.

CCV of Method Arguments

What is CCV? Just another example of America's love of three-letter acronyms. Actually, for lack of a better label, **C**ontent/**C**ontext **V**alidation of class method arguments is an attempt prevent developers from passing incorrect arguments that could otherwise return invalid results or a fatal error. It should always be employed when an argument requires an `OBJECT` instead of a string, especially if there is a risk of ambiguity in the variable type, like `$user` (id, name, or object?) or, for instance, `$date_onsite_planned` (string, integer, DateTime object, or CustomDateTime object?).

Some of the built-in PHP functions can be employed with some degree of success: `is_array`, `is_bool`, `is_callable`, `is_double`, `is_float`, `is_int`, `is_integer`, `is_long`, `is_null`,`is_numeric`, `is_object`, `is_real`, `is_resource`, `is_scalar`, `is_string`, `isset`. Never use `is_object` when you can lock it down to the specific class with `instanceof`. It should also be noted that `is_a` is a function, whereas `instanceof` is a language construct.

Examples of CCV:

```
public function __construct($id=null,$user=null) {
        parent::__construct();
        if($id && is_numeric($id)){
                $this->byId($id);
        } else if (strlen($id)) {
                $this->byName($id);
        }
        if ($user instanceof User)
                $this->code = $user->language;
}
```

```
class Log extends View {
    public $html;
    public $options = array(
        'input' => 'text',        // or 'html'
        'output' => 'html',     // also 'text' or 'raw'
        'append' => null,        // or 'echo' which will immediately echo the input
    );

    function __construct($options) {
        parent::__construct();
        if (is_array($options) && count($options)) {
            $this->options['input'] = $this->setDefault($options['input'],$this-
>options['input']);
            $this->options['output'] = $this->setDefault($options['output'],$this-
>options['output']);
            $this->options['append'] = $this->setDefault($options['append'],$this-
>options['append']);
        }
    }
```

```
    function __construct($user = NULL, $store = NULL, $project = NULL) {
        parent::__construct();
        if ($user instanceOf User)
            $this->user = $user;
        if ($store instanceOf Store)
            $this->store = $store;
        if ($project instanceOf Project)
            $this->project = $project;
```

Unit Testing

Our Scrum trainer, Mike Cohen, stressed writing unit tests *before* writing the application code. "At the base of the test automation pyramid is unit testing. Unit testing should be the foundation of a solid test automation strategy and as such represents the largest part of the pyramid." [7]

So you might think unit testing is ho hum and just icing on the cake? The PHP developers Mike Naberezny and Matthew Weier O'Phinney at Zend don't share this opinion. Twelve of their slides (38-49) for their presentation on PHP Developer Best Practices were dedicated to the topic. [16] The following are some of the main points – taken verbatim – from those slides (my emphasis added in boldface):

- Untested code can be fragile and prone to regression.
- **No time to write tests? Start writing tests instead of reloading your browser and doing senseless debugging. Increase your productivity and product quality.**
- Start by testing the most critical aspects of your code, strive for testing all of your code. Be practical.
- `PHPUnit` is one of the most feature-rich and widely-used testing frameworks. [4]
- Learning to write good object oriented code that is easily testable takes practice and discipline.
- **Wrapping your functions in classes is not the same as object oriented design.**
- A great deal of PHP code is extremely difficult to test due to poor design. Learn to design for testability.
- Increase your confidence in changes. Your tests will fail if you break something.

To see some examples of PHPUnit testing for our suite of integrated web applicaitons, browse the appropriate relative path.

Designing PHP Classes to Represent MySQL Database Tables

Our in-house PHP Framework offers special Core methods tailor made for querying and manipulating MySQL database tables:

```
public function byId($id,$join_tables=null)
```

```
public function byName($name,$join_tables=null)
```

```
public function byWhere($where=null,
                        $order_by=null,
                        $return_scalar_for_single_row=1,
                        $limit=0,
                        $offset=0)
```

```
public function add($data)
```

```
public function update($new_values,$where=null)
```

```
public function delete($val)
```

```
public function columnSum($column,$where=null)
```

```
public function deactivate($val=null)
```

```
public function mostRecent($where)
```

```
public function count($where)
```

```
public function idName($name_column=null,
                       $case=null,
                       $where=null,
                       $order_by=null)
```

```
public function nameId($name_column=null,
                       $case=null,
                       $where=null)
```

```
public function listOfNames($ids=null,
                            $name_column=null,
                            $case=null,
                            $delimiter=null)
```

```
public function listOfShortNames($ids)
```

```
public function allIds($active=1)
```

```
public function allRecords($active=1)
```

```
public function assoc($where=null,
                      $order_by=null,
                      $limit=0,
                      $offset=0)
```

```
public function duplicates($rec_to_add)
```

These methods require you to set certain public and/or protected variable names in the PHP class defining the MySQL table in order to function properly. The most commonly used special purposed, database-centric PHP variables are:

`$id`, `$active`, `$pk`, `$table`, `$index_column`, `$name_column`, `$shortname_column`, `$active_column`, `$has_active_column`, `$modified_column`, `$delimiter`, `$join_tables`, `$parent_fk_colmn`, `$child_class`, `$data_type` (an array of *key-value* pairs: *public variable name* ⇒ *data type*), `$upgraded_column_names` (an array of *key-value* pairs: *old column name* ⇒ *new public variable name*), `$child_class`, `$group_by`, `$column_aliases` (an array of *key-value* pairs: *public variable* ⇒ *alias public variable name*), `$unique` (a simple array of column names, which when *combined*, must be unique), `$joined` (a complex array of arrays defining table joins).

