Tangelo Web Framework Documentation

Release None

Kitware, Inc.

January 21, 2016

Contents

1 Getting Started **3**
- 1.1 Quick Start . . . 3
- 1.2 Hello World . . . 3
- 1.3 Diving Deeper . . . 3

2 Using Tangelo **5**
- 2.1 Installation . . . 5
- 2.2 Setup and Administration . . . 7
- 2.3 Basic Usage . . . 10
- 2.4 Tangelo Web Services . . . 11
- 2.5 Tangelo Plugins . . . 16

3 Tutorials **21**
- 3.1 Building a Tangelo Web Application from Scratch . . . 21
- 3.2 Driving a Visualization with SQLAlchemy . . . 23
- 3.3 Fiddling with the Bundled Examples . . . 32

4 Command Line Utilities **35**
- 4.1 `tangelo` . . . 35
- 4.2 `tangelo-passwd` . . . 36

5 The Tangelo API **37**
- 5.1 Python Web Service API . . . 37
- 5.2 Tangelo JavaScript Library . . . 40
- 5.3 Bundled Plugins . . . 42

6 Information for Developers **65**
- 6.1 Coding Style Guidelines . . . 65
- 6.2 Creating Tangelo Releases . . . 69
- 6.3 Developing Visualizations . . . 71

7 Indices and tables **73**

Tangelo is a web application driver, implemented as a special-purpose webserver built on top of CherryPy. Tangelo paves the way for you to use HTML5, CSS, JavaScript, and other web technologies such as jQuery, D3, Bootstrap, WebGL, Canvas, and Vega to create rich web applications - from traditional, static pages, to cutting-edge, visual, dynamic displays.

But **Tangelo also considers Python scripts to be part of your application, alongside your HTML and JavaScript files**, running them on your behalf to do anything from retrieving a few database results for display, to engaging with powerful computational engines such as Hadoop to compute complex results. In other words, **Tangelo reimagines the web application by making Python a first-class citizen**, effortlessly integrating your Python code into your web applications without your having to adapt to complex web frameworks, worry about routing, or engage in the busy work of putting up application scaffolding.

To help in creating these newly enriched web applications, Tangelo exports the Tangelo API, a collection of Python and JavaScript functions, standard plugins, and the means to create custom plugins of your own, to create your own Tangelo applications.

This document describes all the pieces that fit together to make Tangelo work.

Please visit the Tangelo homepage and GitHub repository, and read our ongoing blog series for more information.

CHAPTER 1

Getting Started

1.1 Quick Start

1. Make sure you have Python 2.7 and Pip installed (on Linux and OS X systems, your local package manager should do the trick; for Windows, see here).
2. Open a shell (e.g. Terminal on OS X; Bash on Linux; or Command Prompt on Windows) and issue this command to install the Tangelo package:

```
pip install tangelo
```

   (On UNIX systems you may need to do this as root, or with `sudo`.)

3. Issue this command to start Tangelo, serving the example pack:

```
tangelo --examples
```

4. Visit your Tangelo instance at http://localhost:8080.

1.2 Hello World

Follow these steps to create an extremely simple Tangelo application:

```
$ mkdir hello
$ cd hello
$ vim helloworld.py
```

```
import datetime

def run():
    return "hello, world - the current time and date is: %s\n" % (datetime.datetime.now())
```

```
$ tangelo --port 8080
$ curl http://localhost:8080/helloworld

hello, world - the current time and date is: 2015-03-31 14:29:44.29411
```

1.3 Diving Deeper

- To tinker with the Tangelo examples directly, see *Fiddling with the Bundled Examples*.

- To learn how to create your own Tangelo application from scratch, see *Building a Tangelo Web Application from Scratch* and the other *tutorials*.

CHAPTER 2

Using Tangelo

2.1 Installation

There are two ways to install Tangelo: from the Python Package Index (PyPI), or from source. Installing from PyPI is simpler, but limited to public release versions; installing from source is slightly more complicated but allows you to run cutting-edge development versions.

2.1.1 Installing from the Python Package Index

The latest release version of Tangelo can always be found in the Python Package Index. The easiest way to install Tangelo is via Pip, a package manager for Python.

1. Install software dependencies

Install the following software:

- Python 2.7
- Pip

On Linux and OS X computers, your local package manager should be sufficient for installing these. On Windows, please consult this guide for advice about Python and Pip.

2. Install the Tangelo Python package

Use this command in a shell to install the Tangelo package and its dependencies:

```
pip install tangelo
```

You may need to run this command as the superuser, using `sudo` or similar.

