3
Validating Form Input on the Server Side

This chapter will look at the topic of server-side validation, albeit from a lens that is unorthodox on a couple of counts.

First, part of the established school of thought is that server-side validation should define what is correct and be as strict as possible in enforcing the expectations. What this claim means is that server-side validation should impose assumptions as rigidly as possible. Perhaps there was a day when it was legitimate to assume that all addresses worked like United States addresses and were written in ASCII, not Unicode. In today's world, websites will see users from the whole world, and it is no longer best practice to strive to be strict in enforcing assumptions, such as can be made about United States postal addresses.

Second, in a bygone era where a hard drive was comparable in size to a large laundry appliance and had a storage capacity in the tens of megabytes, it made sense to be as thrifty with individual bytes as possible and truncate data so the databases could have limited column widths.

Now, both obsolete assumptions should be modified in favor of usability and globalization best practices.

In this chapter, we will:

- First, look at what has been the standard security perspective, a perspective still worth understanding even as we depart from it on *some* points.
- Look at what is provided "for free" in terms of security and validation in Django.
- Look at what Django provides as of version 1.2 in terms of hooks that can be used and adapted for specific validation needs.

- Discuss usability best practices as they are mostly followed in Python—but not as well in Django's default way of doing things—and how the defaults can be compensated for.
- Look at GPS coordinates as an example of a legitimate place to exercise server-side validation.
- Discuss validation as an area where less is more.
- Look at why traditional validation is "negotiating with pistols", and what can be done as an alternative—underscoring that Python and Django already take care of several important security considerations.
- Look at a better usabilility/UX alternative requiring every field be filled out before a user may move on.

The standard lecture: low-level validation

Let us look first at the mindset for low-level validation, and look at the use of regular expressions in validation for security.

Matching regular expressions

Though, strictly speaking, validation on the server-side is more than a security measure, client-side validation is not the right place for security validation. Client-side validation is a courtesy to the user that can allow more graceful handling of honest end-user mistakes. Server-side validation should be based on the premise that input is guilty until proven innocent of being malicious, and no assumption may be made that input comes from your client-side software at all. If you have a page or subsite that handles Ajax requests, it should stand up to any malicious request that could be sent to that URL from a standard or *non-standard* client.

For example, you have a JavaScript date picker that sets a form field to an ISO-8601 *yyyy-mm-dd* format, such as "2001-01-01" for when an event is to be scheduled. You want to at least perform a regular expression check or equivalent, such as:

```
is_valid = re.match(r"^\d{4}-\d{2}-\d{2}$")
```

This may be appropriate as a first layer of validation, although by itself it is incomplete. "9999-99-99" matches this regular expression but does not qualify as a valid date. Furthermore, if the date is for an event being scheduled, it should be in the future. "1327-12-25", however valid as a date, cannot be a valid date in the future. On that point there is a moral tale:

A programmer has a problem involving validating strings.
The programmer thinks, "I know! I'll use regular expressions."
Now the programmer has two problems.

Regular expressions may have a place, but they bear liabilities: they are terse and cryptic. So errors will not jump out as easily as in Python-style "executable pseudocode". This is a major problem for a tool used for security, and there are some things that are either very difficult to do or impossible. A regular expression that would accurately test for valid dates would be a completely undecipherable screenful of code. A regular expression that would in addition test for a date that is today or in the future would not only be a completely undecipherable screenful of code—it would have to be a *dynamically generated* completely undecipherable screenful of code!

All this is to say that regular expressions should not be our only tool for validation, even if it is tempting to use them that way.

You cannot guarantee absolutely valid data

One further point is that though we speak of "validation", in general it is not possible to fully validate input, in the sense that we have guaranteed it is correct. If a user has selected a date from a date picker, and accidentally clicked on the option next to the intended date; there is no real way we can protect against such errors. There might be some things we can do to the user interface so such errors are more likely to be noticed by the user, but "validation" can't prove that we have the date the user intended.

Likewise, if we request a user's name, "dfsjkhlasdlhjksdfjhklds" is not appropriate input. Although, it would be a fool's errand to try to make an input validation routine that would accept any real personal name in Unicode but would avoid garbage like "dfsjkhlasdlhjksdfjhklds". The point of validation is not to guarantee that input is valid, but to guarantee that certain foreseeable types of invalid input are not given, and in particular to avoid malicious input.

Validating can detect (some) malicious input

A canonical example of malicious input is SQL injection. If, in PHP, you execute a constructed query of:

```
"INSERT INTO comments_log (comment, timestamp) VALUES (
                         '" . $_POST['comment'] . "', NOW());"
```

And `$_POST['comment']` is something like:

```
"', NOW()); DELETE FROM comments_log; --"
```

Then the executed SQL statements will effectively be:

```
INSERT INTO comments_log (comment, timestamp) VALUES ('', NOW());
DELETE FROM comments_log; --', NOW());
```

With that, the `comments_log` table is effectively wiped out.