- Specialized Public (or Protected) Variables with reserved functions within our PHP framework, for example:

```
public $id;
public $active;
protected $order_by = array('date'=>'DESC'); // See class Mim_Price
public $name_column = 'brand';  // See class Brand
public $shortname_column = 'short'; // See class Brand
public $delimiter = ', ';  // See class Brand
public $data_type = array(
    // Valid, supported data_type values are:
    //        'date','datetime','currency','percent','number'
```

```
    'sourcing_approval_date' => 'datetime',
    'vendor_quote_review_date' => 'datetime',
    'vendor_shipping_review_date' => 'datetime',
    'rts_date' => 'date',
    'rts_marked_date' => 'datetime',
    'ship_date' => 'date',
    'shipping_marked_date' => 'datetime',
    'receive_date' => 'date',
    'received_marked_date' => 'datetime',
);
protected $join_tables = 1;
protected $table = 'cmr_lines';
protected $pk = 'cmr_line_id';
protected $has_active_column = 1;
protected $upgraded_column_names = array(
    'cmr_line_id' => 'id',
    'shipping_tracking_information' => 'tracking',
);
protected $parent_fk_column = 'cmr_id';
protected $child_class = 'Project'; // See class Store
protected $group_by = 'cmr_id';
protected $unique = array(
    'cmr_id',
    'item_id',
    'active',
);
protected $column_aliases = array('sku'=>'part_number');
protected $joined = array(
    'Mim_Item' => array(
        'type' => 'INNER',
        'class' => null,
        'fk_column' => 'item_id',
        'columns' => array('description'),
    ),
    'Mim_Parent_Child' => array(
        'type' => 'INNER',
        'class' => 'Mim_Item',
        'fk_column' => 'parent_child_id',
        'columns' => array(),
    ),
    'Mim_Sku' => array(
        'type' => 'INNER',
        'class' => 'Mim_Parent_Child',
        'fk_column' => 'parent_id',
        'columns' => array('sku'),
    ),
    'Vendor' => array(
        'type' => 'LEFT',
        'class' => 'Mim_Sku',
        'fk_column' => 'vendor_id',
        'columns' => array('vendor_id','vendor_name'),
```

```
    ),
);
```

- It is requirement of our PHP framework that the *minimal* `__construct` method be this boiler plate code, where `$id` represents the value of the PRIMARY KEY of a row in the table:

```
function __construct($id=null) {
    parent::__construct();
    if ($id) {
        $this->byId($id);
    }
}
```

- In the case of table with a *unique* name column, the protected or public variable `$name_column` should be set to the name of that column in the schema, and the following boiler plate `__construct` method should include the `elseif` block which facilitates the optional instantiation of an object using a value in that name column:

```
function __construct($id=null) {
    parent::__construct();
    if ($id && is_numeric($id)){
        $this->byId($id);
    } elseif (strlen($id)) {
        $this->byName($id);
    }
}
```

- Use the following boiler plate `__construct` method if the class representing a MySQL database table should have the option to join related tables:

```
function __construct($id=null,$join_tables=null) {
    parent::__construct();
    $join_tables = $this->setDefault($join_tables,$this->join_tables);
    $join_tables = $this->boolval($join_tables);
    if ($id && is_numeric($id)) {
        $this->byId($id,$join_tables);
    } elseif (strlen($id)) {
        $this->byName($id,$join_tables);
    }
}
```

- Any additional input arguments should be in second (or third, if `$join_tables` is present) position, i.e. in *last* or *final* position:

Leverage the Framework

- To build upon the concepts touched upon in the PHP DB Classes section, and to demonstrate the actual execution of the code sample from the previous section on Application Flow Documentation, the efficiency of the using our PHP framework classes and methods shines here.

```
// Perfect Example of Leveraging the Framework
// Let's take a look at how tight this code really is without the comments.
// That's a lot of functionality for only 26 lines of code:
$project = new Project(5926);
$project->child_class = 'Change_Order';
echo "$project->name - $project->brand [Oracle ID: $project->oracle_id]";
$change_orders = $project->children();
$co_numbers = array(8,11,12,13,16);
$co_columns = array(
    'number','total_not_to_exceed','overhead_profit','permit_fee',
    'sales_tax','reason_code','po_number','cas_number','processed_date'
);
$co_item_columns = array('wbs_code','category','type','units','cost','description');
foreach ($change_orders as $co_id) {
    $co = new $project->child_class($co_id);
    if (in_array($co->number,$co_numbers)) {
        $co_info = $co->valuesOf($co_columns,$formatted=1);
        echo "Change Order:";
        $co->dump($co_info);
        $co_items = $co->children();
        foreach ($co_items as $co_item_id) {
            $co_item = new $co->child_class($co_item_id);
            $co_item_info = $co_item->valuesOf($co_item_columns,1);
            echo "Change Order Line Item:";
            $co->dump($co_item_info);
        }
        echo "";
    }
}
```

Defining the `public $data_type` array for certain DB columns that store currency values, percent values, and numeric values in the `Change_Order` and `Change_Order_Item` classes respectively, defining the `Change_Order` class to be the `$child_class` of the `Project` class, defining the `Change_Order_Item` class to be the `$child_class` of the `Change_Order` class, along with the use of the `Core` methods `children()` and `valuesOf($column_names,$format=[0 or 1])` was pivotal in demonstrating the true power of the framework.
Once these tools are put into play, there is hardly a need for writing SQL queries or to format percent values, dollar amounts, or date/time formats coming out of the database:

```
        [total_not_to_exceed] => $3,900.00
        [overhead_profit] => 6.00%
        [permit_fee] => $0.00
        [sales_tax] => $0.00
        [processed_date] => 05/31/2012 06:32 am
        [units] => 109
        [cost] => $20.23
```

Using Git *Wisely*

Revision control systems like CVS, SVN, Git, or Hg play a major role in collaborative software development. Curiously enough, they are even used by non-software developers who need to refine text documents through several iterations of revision and by individual software developers working in their own silo apart from any collaborative projects. The benefits of being able to go back to previous versions of your code, back to a "last known working configuration," or to fork off a new branch and "take a walk on the wild side" with your application without having to destroy a stable working version are enormous. The power of new tools like SVN, Git, or Hg are, however, like a double-edged sword: if not careful, you may end up hurting yourself or your fellow co-developers. With Git's distributed RCS design, a lot of that danger has been mitigated, however, when used unwisely or carelessly, there is still a possibility of shooting oneself in the foot. These best practices should be effective in avoiding those pitfalls:

- ***Know* which branch you currently have checked out**. `git branch` should be easy to find in your Linux (or OS X) command line history. If you already started work and made changes with the wrong branch checked out, there is way to resolve this problem for each of two possible situations:

  1. **The code is new**, i.e. a new file that is not yet being tracked in Git:
     `git checkout <correct branch name>`
     After completing work on the new item(s), do a
     `git add <new_filename>`
     And then a `git commit`

  2. **The code is already being tracked** and attempting to checkout the correct branch results in a Git error message telling you that your changes would be lost if it were to allow you to check out the other branch:
     Complete your work as usual as if this is the correct branch, but note the names of those files you have altered.
     Do a `git add <name of altered file>` for each and every file you have altered, then:
     `git commit`
     Copy the SHA1 hash of this new commit into your Clipboard buffer
     `git checkout <correct branch name>`
     `git cherry-pick <SHA1 hash>`