2.1.2 Building and Installing from Source

Tangelo is developed on GitHub. If you wish to contribute code, or simply want the very latest development version, you can download, build, and install from GitHub, following these steps:

1. Install software dependencies

To build Tangelo from source, you will need to install the following software:

- Git
- Python 2.7

- Virtualenv 12.0
- Node.js
- Grunt

2. Check out the Tangelo source code

Issue this git command to clone the Tangelo repository:

```
git clone git://github.com/Kitware/tangelo.git
```

This will create a directory named `tangelo` containing the source code. Use `cd` to move into this directory:

```
cd tangelo
```

3. Install Node dependencies

Issue this command to install the necessary Node dependencies via the Node Package Manager (NPM):

```
npm install
```

The packages will be installed to a directory named `node_modules`.

4. Select your Virtualenv version

If the Virtualenv executable you wish to use is invoked in a non-standard way, use the Grunt `config` task to let Grunt know how to invoke Virtualenv. For example, on Arch Linux systems, Virtualenv for Python 2.7 is invoked as `virtualenv2`. In such a case, you would issue the following Grunt command:

```
grunt config:virtualenv:virtualenv2
```

By default, Grunt will assume that Virtualenv is invokable via `virtualenv`. *Note that, in most cases, you will not have to complete this step.*

5. Begin the build process

Issue this command to kick off the Grunt build process:

```
grunt
```

The output will include several phases of action, including: bcreating a virtual environment (in the directory named `venv`), building documentation, creating a Tangelo package, and installing that package to the virtual environment.

Watch the output for any errorrs. In most cases, an error will halt the process, displaying a message to indicate what happened. If you need any help deciphering any such errors, drop us a note at tangelo-users@public.kitware.com.

6. Launch Tangelo

If all has gone well, you can now try to run Tangelo, using this command:

```
./venv/bin/tangelo --examples
```

The Tangelo executable comes from installing the built Tangelo Python package into the development virtual environment, so the command assumes you are in the root of the Tangelo repository, since that is where the virtual environment is created by the build process.

If you open a web browser and go to http://localhost:8080, you should see a welcome message along with the Tangelo Sunrise. If instead you receive an error message about port 8080 not being free, you may need to launch Tangelo on a different port, using a command similar to the following:

```
./venv/bin/tangelo --examples --port 9090
```

Running the Test Suites

Tangelo comes with a battery of server and client tests. To run these, you can invoke the Grunt test task as follows:

```
grunt test
```

This runs both the server and client tests. Each test suite can be run on its own, with:

```
grunt test:server
```

and:

```
grunt test:client
```

Each of these produces a summary report on the command line. To view details such as individual test results, or details about code coverage, you can launch Tangelo to serve the HTML reports with the Grunt serve task:

```
grunt serve:test
```

and point a web browser at http://localhost:50047.

2.2 Setup and Administration

While the *Getting Started* instructions will get you exploring the Tangelo examples in just two commands, Tangelo has a rich set of configuration options that can be used to administer Tangelo effectively. This page will discuss configuration and deployment strategies, including suggestions for best practices.

2.2.1 Configuring and Launching Tangelo

The simplest way to launch a Tangelo server is to use this command:

```
tangelo
```

(This command causes Tangelo to begin serving content out of the current directory on the default port, 8080.)

Tangelo's runtime behaviors are specified via configuration file and command line options. Tangelo configuration files are YAML files representing a key-value store ("associative array" in YAML jargon) at the top level. Each options is specified as a key-value pair: the line starts with the name of the key, then a colon followed by a space, and then the value.

The example configuration found at `/usr/share/tangelo/conf/tangelo.local.conf` reads something like the following:

```
hostname: 0.0.0.0
port:     8080
```

This minimal configuration file specifies that Tangelo should listen on all interfaces for connections on port 8080. By contrast, `tangelo.conf.global` looks like this:

```
hostname: 0.0.0.0
port:     80

user:     nobody
group:    nobody
```

This configuration file is meant for the case when Tangelo is to be installed as a system-level service. It will run on port 80 (the standard port for an HTTP server) and, though it will need to be started with superuser privileges, it will

drop those privleges to run as user `nobody` in group `nobody` to prevent damage to the system should the process be, e.g., hijacked by an attacker.

To run Tangelo using a particular configuration file, `tangelo` can be invoked with the `-c` or `--config` option:

```
tangelo -c ~/myconfig.yaml
```

Alternately, instead of using a yaml file, the configuration can be specified as a JSON string

```
tangelo -c '{"plugins":[{"name":"ui"}]}'
```

Strictly, if the value passed to the config option is *not* the name of an existing file *and* starts with a `{`, then it is expected to be a valid YAML or JSON string.

When the flag is omitted, Tangelo will use default values for all configuration options (see *Configuration Options* below).

Finally, all configuration options can also be specified on the command line. This has the effect of overriding whatever value may be set in the specified configuration file. This can be useful for, e.g., using a single configuration file for multiple Tangelo instances, but varying the port number.