I gave the example in PHP, not specifically to pick on PHP, but because there is no particularly straightforward way to accidentally create this vulnerability in Python (or Django). In Python and Django you can't shoot yourself in the foot like this unless you really go out of your way to do so. If you write direct SQL in Python in the preferred, standard way, Python will handle "escaping" of the SQL so that this doesn't happen. And if you do standard things in a standard way in Django, Django will use Python in such a way that this doesn't happen. To minimally correct our PHP example, if the query is constructed as follows, with a backslash added before each single quote, we have:

```
"INSERT INTO comments_log (comment, timestamp) VALUES (
    '" . str_replace("'", "\\'", $_POST['comment']) . "', NOW());"
```

That still isn't 100% correct because backslashes themselves aren't escaped. But, even without correcting that additional bug, we have addressed the vulnerability so that the code behaves in the programmer's intended way when people enter text that includes a single quote. (The genuinely correct solution in PHP is to use provided escaping functions: `pg_escape_string()`, `mysql_escape_string()`, `mssql_real_escape_string()`, and similar functions.)

The Django way of validation

In Django, many of the low-level requirements are irrelevant. This is not because they need to be done, but because the framework takes care of certain things for you. Django assumes that the input is guilty of being potentially malformed, until proven innocent.

Django gives you some things for free

In Django, much validation is already taken care of for you. Django offers a set of fields including: (`forms.`)`BooleanField`, `CharField`, `ChoiceField`, `TypedChoiceField`, `DateField`, `DateTimeField`, `DecimalField`, `EmailField`, `FileField`, `FilePathField`, `FloatField`, `ImageField`, `IntegerField`, `IPAddressField`, `MultipleChoiceField`, `NullBooleanField`, `RegexField`, `SlugField`, `TimeField`, and `URLField`.

These fields are stored in the database using the database's built-in types: an `EmailField` or a `URLField` ends up being stored by default as a `VARCHAR`. However, Django provides validation that they are correct as e-mail addresses or URLs, and (though this can be optionally turned off) Django will check that a URL exists and does not give a 404 on attempted access. The validation that comes along more or less "for free" in Django is quite a lot; malformed input like `GET /?%?%?%?%%%%&&&& HTTP/1.1` will be rejected as malformed before your validation has to deal with it. If input is passed on, it has probably passed a sanity check. `FloatField` and `IntegerField`, for instance, will be coerced to floats and integers with an exception raised if the text of the CGI input value cannot be so coerced.

There are several steps in validation; the different steps serve as hooks that can be customized, often (but not always) by overriding the appropriate method. Note that in some cases the best practice is to add specific validation to an existing field type rather than create a new field type. If we wish to confirm that a product key passes our cryptographic check for a valid product key, we could make a `CharField` of appropriate size and set its validator to throw a `ValidationError` if it did not pass our check.

The steps in Django's validation

The specific steps of validation as of Django 1.2, in rough order, are:

1. `to_python()` returns an appropriate Python datatype, such as `int` for `IntegerField`, or else throws a `ValidationError`. Here and elsewhere, any `ValidationError` should be constructed with a string explaining how the form input failed validation.
2. `validate()`, for the field, handles any validation that shouldn't be handled by validators (which are functions that can be specified in a field's construction). It takes a value in the correct datatype and raises a `ValidationError` on error. It should not alter the value.
3. `run_validators()` runs all of a field's validators, and aggregates any `ValidationErrors` (if there are more than one) into a single `ValidationError`.
   It should not usually be necessary to override the `run_validators()` method itself; validators to feed into `run_validators()` can be specified on field construction.
4. The field's `clean()` method runs `to_python()`, `validate()`, and `run_validators()`, and propagates any errors generated. The cleaned version is returned, and used to populate the form's `cleaned_data` dictionary.

5. Any form subclass's `clean_<fieldname>()` method can be overridden, will need to look up data from the form's `cleaned_data` dictionary, and will return the cleaned data (whether changed or unchanged). If we subclassed form and wanted to confirm that an included serial number was legitimate, a `clean_serialnumber()` method would be appropriate and should raise a `ValidationError` if the serial number is not legitimate.
6. Any form subclass's `clean()` method. This hook in particular is intended for requirements that do not strictly belong to individual fields. For instance, if you require that someone filling out a help request provide either phone or e-mail contact information, then neither the phone nor the e-mail field individually is required. But, a form subclass's `clean()` method is a natural place to check and raise an error if the user has not provided at least one. Alternatively, a feedback form might be explicitly intended for users who wish to leave feedback but may or may not wish to be contacted. In that case, it may make sense to have a checkbox for whether the user wishes a response, and require contact information only if the user desires a response. If a user is presented a form to specify options for an order and some combinations do not make sense, the form subclass `clean()` method can also be used to check that the user has not selected a combination of options that cannot be fulfilled.

A more sensible and cruelty-free approach to validation

To give an example of use, let us initially develop a form to collect a telephone number from end users. One basic usability concern is that while it may be entirely appropriate to store a telephone number in the database as a string of digits, it is both lazy and rude to users to impose "validation" that says "If you type any non-digit character at all, I'm going to reject it as wrong." The same principle is true with credit cards, although if we are writing code to handle credit cards in Django and we are not (for instance) the developer of a major ecommerce app, we are probably reinventing the wheel and therefore creating trouble for ourselves, when we could reuse existing debugged software. For credit cards, it is rude to demand that people type in credit cards as "4111111111111111" because you choose not to show the courtesy to write software that will overlook non-numeric characters and accept credit card numbers entered like "4111 1111 1111 1111" or "4111-1111-1111-1111".