- **Make a backup copy of your working branch.** Never trust the integrity of the de facto "central Git repository" (usually called *origin*). Git cannot mangle or lose your code, but humans can make mistakes when using Git, like pushing untested code to the central repository, so you need to be cautious. This is actually the very reason we switched to Git. At least we have enough

copies of working branches in our 7 or 8 repositories that recovery is always an easy task. With a non-distributed, "Central Repository" architecture like Subversion (SVN), such a human error can have catastrophic consequences. At any rate, once you have your branch in a good working state, make a backup of that branch with some name you will remember, like *my_prod* or maybe *prod_stable*: + `git branch -f prod_stable`

- **Do only `git pull` and hit `Return`** now that our Git `config` files in our sandboxes have been rewritten to *always* pull from the correct remote branch *automatically*. **Do not specify the remote repository and branch.** This is dangerous and very prone to user error:

`git pull origin <branch>` must be the *same* branch name as the one you currently have checked out, otherwise you will be *merging* changes from the wrong branch into your current branch, a change that unfortunately is very time consuming to rectify if a backup — as described immediately above — has not been created.

- **When composing a comment for a commit, write as much detail as you can.** Comments like "fixed some stuff" are counterproductive and help no one, including yourself when you eventually have to go back and maintain that code later on.
- **Always include the correctly formatted Pivotal Tracker ID in your Git comments**, for example `[#74894116]`
- Which is better? Fewer commits with maybe hundreds of lines of new or changed code? Or, several commits, each containing only related code changes? It depends:

  1. **For *new* tools or features that have not yet been released**, ever, for which you will will be using `git add` to get them into Git, **bundle them all up into a single commit.** If you've already been doing incremental commits in your sandbox Git repository, that is great. It is in itself a "best practice." Just use `git log` to count how many of these incremental commits you have made. For example, you just made 12 commits, now you can perform an interactive *rebase* to roll them into a single commit:
     `git rebase -i HEAD~12`
     during which you will be thrown into `vi` to edit all the lines except the first line (the oldest commit) and change the command in the first column to be a single letter "s" which stands for "squash." After saving this with `Esc :wq`, you will once again be thrown into `vi` to edit the comments, which if your original comments were good, all you will need to do is remove some empty lines and Git hints/labels inserted by Git Rebase, otherwise you'll need to compose a comprehensive set of comments from scratch. After saving the comments with `Esc :wq`, the interactive *git rebase* process should be complete.

  2. **For all other situations, bug fixes, updates, upgrades, and minor new features to existing tools,** the best practice is to have a 1-to-1 relationship between each single fix, update, minor new feature and its commit. In other words, do *not* bundle unrelated features, updates, or bug fixes into a larger commit. If one ever has to back out of a code update because it is not working as planned, then it will be much easier to remove a single "unsuccessful feature" by itself than to have to remove, for instance, a *combined* commit of "unsuccessful feature" + "successful bug fix" since this will require re-editing the code to put the "successful bug fix" back into the codebase, and then making a new separate commit for the "successful bug fix." **In short, for most situations, frequent, single-topic commits are the best practice.** [10]

- Before you commit changes, *know* what you are really committing: `git diff`
- Always perform a `git pull` *before* your `git push all`
- Try to maintain a consistent, preventative workflow, for example:
  1. Check out your `prod` branch
  2. Fix some bug in the code based on a bug report in Pivotal Tracker
  3. Test your bug fix thoroughly
  4. Commit and push your changes to the central Git repository for someone else to QA:

```
git branch                 # to verify you are on the correct branch, in this
case "prod"
git branch -f prod_working  # to make a backup of a known "good" prod branch
git pull                   # retrieve any new commits that may have been pushed
to origin
git add filename.php       # substitute filename.php with the real filename of
the file you altered
# Include the Pivotal Tracker ID in your Git comment
git commit -m '[#74894116] Enabled Directors access to comment on RFI'
git push all
```

- **Merge Conflicts** and How to resolve them

  Git inserts 3 lines of demarcation in each source code file, in which the changelog has found conflicts that cannot be resolved by standard logic. In these cases human intervention is required.

```
<<<<<<< HEAD:mergetest
    $test = array();
    $count = 0;
    $max_num = 5000;
    $debug = false;
=======
    $temp = array();
    $i = 0;
    $max_num = 10000;
    $debug = true;
>>>>>>> 4e2b407f501b68f8588aa645acafffa0224b9b78:mergetest
```

  `<<<<<<<` indicates the start of the lines that had a merge conflict. This first block of code is from the file (usually labeled HEAD) that you were trying to merge the changes into. `=======` Indicates the break point used for comparison. It separates the changes that have been committed (the first block of code above this double-line) from the changes coming from the merge (the second block of code below this double-line) to visually see the differences. `>>>>>>>` Indicates the end of the block of code that had a merge conflict. Conflicts can only be resolved by manually editing the file. This is typically accomplished by choosing one of the two code blocks to keep and discarding the other one.

The three lines inserted by Git `<<<<<<<`, `=======`, and `>>>>>>>` must be removed or you will be pushing broken code to team members, or even worse, to the live, production site!

The easiest way to check for any remaining lines of merge conflict demarcation is to leverage git grep:

```
$ git grep -nI '<<<<<'
$ git grep -nI '>>>>>'
test.php:187:>>>>>>> 16153ab7cee2bfcd73023caae4e00ffa21868c5c
test.php:403:>>>>>>> 16153ab7cee2bfcd73023caae4e00ffa21868c5c
```

In this example, it becomes readily apparent that the PHP script file test.php had two different merge conflicts. Although the two conflicting blocks of code appear to have been resolved, the failure to remove the third line of merge conflict demarcation in both cases will result in a fatal syntax error if the web page is accessed.

Database Revision Control

In his blog, *Coding Horror*, Jeff Atwood writes,

> When I ask development teams whether their database is under version control, I usually get blank stares.... When it comes to version control, the database is often a second or even third-class citizen.... I don't know how you can call yourself a software engineer and maintain a straight face when your database isn't under exactly the same rigorous level of source control as the rest of your code. [3]

Atwood's co-author, Scott Allen, asserts there are three rules for successfully working with databases. [1]

1. Never use a shared database server for development work.
2. Always Have a Single, Authoritative Source For Your Schema.
3. Always Version Your Database.

Fortunately, we have been adhering to the first two rules for years now. Each developer works in their own sandbox with their very own database. The single authoritative source for our schema has traditionally been the production database, once changes have been tested, QA'ed, and implemented on the production machine. It's the third rule that has been a gray area for some time. Daily snapshots of the production database and replication to a remote database server are better than nothing, but there is certainly room for improvement.

In our shop, we basically have two very different categories of database changes that need archival:

1. Day-to-day end-user transactions, such as new bids, orders, approvals, alerts, etc.

2. Structural changes to the database (new tables, indices, column data type changes, etc.) and changes to tables whose primary function is data normalization, such as new records or updates to tables like `users`, `brand`, `oasys_folder`, etc.

The first category is so extensive and volatile with end-users constantly creating and updating orders and requests 24x7 that the simplest and most effective approach is a combination of real-time replication to a remote database server functioning as a slave along with daily snapshots using the following naming convention:

```
mysqldump -uroot -ppassword db_named_foo > yyyymmdd_His-db_named_foo.sql
```

The second category, however, is one of very special concern for developers and testers. In Allen's related article on versioning databases, he mentions *change* scripts:

> By "change", I mean a change to a table, index, key, constraint, or any other object that requires DDL, with the exception of views, stored procedures, and functions.
>
> — Allen

His change scripts seem to be focused on the same database changes described in this second category. For these types of changes we use a second bare Git Repository called "sql" located under `/opt/git/sql.git`.

Creating DBRC Scripts

It is an extremely bad practice to do one's database development work in the Production Database. In many cases, it is not even possible without disrupting normal website operations for our end users. No matter where the iterative design process is happening, it does not lend itself to collaboration or the tracking of design/structural changes over a long period of time. By leveraging Git and creating SQL scripts to be versioned, these roadblocks are removed.

In the "sql" Git repository you cloned under your home directory, you will need to create robust SQL scripts that can be reloaded without causing errors or loss of existing data. **There are basically three kinds of scripts/situations you will need to master** until this process becomes automated (or a web front end is written for it):

1. **Basic table changes that can be loaded repetitively without causing errors.** This type will mostly consist of changing data or collation type of existing columns. Here is a good example:

convert_users_to_utf8.sql

```
/* Upgrade users table to UTF-8 from Latin-1 */
/* This is to ensure that old data in Latin-1 encoding gets correctly converted to
UTF-8 without corruption. */
ALTER TABLE  `users` DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci;
ALTER TABLE `users`
  CHANGE `username` `username` VARCHAR(30) CHARACTER SET utf8 COLLATE
utf8_unicode_ci NULL DEFAULT NULL,
  CHANGE `password` `password` VARCHAR(64) CHARACTER SET utf8 COLLATE
utf8_unicode_ci NULL DEFAULT NULL,
  CHANGE `user_hash` `user_hash` VARCHAR(64) CHARACTER SET utf8 COLLATE
utf8_unicode_ci NULL DEFAULT NULL,
  CHANGE `name` `name` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_unicode_ci NULL
DEFAULT NULL,
  CHANGE `first_name` `first_name` VARCHAR(64) CHARACTER SET utf8 COLLATE
utf8_unicode_ci NULL DEFAULT NULL,
  CHANGE `last_name` `last_name` VARCHAR(64) CHARACTER SET utf8 COLLATE
utf8_unicode_ci NULL DEFAULT NULL,
  CHANGE `title` `title` VARCHAR(64) CHARACTER SET utf8 COLLATE utf8_unicode_ci
NULL DEFAULT NULL,
  CHANGE `email` `email` VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_unicode_ci
NULL DEFAULT NULL,
  CHANGE `type` `type` VARCHAR(30) CHARACTER SET utf8 COLLATE utf8_unicode_ci NULL
DEFAULT NULL,
  CHANGE `wwr_import_name` `wwr_import_name` VARCHAR(255) CHARACTER SET utf8
COLLATE utf8_unicode_ci NULL DEFAULT NULL,
  CHANGE `brands` `brands` VARCHAR(50) CHARACTER SET utf8 COLLATE utf8_unicode_ci
NULL DEFAULT NULL,
  CHANGE `zone` `zone` VARCHAR(50) CHARACTER SET utf8 COLLATE utf8_unicode_ci NULL
DEFAULT NULL;
```

2. **Structural additions** (columns, indices, etc.) to a table **that would normally cause errors or fail if run repetitively** on the same table. An example:

   create_users_add_language.sql

```
/* Add the new column "language" in a way that MySQL will not throw an error if the
column already exists. */
/* Technique developed by Nariman Shariat */
SET FOREIGN_KEY_CHECKS=0;
DROP TABLE IF EXISTS `temp_users`;
CREATE TABLE temp_users LIKE users;
INSERT INTO temp_users SELECT * FROM users;
DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
  `user_id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(30) COLLATE utf8_unicode_ci DEFAULT NULL,
  `password` varchar(64) COLLATE utf8_unicode_ci DEFAULT NULL,
  `user_hash` varchar(64) COLLATE utf8_unicode_ci DEFAULT NULL,
```

```
  `name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `first_name` varchar(64) COLLATE utf8_unicode_ci DEFAULT NULL,
  `last_name` varchar(64) COLLATE utf8_unicode_ci DEFAULT NULL,
  `title` varchar(64) COLLATE utf8_unicode_ci DEFAULT NULL,
  `email` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `type` varchar(30) COLLATE utf8_unicode_ci DEFAULT NULL,
  `region_id` int(11) DEFAULT NULL,
  `wwr_import_name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `brands` varchar(50) COLLATE utf8_unicode_ci DEFAULT NULL,
  `zone` varchar(50) COLLATE utf8_unicode_ci DEFAULT NULL,
  `vendor_id` int(10) unsigned DEFAULT NULL,
  `preferred_name` varchar(100) COLLATE utf8_unicode_ci DEFAULT NULL,
  `profile_organization_id` int(11) DEFAULT NULL,
  `profile_country_id` int(11) DEFAULT NULL,
  `profile_city_id` int(11) DEFAULT NULL,
  `language` enum('en_US','en_GB','zh_CN','es_ES','fr_FR','it_IT','ja_JP')
              COLLATE utf8_unicode_ci NOT NULL DEFAULT 'en_US',
  `active` int(1) DEFAULT '1',
  `notify` tinyint(3) unsigned NOT NULL DEFAULT '0',
  `last_access` datetime DEFAULT NULL,
  PRIMARY KEY (`user_id`),
  UNIQUE KEY `username` (`username`),
  KEY `first_name` (`first_name`),
  KEY `last_name` (`last_name`),
  KEY `title` (`title`),
  KEY `last_access` (`last_access`),
  KEY `notify` (`notify`),
  KEY `vendor_id` (`vendor_id`),
  KEY `profile_organization_id` (`profile_organization_id`),
  KEY `profile_country_id` (`profile_country_id`),
  KEY `profile_city_id` (`profile_city_id`),
  KEY `language` (`language`),
  CONSTRAINT `users_ibfk_1` FOREIGN KEY (`vendor_id`)
        REFERENCES `vendors` (`vendor_id`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `users_ibfk_2` FOREIGN KEY (`profile_organization_id`)
        REFERENCES `choice_attributes` (`id`) ON DELETE SET NULL ON UPDATE SET
NULL,
  CONSTRAINT `users_ibfk_3` FOREIGN KEY (`profile_country_id`)
        REFERENCES `choice_attributes` (`id`) ON DELETE SET NULL ON UPDATE SET
NULL,
  CONSTRAINT `users_ibfk_4` FOREIGN KEY (`profile_city_id`)
        REFERENCES `choice_attributes` (`id`) ON DELETE SET NULL ON UPDATE SET NULL
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci
COMMENT='All user data is stored';
INSERT INTO users (`user_id`,`username`,`password`,`user_hash`,`name`,
  `first_name`,`last_name`,`title`,`email`,`type`,`region_id`,`wwr_import_name`,
  `brands`,`zone`,`vendor_id`,`preferred_name`,
  `profile_organization_id`,`profile_country_id`,`profile_city_id`,
  `active`,`notify`,`last_access`)
SELECT `user_id`,`username`,`password`,`user_hash`,`name`,
  `first_name`,`last_name`,`title`,`email`,`type`,`region_id`,`wwr_import_name`,
```

```
  `brands`,`zone`,`vendor_id`,`preferred_name`,
  `profile_organization_id`,`profile_country_id`,`profile_city_id`,
  `active`,`notify`,`last_access` FROM temp_users;
/* DROP TABLE IF EXISTS `temp_users`; #taking this out to have a safety net */
SET FOREIGN_KEY_CHECKS=1;
```

3. **Changes to the data only:** the most common situation is the addition or changes to data in common lookup tables used mainly for normalizing data, i.e. tables to which form data submitted by users is never stored. This is perhaps the easiest to write because `mysqldump` generates the SQL for you. It just needs to be sanitized in order for real changes to the table data to be recognized as such by Git, for instance, these types of nonessential additions to the SQL by `mysqldump` need removal:

```
/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
-- MySQL dump 10.13  Distrib 5.5.35, for debian-linux-gnu (x86_64)
-- Host: localhost    Database: db_named_foo
-- Dump completed on 2014-07-25 21:19:10
```

Obviously we do want to track a change in our scripts just because we upgraded MySQL to a more recent release. Nor do we want to track which sandbox database the data comes from if the data has not changed at all. Lastly, MySQL's timestamp of when the dump completed will generate a change in the file for Git every single time. **Again, we only want to track changes in the data.**
Also, it will be common that we will need consolidate multiple tables into a single, loadable SQL script that represents all the database changes that form a logical unit within the context of a new feature or new tool that is being developed. To illustrate this, the `i18n` tables we use for storing language specific data will be used as an example of the build process:

1. **Create the Shell Script** to generate and concatenate the *sanitized* output from `mysqldump`:

   i18n.sh

   ```
   #!/bin/sh
   echo "SET FOREIGN_KEY_CHECKS=0; " > create_i18n_tables.sql
   mysqldump -uroot -p'password' --compact --add-drop-table db_named_foo i18n_content
   | grep -v '^\/\*![0-9]\{5\}.*\/;$' >> create_i18n_tables.sql
   mysqldump -uroot -p'password' --compact --add-drop-table db_named_foo i18n_language
   | grep -v '^\/\*![0-9]\{5\}.*\/;$' >> create_i18n_tables.sql
   echo "SET FOREIGN_KEY_CHECKS=1; " >> create_i18n_tables.sql
   ```

2. **Set the permissions on the shell script for execution**. Using the filename from the example above:
   `chmod 750 i18n.sh`

3. **Run the shell script**: `./i18n.sh`

If these steps were followed correctly, an new SQL file named `create_i18n_tables.sql` should have been created.

Adding the DBRC Scripts to Git

Moving forward with the examples above, four new scripts have been created and need to be tracked by Git.

```
$ git add convert_users_to_utf8.sql
$ git commit -m 'Upgrade users table to UTF-8 from Latin-1'
[master 47516df] Upgrade users table to UTF-8 from Latin-1
 1 file changed, 16 insertions(+)
 create mode 100644 convert_users_to_utf8.sql
$ git add create_users_add_language.sql
$ git commit -m 'Add new column "language" to the users table which is a prerequisite
for i18n'
[master 8a7cfba] Add new column "language" to the users table which is a prerequisite
for i18n
 1 file changed, 66 insertions(+)
 create mode 100644 create_users_add_language.sql
$ git add i18n.sh
$ git commit -m 'Updated i18n.sh to include convert_users_to_utf8.sql and
create_users_add_language.sql'
[master a8b8073] Updated i18n.sh to include convert_users_to_utf8.sql and
create_users_add_language.sql
 1 file changed, 8 insertions(+), 6 deletions(-)
 rewrite i18n.sh (78%)
$ git add create_i18n_tables.sql
$ git commit -m 'Updated i18n tables to include new content db_named_foo added for
recent testing'
[master 0cd7646] Updated i18n tables to include new content db_named_foo added for
recent testing
 1 file changed, 18 insertions(+), 3 deletions(-)
$ git push origin master
Counting objects: 15, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (12/12), 2.85 KiB, done.
Total 12 (delta 7), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To /opt/git/sql.git
   3e16011..0cd7646  master -> master
```

Deploying and Implementing the DBRC Scripts

1. **Pull the latest changes from our de facto "central" *sql* Git repository**

```
$ git pull origin master
remote: Counting objects: 18, done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 15 (delta 8), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From s3.our-domain-name.com:/opt/git/sql
 * branch            master     -> FETCH_HEAD
Updating e88bebb..0cd7646
Fast-forward
 convert_users_to_utf8.sql     |   16 ++++++++++
 create_i18n_tables.sql        |   21 ++++++++++++--
 create_users_add_language.sql |   66 ++++++++++++++++++++++++++++++++++++++++++++
 i18n.sh                       |    8 +++--
 load_sql                      |   13 ++++++++
 5 files changed, 118 insertions(+), 6 deletions(-)
 create mode 100644 convert_users_to_utf8.sql
 create mode 100644 create_users_add_language.sql
 create mode 100755 load_sql
```