Returning to telephone numbers, for very good reason people usually do not write, for example, U.S. telephone numbers like "8005551212". There is some variety in the wild between formats like "(800) 555-1212", "800-555-1212", and "800.555.1212"; but all of them break the numbers up because giving people 10 digits in a row is unnecessarily cruel. Furthermore, standardizing a credit card number to a string of digits desired for database storage is an introductory-level exercise:

```
from django.db import models

class CreditCardNumber(models.TextField):
    def clean(self):
        return re.sub(r'\D', '', str(self))
```

Or, if we wish to be able to store "(800) 555-1212 x123" as "8005551212x123" and perhaps vanity numbers like "(888) 4-VANITY" as "8884VANITY," then:

```
from django.db import models

class PhoneNumber(models.TextField):
    def clean(self):
        return re.sub(r'\W', '', str(self))
```

And perhaps add `.lower()` or `.upper()` to standardize case. (Alternatively, it may make sense to only allow a whitelist of letters, perhaps just 'x'. Where a digit means a number dialled and an 'x' only means "This is not for dialling; wait for the phone to be picked up and then go on dialling digits," or perhaps an algorithm could store upper-case letters for vanity values and a lowercase 'x' for extensions.)

Things get murkier

So what would this look like in Django? We would subclass form and add a validator, but there's another issue that comes up. What's the maximum length of a phone number that we need to support? Even if we restrict our attention to U.S. telephone numbers, the previous examples show a 10 character limit to be shortsighted, and sometimes a bit of a straitjacket. It is a relatively common occurrence that the number a person leaves is 10 digits, plus an extension of 3 to perhaps 5 digits. Less commonly, conference calls can have more like 7-10 digits conference PINs.

In the U.S. alone, possible phone numbers can be a bit open-ended in length, and writing code with only the U.S. in mind is breakage by design. How many digits can phone numbers be worldwide?

The zero-one-infinity rule: a cardinal rule of thumb in usability

This is a good time to make a point that the Django documentation at `http://docs.djangoproject.com/en/1.2/` or `http://djangobook.com/` never make clearly: there is a rule of thumb in interface design of "Don't allow _____ at all, or allow up to one of ______ but not more, or allow as many of _____ as available resources support." This principle is called the **zero-one-infinity rule**, and while zero and one both have a place, two or twenty but not more are breakage by design.

In traditional SQL, breaking the zero-one-infinity rule is almost required. Normally in declaring a column, you declare where it will break the zero-one-infinity rule. You may try to add a healthy bit of slack in *how* you do this, but you ordinarily create a column for people's names as `VARCHAR(20)`, or `VARCHAR(100)`, or whatever, but whatever the limit you choose, you specify an arbitrary line and guarantee that if a user enters *one* character more, let alone twenty, you will drop it on the floor.

An improvement on Django's advertised approach

Django is Pythonic in many ways, and Python observes the zero-one-infinity rule well:

- Strings can be of any length that available resources will support
- Lists and tuples may likewise be as large as can be supported
- Partly through deft handling of integer overflow, you can have correct integer calculations with integers of three digits or three thousand

However, Django follows SQL and not Python to present "breakage by design" as the normal way of handling most situations: fields such as `CharField`, `EmailField`, and the like are built on SQL's `VARCHAR` and presented in the documentation with the request that you specify an arbitrary length for `CharField`, `EmailField`, and the like so that if a user provides just one character more, the data will be truncated.

Fields like `EmailField` can be subclassed to use `TEXT` instead of `VARCHAR`, so that there is no arbitrary threshold beyond which data will be corrupted. For instance:

```
from django.db import models

class TextEmailField(models.EmailField):
    def get_internal_type(self):
        return 'TextField'
```

To return to telephone numbers, yet another solution, and perhaps a better one, would be to preserve the formatting characters so that the formatting provided by the user is still available, save it as `TEXT`, and declare that two phone numbers are equal if they contain the same digits:

```
from django.db import models

class TextPhoneField(models.TextField):
    def __eq__(self, other):
        try:
            return self.remove_formatting() == other.remove_
formatting()
        except:
            return False
    def remove_formatting(self):
        return re.sub(ur'\D', u'', str(self))
```

This overloads `==`, and respects Python's duck typing by avoiding `isinstance`: if someone else creates a class that does the work of a `TextPhoneField` but is not descended from it, we don't want to declare all of its instances unequal to any `TextPhoneField`, because it is not an instance or a subclass. The way we remove formatting has been offloaded to its own method, both to make it available and to allow changes (for instance, if we decide to change what characters we do and do not remove, perhaps by replacing the regular expression `ur'\D'` with `ur'\W'`).

Let us look at one more detail of Django validation before building a validated form for our case study. Django fields can be created with the argument `required = True` or `required = False`, defaulting to `required = True`. For instance, a `CharField` or `TextField` created the default way will register a validation failure on an empty value. There are at least two different cases where you will want to set `required = False`: if you ever might want to allow an empty value (it is usually not a user interface best practice to make people fill out all fields, all the time); or if you want to have a useful `BooleanField`, normally rendered as a checkbox (the `BooleanField`, like other fields, defaults to being required that you fill it in, meaning that it defaults to being rejected as invalid if you do not click a checkbox—the meaning of the checkbox is not "Do you mean 'Yes' or 'No'?" but "Say 'Yes', and give your rubber stamp here.").

A validation example: GPS coordinates

Let's take a slightly more involved example of input validation: a field for GPS coordinates. This is complicated in part because "GPS coordinate" does not *exactly* mean a single system, but one of a few different systems. For this specific system, which is something of an exception, we *could* use the `VARCHAR` based `CharField`. This is because like the information stored in a `Date` or `DateTime`, there actually is an upper limit to how long a useful value will be. A measurement to the nearest thousandth of a second is specific to within a distance you could measure with a handheld ruler. We will allow some added space, but the goal is not specifically to create a GPS coordinate field that allows subatomic precision.

Among the ways an address could be specified are the following:

```
36° 09' 55.8"N, 86° 46' 58.8"W
36.16586 -86.78425
37° 25.330'N, 122° 5.039'W

N36 09 55.781
W86 46 58.287
```

We may find it helpful to use regular expressions at some point. While it should, in principle, be recommended to make a single regular expression call to handle all these cases, much better would be to break it down into a series of manageable tests, in this case returning success if input matches a test as valid, and then raising a `ValidationError` if none of the tests could interpret it as valid. We will specify a maximum length, but for now leave it unspecified, something to be determined after we know how long the longest validating entry will be.

Our code is as follows:

```
from django.db import models
from django.core.exceptions import ValidationError
import re

def gps_validator(value):
    # Create a normalized working copy of the value.
    working_copy = value
    working_copy = working_copy.replace(u'\n', u',')
    working_copy = working_copy.replace(u'\r', u',')
    working_copy = re.sub(ur',*$', '', working_copy)
    working_copy = re.sub(ur',+', u',', working_copy)
    if not u',' in working_copy and not \
        re.match(ur'.* .* .*', working_copy):
            working_copy = working_copy.replace(u' ', u',')
    working_copy = re.sub(u'[\00B0\2018\2019\201C\201D\'"]', ' ',
                          working_copy)
    working_copy = working_copy.replace(u',', u', ')
    working_copy = re.sub(ur'\s+', u' ', working_copy)
    working_copy = working_copy.strip()
    working_copy = working_copy.upper()
    # Test the normalized working copy against regular
    # expressions for different kinds of GPS format.
    if re.match(ur'[-NS]? ?\d{1,3} [0-5]\d [0-5]\d(\.\d+)[NS]?,
                [-EW]? ?\d{1,3} [0-5]\d [0-5]\d(\.\d+)[EW]?',
                working_copy):
        return working_copy
    elif re.match(ur'[-NS]? ?\d{1,3} [0-5]\d(\.\d+)[NS]?,
                  [-EW]? ?\d{1,3} [0-5]\d(\.\d+)[EW]?',
                  working_copy):
        return working_copy
    elif re.match(ur'[-NS]? ?\d{1,3}(\.\d+)[NS]?,
                  [-EW]? ?\d{1,3}(\.\d+)[EW]?',
                  working_copy):
        return working_copy
    else:
        raise ValidationError(u'We could not recognize this as a valid
GPS coordinate.')

class GPSField(models.TextField):
    default_error_messages = {
        u'invalid': u'We could not recognize this as a valid GPS
coordinate.',
      }
    default_validators = [gps_validator]
```

The regular expressions are cryptic and ugly, but, in a nutshell, what is done is to coerce the data to a normalized form, and then test for whether the normalized form of the input looks like "N36 09 55.781, W86 46 258.287" or other comparable forms.

Avoiding error messages that point fingers and say, "You're wrong!"

We might also briefly stop to note a principle of user experience and user interface. The computer must at times bear bad news, but there is a difference in how the software comes across between saying, "We could not recognize this as valid input" and "You messed up."

Your software will be better liked and better received if it handles validation failures by saying "**We** could not recognize this as valid input", "**We** need additional information before we can continue", and so on than if it says, "**You** entered data that was invalid", or "**You** failed to provide required information". If nothing else, there is much less egg on the programmer's face if the senior vice-president is an avid geocaching hobbyist who knows GPS inside and out, and enters GPS data in a perfectly valid format that we didn't know about. It is much better to tell a hobbyist vice-president who knows GPS better than we do, "We couldn't recognize this as a valid format." than, effectively, *You idiot, we know better than you that you don't know how to write a valid GPS location.*

Now, after the fact, if we allow for specification of up to a thousandth of a second, a "maximally long believable cleaned input" might be:

```
N100 10 10.111, S100 10 10.111
```

That creates a Unicode string of length 30. We might suggest that a good programmer might be slightly nervous about fixing a length of 30, probably because if someone adds extra whitespace, or in light of unforeseen future developments, it makes sense to use GPS coordinates to sub-millimeter precision and so people start to enter data like:

```
N 100°, 10', 10.11111",
S 100°, 10', 10.11111"
```

Our regular expressions may be slightly more future-proof because they are written flexibly. But even here, there is reason to be a bit nervous about choosing `VARCHAR` over `TEXT`.

Validation as demanding that assumptions be met

To validate is to demand that user data conform to your expectations, meaning *assumptions*, about what constitutes valid data. It's hard to make assumptions that are valid the world around.