2. **Implementation of a DBRC script** involves invoking the `load_sql` script which accepts two arguments: *script_name* and *recipient_database_name*

```
./load_sql create_i18n_tables.sql db_named_foo

Loading SQL script: create_i18n_tables.sql
into MySQL Database:   db_named_foo

-----------------------------------
Elapsed Time: .460401884 seconds
-----------------------------------
```

Best Practices

- **Pull changes from the sql repository *before* you do any type of design work.** If you see changes, load them. This will save yourself some headache and grief later on.
- **Notify the entire team if you are planning to update/add data to any common look-up tables.** It will be impossible for Git to merge MySQL data dumps due to the `PRIMARY KEY` values being using in the SQL inserts within the dump. Thus, communicate your estimated delivery time in case others plan to work with the same data, such as `i18n_language`. The *only* way to manage this kind of situation is to take turns and go through the entire workflow after each data update. This example shows what multiple developers can collaborate on adding new translations to the `i18n` tables:

```
$ cd ~/sql
$ git pull
$ ./load_sql create_i18n_tables.sql db_username   # Insert your username to match your
DB's name; this loads new changes
# Do your work on inserting or updating records
$ ./i18n.sh                         # Create your new DBRC sql script
$ git add create_i18n_tables.sql    # Tell Git you want to commit this changed file
$ git commit -m "Added new data"    # Commit the change with an appropriate, detailed
comment about what you did and why you did it
$ git push                          # Push the changes
```

Whenever you push changes to a Git Repository shared by the entire team, make sure to send us a chat message indicating which repository, branch, and some details about what changes you have pushed.

And alternative would be to use `db_test` as a central database – as opposed to using one's sandbox database – for highly volatile tables that need frequent changes by multiple developers. This still, however, does not alleviate the need for generating the DBRC sql script used to track changes. It simply reduces the chance of merge conflicts when developers fail to communicate their plans or work collaboratively.

Agile, Scrum and Project Management

Of the three elements necessary for successful software development (Object Oriented code, Revision Control, and Agile Methodology), Agile is arguably the most important of them all. It allows a team to *manage the expectations* of stakeholders: you know, the people are typically footing the bill for your work. If they are unhappy, you will be unhappy.

Whichever methodology a team chooses to implement (Scrum, Kanban, Scrum-ban, etc.) it is important for all team members to understand the process, the policies entailed, and to be consistent in the implementation of them.

Once the standard length of a sprint has been chosen, roles assigned, and education of the entire team on the theory of your specific agile methodology has been completed, the next step is finding some software that will be essential to tracking stories, tasks, ownership, and the various levels of "state". After trying on Agilo trac, JIRA, Pivotal Tracker, and Planbox for size, we finally decided Pivotal Tracker was a keeper.

Roles and Functions

In scrum there are roles that often cover multiple functions in the traditional waterfall methodology. For instance, scrum does not recognize a full-time QA role as a scrum team member. Anyone can test the product. A scrum team is technically comprised of only three roles: scrum master, product owner, and the development team. The stakeholder role is part of scrum, but stakeholders are not part of the scrum team. It is common practice, however, that software development organizations using scrum will assign a specific role or function to each member of

the *development team* usually dependent on each individuals area of technical expertise. Scrum "development teams are cross-functional, with all of the skills as a team necessary to create a product increment." #wikipedia:scrum

QA Analyst within Scrum

As defined by agile/scrum methodology in the previous section, the entire responsibility of QA and testing in general does not fall entirely on the QA analyst. In fact, due to the highly collaborative requirements of scrum, the QA analyst should have less work to do in the trenches since a good deal of testing should have already been performed by the developers, especially regarding unit testing. Also the application and UI testing should also have been thoroughly performed by the developers prior to handing the product off to the QA analyst for testing.

Scrum does, however, place some extra responsibilities on the QA analyst not normally found in non-agile methodologies: the need to fully understand the business rules and functional requirements — both of which comprise the *acceptance criteria* — which necessitates collaborating with the product owner, who typically has a good understanding of the business rules, and with the developer(s), who normally will be scoping out these product requirements in terms of functional requirements within the scope of database structures and code changes:

> After working for nearly two years as a quality assurance (QA) analyst on a Scrum team, I have learned that the role of QA in Scrum is much more than just writing test cases and reporting bugs to the team.
>
> The QAs can pair up with developers for writing unit test cases and for discussing acceptance criteria. The more these roles work together, the greater the shared clarity will be on requirements. The increased clarity that results from working together will reduce the questions and doubts developers often encounter during coding time, which produces greater efficiency and a big time savings for developers and testers alike. [9]
>
> — Priyanka Hasija, My Experience as a QA in Scrum

Working Towards Delivery

Knowing when software is ready to be released is the biggest challenge of all. The best QA skills and acute attention to detail are worthless if testing is focused in the wrong places. Scrum courses always touch on the need for testing, but they rarely provide a recipe for the logistics of it. How do you write the test cases? Who is responsible for writing test scenarios? What is the difference between *acceptance criteria* and a *test scenario*?

First of all, some definitions are in order. At the top of the food chain are the *acceptance criteria* which are usually driven by a *business rule*. That rule can be defined in terms of a *functional requirement*. Functional reqirements get instantiated, ususally by a QA analyst, as a set of *test scenarios*. *Efficient acceptance tests are all that's required* provides a very easy to understand real life example of these concepts from a software application written for the insurance industry. [25]

This is *required* reading for the entire scrum team since the definitions of *business rule* versus *functional requirement* versus *test scenario* and real life examples of each must be easily understood and recognized for the logistics of "working towards delivery" — within the context of agile/scrum — to make sense.