Old-school: conform to our U.S.-based assumptions!

Let's take a somewhat standard, old-school approach to validation of what goes in the database. This approach says that you don't want junk data in your database and you define strict rules and take every possible step to prevent junk data entering. The initial, U.S.-centric form has:

- Street address, line 1 (*required*)
- Street address, line 2 (*optional*)
- City (*required*)
- State (*required, and specified by a dropdown menu*)
- Zip code (*required*)

This is U.S.-centric but not entirely fair to U.S. residents themselves. For starters, some forms like this out in the wild reject valid input. There is a five-digit and a nine-digit version of a ZIP code. Although the diligent nine-digit zip code is preferred and is intended to allow mail to reach its destination faster, enter a nine-digit zip code on many forms like this and you will get a response amounting to "This is unacceptable; please try again and give a real zip code this time."

But even if that problem is not present, there is another problem as far as the state goes. If we just let people type in a two-letter state code, we are inviting typos, even if many of them could be caught by straightforward server-side checks. But if we require a drop-down menu so they *can't* type an invalid state, something really funny happens even if every single record has a real state. Valued customers who may not have spent long hours on Pac-Man (let alone growing up on smartphones and SecondLife) try their best to handle the long dropdown menu but still leave addresses like:

Ashley Jones

1745 Broadway

New York, **NC** 10019

Whereas we could tell there was a problem if this person accidentally made a typo corresponding to an invalid state, here the error is that a different valid state has been (mis-)registered as the customer's state. And what is worse, we get many more of these invalid records than if we just let customers type in their own states. Not only is the long dropdown menu cruel to users, it makes a good many users have more trouble entering the data they intended. As a result, the dropdown menu delivers more than a painful user experience. It delivers significantly and substantially more corrupt data in the database than if we had just been "lazy" and let them type a two-letter state code. And it delivers corrupt data that is harder to identify than typos like "NU" which are usually not valid state codes and are therefore often more straightforward to identify as corrupt data.

What do we do if we want to take a form like this and make it more international? One solution that perhaps all of us have seen is:

- Address, line 1 (*required*)
- Address, line 2 (*optional*)
- City (*required*)
- State, province, district, region, or territory (*required*)
- Zip or postal code, if applicable (*optional*)
- Country (*required*)

And we can particularly offend Canadians, who wince at (real or imagined) evidence that the U.S. lumps them in as the 51st state, by making a "state/province/…" menu that intermingles U.S. states and Canadian provinces and then leaves the rest of the world's regions as "Other (please specify below:)". This solution is probably meant as a gesture of warmth from the U.S. to Canada, but it is a gesture of warmth that quite probably did not include consulting a Canadian about Canadian sensibilities.

There is at least one U.S.-specific concession that actually makes a lot of sense. For the New Yorker who didn't grow up on video games, putting all of the countries in alphabetical order may end up with a problematic number of street addresses like:

> Ashley Jones
>
> 1745 Broadway
>
> New York, NC 10019
>
> United Arab Emirates

If we wanted to trust the user's judgment a little more, we could let people fill out their country as text. That could result in less-uniform data, as "USA", "US", "U.S.A.", and "U.S." would probably all appear in any number of U.S. residents. But it would both be merciful to the end user and refrain from the cruelty of making people pick a needle out of the haystack of a long dropdown menu, *and* result in better and more correct information in our database.

Adding the wrong kind of band-aid

The solution that is taken, in many cases, is to make "United States of America" the first option of the menu so as to avoid giving U.S. residents the needle-in-a-haystack cruelty of finding their country buried under the "U" section of an alphabetical list of every country in the world. And it is needlessly cruel to residents of the United States of America… or the United Kingdom… or United Arab Emirates… or…

This leaves out the internationalization issue that not all addresses are constructed like U.S. addresses. If you take the original structure of:

- Street address, line 1 (*required*)
- Street address, line 2 (*optional*)
- City (*required*)
- State (*required, and specified by a dropdown menu*)
- Zip code (*required*)

Appending country and making the "State" and "Zip" slots generic does not make the structure flexible enough to accommodate internationalized address handling. Some countries have more lines in their street addresses, while some have fewer. Part of this is absorbed by the optional second street address line, but there are still addresses worldwide that are mangled if you demand that they fit this format. Even if you will allow letters in a postal code and freeform text for the state-like field.

Then what can be done? It should be almost theoretically possible to just present a dropdown menu first, specifying country, and then have Ajax adjust the form so that there is an appropriate form for the country. On the backend, an equally Rube Goldberg system of rules could enforce this mentality for every locality worldwide, and on the backend at least, people wouldn't get frustrated waiting for the enormous Ajax application to load and possibly crash their browsers. But at least, in principle, this mentality can be internationalized to put its heavy hand on every international address in fully localized form.

But there is an alternative—*A single textarea.*

Perhaps some programmers might feel like they're not doing their jobs if they "just" give a textarea for address without further doing diligent work, like making people suffer through long dropdown menus, to convince themselves they're preventing people from getting bad data into the system. Even if it means that for every one case where they prevent an address like "…New York, NU…" from entering the system, ten addresses like "…New York, NC…" get through validation. But really, honestly, it's better to let go of the control freak mentality, stop making people jump between fields on your particular form, and just type an address they know well in an ordinary, clearly legible, and attractively styled textarea. *Really, less is more.*

And one additional point to underscore that less is more: Even in the U.S., I've run aground with "more is more" computers that negotiated with pistols. One "negotiated with pistols" demand is that if you have a valid address, you include a street address with a house number. And my address was:

> Jonathan Hayward
>
> Department of Theology
>
> Fordham University
>
> Bronx, NY 10458-9993

And this was treated as not an acceptable address, **period**. Never mind that the ZIP+9 alone told which department in the university to send correspondence to. I did not have the expected street address beginning with a number, and therefore, by definition it seemed, I could not have entered a valid address and I could not be allowed to move forward. And this occasional loss of U.S. customers is a way to drive away customers the world around. People in other parts of the world may wish to pat me on the head and say, "Poor baby! You had a taste, once, of what happens to us all the time!"

Making assumptions and demanding that users conform

Let's look at one other type of basic information: a person's name. For this discussion, we will ignore Unicode issues completely and look at problems that come up even if we stick to ASCII. LinkedIn and Facebook alike request a first and last name, but this isn't quite right.

At least names are simple, right?

On the surface, one obvious validation, if we do not coerce case, is that a person will have a first and last name, each of which is an uppercase letter followed by one or more lowercase letters. That's easy enough to test for, but it doesn't allow for a lot more than a first initial, optionally followed by a period. And strictly speaking, it doesn't allow even most American full legal names at birth. Though "Firstname Lastname" is often treated as a person's full name, most full legal names are "Firstname Middlename Lastname." And most people will treat the common exception of the CamelCased last name, such as the Scottish "MacDonald" or the Irish "McDonald".

But here we have the Monty Python sketch where the Cardinals announce "Our two chief weapons are fear and surprise," because there's at least one more common exception. The French "de Balzac"/"de BALZAC", the German "von Friar", or the Dutch "van Driel" or "van den Driel", which in European convention are alphabetized under the capitalized name because "de"/"von"/"van…" is not strictly part of the name. So our two chief exceptions are Celtic CamelCase, and European words translating to "of" (namely "de"/"von"/"van…") that are added before the proper part of the name. And also the case where Americans of Dutch ancestry have CamelCased or otherwise altered the European convention, making a name of VanDriel or Vandriel.

Even in ASCII, things keep getting murkier

So now our three chief exceptions to our general-purpose validation rule are as above, and also that it is becoming more common practice for women to list both their maiden and married names as last names.

Our four chief exceptions to the general-purpose validation rule include Celtic CamelCasing, variations on European "de"/"von"/"van…", women listing two last names, and Orthodox monks and nuns. An Orthodox woman named Sarah Smith who becomes a nun will take a new first name such as Xenia and give up her last name, will be properly addressed as "Mo. Xenia" (that is "Mother Xenia"), and not be referred to with her last name at all. When it is necessary from context to perhaps specify which Mo. Xenia is being referred to, parentheses are expected around her last name. So she would then for clarification be referred to by name as "Xenia (Smith)". So our validation, if it allows for this, will need to allow for no last name, or a last name that begins and ends with parentheses, or both. Orthodox bishops are monks and thus have only one name, but it is written in ALL CAPS, for example "His Grace BASIL".

Our simple validation rule is imposing certain assumptions, and the list of exceptions seems to get longer each time we list it.

Furthermore, the pattern of a first and last name, where the first is a personal name and the last is a family name, is far from universal. In Chinese culture, people have a first and last name, but the first name is the family name and the last is a personal name.

We would propose the following, which almost assuredly imposes some cultural assumptions that do not hold in all cultures:

- An optional Unicode slot for any honorific(s)
- A Unicode slot for a person's name

- An optional Unicode slot for what in Western culture are letters after a person's name, including not only academic degrees but religious order membership, titles, and an open-ended list of other things that should not exclude being a LinkedIn Open Networker

In U.S. culture, there is an informal society where it is considered unpretentious to simply be addressed by your first name and no honorific. This is not a pattern worldwide and giving people a place to indicate what honorifics are proper to address them by is appropriate. Or, as an alternative which may be pursued for the greater good of the UI, we may have a single slot that holds honorifics, name, and post-nominals.

Now there is nothing wrong with creating heuristics that will try to extract a first and last name, and Django server-side heuristics could be made that would be quite accurate in handling U.S.-style names.

Better validation may be less validation

As far as server-side validation goes, *really, less is more*. Perhaps we as programmers are always looking for ways we can be more diligent, and we look for ways we can work harder that will deliver a better solution. But there's something we need to understand about validation.

Each requirement in validation is a demand that data meet an assumption. Each assumption is a way to make a solution more culture-specific and harder to internationalize, and also less future-proof.

Validation that requires a real five-digit U.S. zip code creates an unpleasant surprise to the customer who diligently gave a perfectly valid nine-digit form in the hope that a product might arrive one day sooner. Old U.S. validation requirements, from when a database really would only hold U.S. information, are almost invariably a legacy obstacle to gracefully handling internationalized and localized information in today's world. There may as well not be a single database validation measure for data like addresses that originally only took U.S. data into consideration that does not create obstacles to handling internationalized data gracefully. Almost by definition, they demand that data meet assumptions that things behave the way they do in the U.S. What if our GPS validator is used in French localization? In English, a period is used to specify a decimal; π to two decimal places would be written, "3.14". In French numbers, the comma does this job, as in "3,14". Our GPS validation routine above, in English localization, would validate "36.16586 -86.78425" but reject a French-style "36,16586 -86,78425".