Each and every new feature, i.e. each user story needs to be discussed by the developer assigned to the story, the product owner, and the QA analyst. **Each of these three roles need to formally organize a meeting**, if only 15 minutes long, to collaborate and reach a consensus on:

1. The validity of the *acceptance criteria*, that is, to enforce clarity, verify each team member has the *same* understanding of those criteria, and if not, reword/redefine the *acceptance criteria* as needed.
2. The developer with ownership of the story will need to express his/her vision of the *functional requirement(s)*, for instance, the *new* logic, the *new* functionality, or the *new* behavior the page(s) will exhibit once his/her code and/or database changes have been realized. This vision is very crucial. The product owner will need to evaluation this vision and ensure it is *precisely* on target with the *acceptance criteria*. If this vision is not spot on, product owner and developer will need to discuss, explain, and collaborate until a consensus is reached.
3. Finally, the QA analyst will take a more active role in this meeting, having actively processed the *acceptance criteria* and *functional requirement(s)* presented by the team, by presenting a rough, ad hoc example of a single iteration of the QA *test scenario* that should effectively test the proposed code changes and/or database changes for the desired effects as defined by the developer's *functional requirement(s)*. The product owner will confirm that the QA analyst has presented a minimum list of roles and/or project roles that are essential to this story. Both the story owner (developer) and product owner will call out any missing test, unnecessary tests, or any other deficiencies in the proposed test scenario.

At the end of the *WTD Meeting* (**W**orking **T**owards **D**elivery), **each of the three participants will need to update the Pivotal Tracker story with more fleshed out versions of what they presented.**

> **Having a clear Definition of Done (DoD) is important to a Scrum team.** A DoD is nothing more than a simple list of team defined completion criteria — all of which must be completed before the user story can be considered done. This list typically includes things such as writing code, performing functional and regression tests, and gaining the approval of the Product Owner. A very simple DoD might include the following:
>
> • Code Complete
> • Unit Test complete
> • Functional / UI Test Complete
> • Approved by Product Owner [9]
>
> — Priyanka Hasija, My Experience as a QA in Scrum

For our team, a more comprehensive, detailed *Pre-Delivery Checklist* is recommended. It's too easy to forget a critical step:

The Pre-Delivery Checklist

Before clicking on Deliver you need to confirm you completed this check list.

- ▢ Write unit tests if applicable, i.e. if any new classes or methods need to be written
- ▢ Write API documentation if applicable
- ▢ Create new classes and methods if applicable
- ▢ Run unit tests
- ▢ Design database tables if applicable
- ▢ Create DBRC scripts if the database needs modifications
- ▢ Add DBRC scripts to the “sql” Git repository
- ▢ Document the application flow as comments in the source code file(s)
- ▢ Write the application code interspersed with comments copied and pasted from the application flow comments
- ▢ Perform application testing using the test scenario you composed in Pivotal Tracker
- ▢ Debug code if applicable
- ▢ Commit the Code in the *current* sprint branch
- ▢ Deploy the DBRC scripts in the test environment
- ▢ Push the code to the *origin*, *test*, and *github* git repositories using `git push all`
- ▢ Perform application testing in the test environment using the test scenario
- ▢ Change ownership to the name of the tester upon *successful* completion of application testing
- ▢ Click on `Deliver`

Pivotal Tracker

- **Compose concise tickets.** It is impossible to be *too* concise. Never assume anything is clear. Always assume you will need to state the obvious.
- **New features (stories) need tasks** unless they are so simple or specific that a only a single task can be written.
- **Every Ticket needs *clear* acceptance criteria** written for it, placed in the ticket description field (normally at the end of the description) with an appropriate heading. Here is a example taken from a ticket in Pivotal Tracker:

Acceptance Criteria:
In addition to the functional requirements stated above, it is important that the display function always return some text, since every page has to have content without any blanks/missing text, and there will be times when a translation is not yet available, so the method will need to account for this condition by failing over to a the source content language, English, and the default locale, `en_US`.

In addition to the explicit Acceptance Criteria provided by ticket authors in Pivotal Tracker, *any single violation of any best practice mentioned in this document is sufficient reason for the tester/approver to* ***reject*** *a ticket.*

- **Write your test scenario.** If your story type is listed as a *Feature*, the very first thing you should do before writing any code is plan out how you will test it based on the Acceptance Criteria. This should be the very first comment in any new ticket of type *Feature*. A typical test scenario might look like this:

Sample Test Scenario

1. Navigate to test page http://test.our-domain-name.com/projects/10852/mto, logged in as username: `richardburton`

Make sure you list a working, live URL in every Test Scenario, and that the URL is pointing to the **Test Environment**, *not* your sandbox!

2. Navigate to the *Invited Vendors:* section
3. Click on the *Invite Vendors* link
4. An "OFI Vendor List" modal should appear containing a list of vendors with a checkbox next to each one. If not, terminate test and report FAILED.
5. Check one of the vendors and then click on `Invite`.
6. The modal should disappear and the page should show a message in read text should appear above the table: "You have successfully invited 1 vendor(s)." If not, terminate test and report `FAILED` on step 6.
7. In the "Vendor Name" column you should be able to find the newly invited vendor's name listed. If not, terminate test and report `FAILED` on step 7.
8. In the "Date Invited" column you should be able to find today's date listed on the same row as the name of the newly invited vendor. If not, terminate test and report `FAILED` on step 8.
9. If "Date Invited" is in a format inappropriate for the locale of the user, for example, the user's language is `en_US` and the date is not in *mm/dd/yyyy* format, terminate test and report `FAILED` on step 9.
10. In the "Invited By" column you should be able to find "Richard Burton" listed on the same row as the name of the newly invited vendor. If not, terminate test and report `FAILED` on step 10.
11. Report `PASSED`.