Caveat: English is something of a lingua franca

Surely our GPS validator could be adjusted to tolerate either locale's decimal marker, but this is the road to Rube Goldberg nightmare code. It might make sense to ask people entering GPS data to use English localization as *lingua franca* for that part of the code. And not only because a GPS field is something we might want to parse and have the computer understand, and it would be a Rube Goldberg code nightmare to write a parser for every possible localized form of GPS coordinates. To native English speakers concerned about cultural chauvinism, we might suggest that it makes a lot more sense, internationally, than we might suspect.

When I was studying in Paris, I remember seeing a keyboard synthesizer, looking to see what was selected, and being utterly shocked to see "HONKY-TONK PIANO," *in English*. I had assumed that in the U.S. synthesizers were labeled in English, and in France synthesizers would just as naturally be labeled in French. But the markings, instrument names, and so on were pure English and I was the only one to find this strange. English is *lingua franca*, or close to it, for technology, and speakers of some other languages find it normal to have an online forum with the user interface in English (and forum threads in their native language).

Latin remained the language of scholarship and international discourse in Europe whether or not there were any native Latin speakers at all. When Europe moved to writing scholarship in people's native languages, book titles remained in Latin for much longer. The 20th century German Ludwig Wittgenstein's seminal *Tractatus Logico-Philosophicus* is written in German, but the title *is in Latin*. And when the work is dealt with in English, the contents are translated to English. But the title, which could be translated in English like "Logical-Philosophical Treatment", is abbreviated to *Tractatus* but seems to *only* be given in Latin even among English speakers who have never studied Latin.

English now is a bit like Latin: it is something of a standard language, especially in matters of technology. And many non-native speakers make it their first choice in discussing technology. Furthermore, it might be pointed out that Google has made extraordinary efforts at localization, with meticulous attention to detail for giving directions in Japan for instance. Google Maps at the time of this writing accepts "36.16586 -86.78425" and pulls it up immediately, but for a location of "36,16586 -86,78425", answers, "We could not understand the location 36,16586 -86,78425".

> We might point out that the error message, "We could not understand…" is well done and minimizes any message of "You don't know what you're doing."

Django comes with validated, *and worldwide*, `EmailField` and `URLField` fields. And it was designed by journalist professionals who would know perfectly well that it is desirable to be able to store phone numbers and postal addresses. However, there is no `PhoneField` or `StreetAddressField` because making worldwide validation for these is a tarpit. There are tools to make a U.S.-centric address field, complete with long, cruel dropdown menus to specify a valid state, but no `WorldStreetAddressField` because that would be a Rube Goldberg tarpit.

Then is there any validation to be done, or do we simply drop it? We might propose one alternative.

We don't have to negotiate with pistols

The standard rule of practice is for validation to negotiate with pistols. So if data doesn't meet your assumptions, you say "I will not let you move forward until you change this data to what I assume it should be." As far as Django server-side validation goes, if you specify that an input fails validation, you guarantee that the input as submitted could never be entered into the database. But there is another client-side option.

For the client side, possibly the most common error is not specifying a complete address including country. And it would be both possible and sensible to run a client-side test to see if you can recognize the provided address as including a country. If you cater to a mainly U.S. clientele, it is also entirely permissible to have the system infer a U.S. address if the address ends in something like "New York, NY 10019-4343".

But you don't have to negotiate with pistols. Instead of refusing to continue unless the user enters an address that meets our assumptions, we could give a dialog that says, "We could not identify what country this address belongs to" with buttons of **Go back and specify a country** and **This address is correct**. This provides a way to identify foreseeable data errors, and asks the user to correct them. But at the same time, it avoids creating an absolute roadblock to the user who specifies an entirely appropriate address that our validation routines didn't foresee.

For a U.S. company keeping a directory of telephone numbers, with, say, four-digit phone extensions. A phone number might be tested, client-side, by stripping out non-word characters, and accepting a phone number of at least ten digits as valid; or a phone number of ten digits, an 'x', and at least one more digit as valid; or exactly four digits optionally preceded by an 'x' as valid; or warn for any other number. That offers significantly better flexibility for people coming from outside the U.S., even if it does not undertake the (again) Rube Goldberg task of checking for country codes and numbers that correspond to the given country code. But if the validation warns, instead of demanding that its assumptions be met, someone who tries to make an entry of "466453" (for texting queries to Google) will be allowed, after a warning, to enter a number not foreseen in the validator's assumptions.