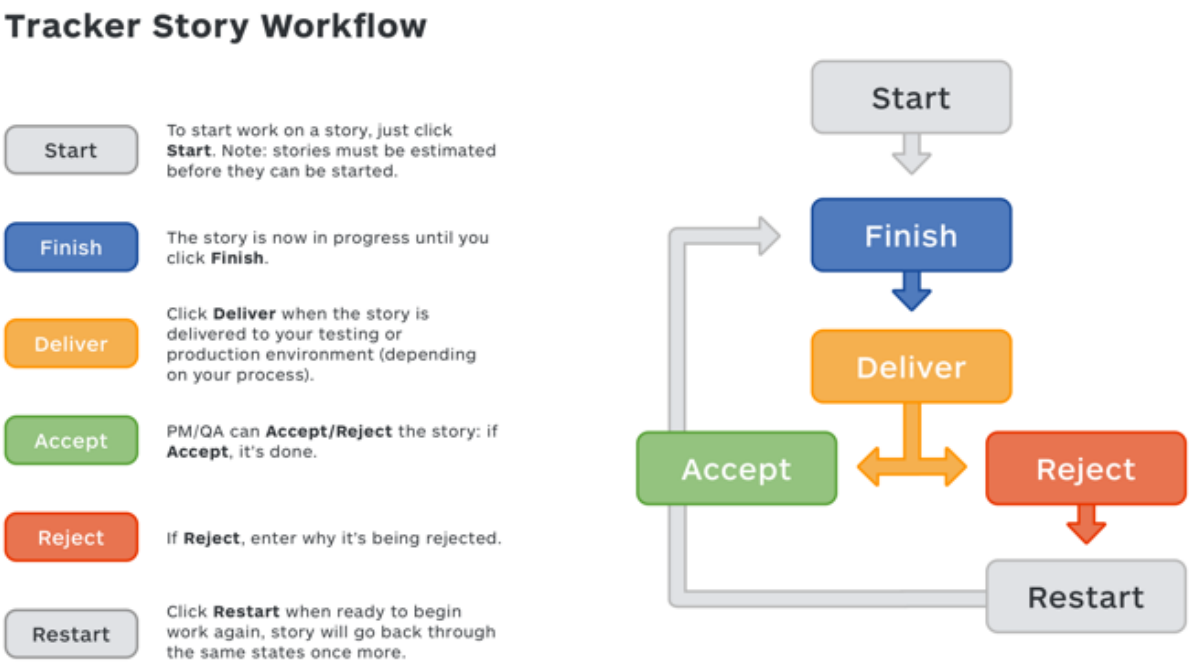
- **Ownership of a ticket** should be clear. Notify team members if you plan to hand it off to another developer. Be *specific* in your requests for assistance. Write a new comment in PT with @NameOfPerson you are asking for assistance and choose one of these templates:

1. Could you please complete task "foo" of this story? I plan to retain ownership.

2. I wish to permanently hand this entire story/bug/chore off to you and request that you assume ownership.

- **Start your tickets!** Don't click on `Start` until you are actually working on it. Likewise, don't leave it with `Start` visible once you have commenced work on it. Once you click `Start` it will then display `Finish` (the next *pending* status).
- **Check off Tasks** as they become completed. Project tracking software is only useful if a team actually uses it as designed. Once all tasks have been completed, the ticket will a show a pair of buttons: `Accept` `Reject`
- **Restart your tickets!** If for some reason your **Delivery** of a story was **Rejected** `Restart` will appear. Read the comments to see why. If they are not clear, contact the tester/approver of the ticket and get clarification. Once clarified, get back on the task(s) and revert the status of the ticket back to *active* once you have commenced work on the resolution. The button should then change back to `Finish` i.e. the next *pending* status.
- Click on `Finish` when you are finished writing your code and you have thoroughly tested it.
- Use Git to pull any new changes into your sandbox's repository.
- Push your completed work to the "Test" repository.*
- Click on `Deliver`

The Various Story States in Pivotal Tracker



References

- 1 Allen, K. Scott. Three Rules for Database Work. *Ode to Code*, 31 Jan 2008.
- 2 Allen, K. Scott (Feb08). Versioning Databases – Change Scripts. *Ode to Code*, 2 Feb 2008.
- 3 Atwood, Jeff. Get Your Database Under Version Control. *Coding Horror*, 2 Feb 2008.

- 4 Bergmann, Sebastian: PHPUnit. Getting Started with PHPUnit. Siegburg, Germany, 2014.
- 5 Brendel, William. Best answer to When Should One Use HTML Entities on StackOverflow, 12 Jan 2009.
- 6 Cabal, Alex: PHP Best Practices – A short, practical guide for common and confusing PHP tasks. 30 Apr 2013, revised 3 Jul 2014
- 7 Cohen, Mike: The Forgotten Layer of the Test Automation Pyramid. Mountain Goat Software – Blog, 17 Dec 2009.
- 8 Harold, Elliotte Rusty: "Encode your XML documents in UTF-8." *IBM Developer*, 30 Aug 2005.
- 9 Hasija, Priyanka: My Experience as a QA in Scrum. *InfoQ*, 17 Jul 2012.
- 10 Hayes, Jay: Small, Distinct Commits Say You Care. *Big Nerd Ranch – Blog*, 25 Sep 2013.
- 11 van Heesch, Dimitri: Doxygen Manual. Documenting the Code. 3 May 2014.
- 12 Hock-Chuan, Chua: Database Programming – An Intermediate MySQL Tutorial. Programming Notes. Nanyang Technological University, Singapore, 29 Oct 2012.
- 13 HTML Purifier: UTF-8 – The Secret of Character Encoding. HTML Purifier Documentation.
- 14 Kleiba: Answers to Windows-1252 to UTF-8 encoding asked by Sam on Stack Overflow, 6 Jan 2010.
- 15 Lockhart, Josh. The new PHP – PHP's experiencing a renaissance, with improvements and new standards. *Ideas*, O'Reilly Media, 4 Mar 2014.
- 16 Naberezny, Mike and Matthew Weier O'Phinney. PHP Developer Best Practices. Presented at ZendCon 2008, 15-18 Sep 2008. The slides in PDF format. *Zend*, 15 Sep 2008.
- 17 Notenboom, Leo. Why do I get odd characters instead of quotes in my documents? *AskLeo!* 13 Sep 2009.
- 18 Pivotal Labs. What's the difference between *Finish* and *Deliver?* *Pivotal Labs Community*, 29 Jun 2011.
- 19 Rutter, Thomas. Best answer to What's the difference between utf8_general_ci and utf8_unicode_ci asked by KahWee Teng on Stack Overflow, 20 Apr 2009.
- 20 Scholtz, Bauke. Best resolution to â€™ showing on page instead of ' asked by Jitendra Vyas on Stack Overflow, 19 Mar 2010.
- 21 Sen, Anith. 5 Simple Database Design Errors You Should Avoid. *simple talk*, 16 Oct 2009.
- 22 Skeet, Jon: Answers to Windows-1252 to UTF-8 encoding asked by user "Sam" on Stack Overflow, 6 Jan 2010.
- 23 Spooner, Chris. 10 HTML Entity Crimes You Really Shouldn't Commit. Line25, 20 Jun 2011.
- 24 Tasker, Ben. Linking a Git repository with Pivotal Tracker. BenTasker.co.uk, 25 Jan 2013.
- 25 Watkins, Warren. Efficient acceptance tests are all that's required. WarrenWatkins.com, 9 Nov 2014.
- 26 Wikipedia. Scrum. (software development)