Doing our best to solve the wrong problem: a story

Perhaps the most inappropriate textbook example of why we need precise requirements tells of a professor who assigned as homework that students should program a four-function calculator. One student turned in a flawless four-function calculator using *Roman numerals*, complete with a full user's manual in Latin. The obvious point that was presented in the example is that the professor would have been better to be more precise in the homework specification, but this is naïve and stupid. The student was giving a crystal-clear signal, "I'm in the wrong class!" It is stupidity worthy of Dilbert's boss to try to solve this problem with more specific requirements. Had the professor been more specific, stating that Arabic numerals were acceptable and Roman numerals were not, this would have accomplished two things:

- First, it would have made the assignment more confusing for most of this student's classmates
- Second, it would have thrown down the gauntlet to any student who was in the wrong class

The student, perhaps, would have produced a fully functional four-function calculator, implementing the specification to the legalistic letter, and the professor, perhaps, would have been surprised to type:

```
> 7*8
56
```

And then find, ostensibly to provide enhanced accessibility, that the program played a sound clip of a painfully loud note on an out-of-tune piano fifty-six times before moving on. And the student had taken thought to set a Unix terminal mode that included disabling the usual ways to send a suspend or interrupt from the keyboard.

The problem when a student hands in a four-function calculator—with Roman numerals and a full Latin user's manual—is that the student is bored silly in the wrong class. The correct solution is to bump the student up some classes until the student's ability and the class's challenge level are in sync. Kneejerk reactions to move towards more legalistic requirements do not help. Part of the Agile experience is that handling requirements more legalistically does not help in the real world, either, not as much as one might think. (The documents and communication of Agile delivering its best solutions probably never have the legalistic precision of a heavyweight waterfall requirements document solving a badly identified problem as precisely as possible.) Communicating clearly, of course, is always an asset, but trying to manage risk and change by trying to be legalistic and pin things down that way is an attempt to solve the wrong problem, which rarely does a good job of addressing the correct real-world problem.

It really does apply to validation

And the same insight holds for the "Big Design Up Front," heavyweight design process mentality for validation. If we deal only with U.S. contacts, we can use a form with:

- Address, line 1 (*required*)
- Address, line 2 (*optional*)
- City (*required*)
- State (*required*)
- Zip (*required*, any valid zip code accepted)

If we are afraid that people will give junk values, with a city like "afdjkfdkj", we may be able to use USPS web services or the website to test whether an actual street address is given. But then what if people use an address across the country? Conceivably, we could locate their IP's geographical location in relation to the address. But this is being silly, not to mention that customers might find it a tad creepy. People who really want to provide junk data, perhaps to preserve the anonymity of their addresses when it is inappropriately demanded, will do so. We can try to do things that will catch common data entry errors; but trying to legalistically keep invalid data out of our databases will accomplish little more than a more legalistic wording for the four-function calculator assignment.

But don't we need to assume data is guilty until proven innocent of being malicious and possibly malformed? Not personally if we are using Django, because Django is intended to take care of those basics for us. If we can find a way to straightforwardly build a model with Django-provided fields, with Django managing the SQL, and still succeed in executing arbitrary SQL by an injection attack, we should contact `security@djangoproject.com` about how we managed to straightforwardly use Django models and fields and create such a vulnerability.

For that matter, if you manage to find a strange and convoluted setup that exposes any vulnerability that you didn't deliberately create, please notify `security@djangoproject.com`. They would like to know even if it turns out, on later inspection, to be spurious. They have tried very hard to make "naïve" use of their models and fields as safe as possible. And while no programmer worth trusting would say "We know that our program is free from vulnerabilities," they have tried very hard, and undergone intense public scrutiny, so that Django can take care of being paranoid about attacks like malformed input and SQL injection for us. At least as far as security basics are concerned, Django is designed to take care of assuming that data is guilty until proven innocent of being malicious and possibly malformed.

Facebook and LinkedIn know something better

We would last make an observation from social networks. Social networks, like Facebook and LinkedIn, thrive on full profiles but demand surprisingly little information to be let in, at least to start off with. The initially required information asks little beyond the core essentials without which the network cannot handle the user. But then they use another dynamic: while demanding lots of information up front is off-putting and would kill them, people like to have complete profiles. And both social networks provide feedback that a particular account is 25% filled in, or 80% filled in, and suggest a concrete and manageable step to take to improve on that percentage.

We will be developing as our model application, an intranet employee photo directory. While not intended to compete with social networks, we will take a cue and require to start with a minimal amount of information and then let people move towards a 100% complete profile, one step at a time.

Summary

We've looked at server-side validation with an emphasis on usability and internationalization concerns, and raised the idea that a validation requirement, especially on the server side, is a demand that user input conform to your assumptions. That is exactly how U.S.-centric assumptions in data validation can leave non-U.S. visitors with no way to move forward giving their address in a form that will pass validation.

Django does a lot for us, including a lot of work to gracefully handle Unicode. We might be advised to use `TextField` instead of `CharField`, and `TEXT` in preference to `VARCHAR` due to a basic usability concern, but Django already does a lot for us, including quite a lot of holding data innocent until proven guilty of being malicious and possibly malformed.

In this chapter we have covered the insights traditionally considered essential to optimal server-side validation. Then we moved on to how Django addresses certain basic security concerns and provides a framework for validation. We introduced one usability principle and best practice—the "zero-one-infinity rule"—and discussed how Django can be adapted to observe it. We also provided an example of how to make a validated field for GPS coordinates, discussed when and where "less is more" with validation, and how this relates to usability and internationalization. We then discussed usability best practices to encourage profile completion while not demanding that users fill in every field before being allowed to continue; this we will see in the next chapter.

Next, let's look at searching on the server side with Django, for which it provides excellent tools.