Configuration Options

The following table shows what fields can be included in the configuration file, what they mean, and their default values if left unspecified.

Option	Meaning	Default value
hostname	The hostname interface on which to listen for connections	`localhost`
port	The port number on which to listen for connections	`8080`
root	The path to the directory to be served by Tangelo as the web root	`.` [1]
drop-privileges	Whether to drop privileges when started as the superuser	`True`
sessions	Whether to enable server-side session tracking	`True`
user	The user account to drop privileges to	`nobody` [2]
group	The user group to drop privileges to	`nobody` [2]
access-auth	Whether to protect directories containing a `.htaccess` file	`True`
key	The path to the SSL key	`None` [3] [4]
cert	The path to the SSL certificate	`None` [3] [4]
plugins	A list of plugins to load (see *Plugin Configuration*)	`None` [5] [4]

[1] This is to say, Tangelo serves from the directory in which it was invoked by default.

[2] Your Unix system may already have a user named "nobody" which has the least possible level of permissions. The theory is that system daemons can be run as this user, limiting the damage a rogue process can do. However, if multiple daemons are run this way, any rogue daemon can

2.2.2 Administering a Tangelo Installation

Administering Tangelo on a particular system requires making some decisions about how Tangelo ought to behave, then implementing those decisions in a configuration file.

For example, as the system administrator you might create a directory on the web server machine at `/srv/tangelo` which would serve as the web root, containing the website front page and supporting materials.

You should then prepare a plugin configuration file that, at the very least, activates the Tangelo plugin:

```
enabled: true
path: /usr/share/tangelo/plugins/tangelo
```

This file can be saved to `/etc/tangelo/plugin.conf`.

It remains to configure Tangelo itself. The hostname should reflect the desired external identity of the Tangelo server - perhaps *excelsior.starfleet.mil*. As this is a "global" deployment, we want to listen on port 80 for connections. Since we will need to start Tangelo as root (to gain access to the low-numbered ports), we should also specify a user and group to drop privileges to: these can be the specially created user and group *tangelo*.

The corresponding configuration file might look like this:

```
# Network options.
hostname: excelsior.starfleet.mil
port: 80

# Privilege drop options.
user: tangelo
group: tangelo

# Runtime resources.
root: /srv/tangelo
```

This file should be saved to `/etc/tangelo.conf`, and then Tangelo can be launched with a command like `tangelo -c /etc/tangelo.conf` (running the command with `sudo` may be necessary to allow for port 80 to be bound).

2.2.3 A Note on Version Numbers

Tangelo uses semantic versioning for its version numbers, meaning that each release's version number establishes a promise about the levels of functionality and backwards compatibility present in that release. Tangelo's version numbers come in two forms: *x.y* and *x.y.z*. *x* is a *major version number*, *y* is a *minor version number*, and *z* is a *patch level*.

Following the semantic versioning approach, major versions represent a stable API for the software as a whole. If the major version number is incremented, it means you can expect a discontinuity in backwards compatibility. That is to say, a setup that works for, e.g., version 1.3 will work for versions 1.4, 1.5, and 1.10, but should not be expected to work with version 2.0.

The minor versions indicate new features or functionality added to the previous version. So, version 1.1 can be expected to contain some feature not found in version 1.0, but backwards compatibility is ensured.

The patch level is incremented when a bug fix or other correction to the software occurs.

theoretically gain control of the others. Therefore, the recommendation is to create a new user named "tangelo", that also has minimal permissions, but is only used to run Tangelo in privilege drop mode.

[3] You must also specify both *key* and *cert* to serve content over https.

[4] That is to say, the option is simply unset by default, the equivalent of not mentioning the option at all in a configuration file.

[5] This option can *only* appear in the configuration file; there is no command line equivalent.

Major version 0 is special: essentially, there are no guarantees about compatibility in the 0.*y* series. The stability of APIs and behaviors begins with version 1.0.

In addition to the standard semantic versioning practices, Tangelo also tags the current version number with "dev" in the Git repository, resulting in version numbers like "1.1dev" for the Tangelo package that is built from source. The release protocol deletes this tag from the version number before uploading a package to the Python Package Index.

2.3 Basic Usage

Once it is set up and running, Tangelo's basic usage is relatively straightforward. This chapter explains how Tangelo serves web content, a best practices guide for organizing your content, and how to use HTTP authentication to protect your content.

2.3.1 Serving Web Content

Tangelo's most basic purpose is **to serve web content**. Once Tangelo is running, it will serve content from two types of locations:

Web root directory. Visiting most URLs (whose first path component is not `plugin`; see below) will cause Tangelo to serve content out of the *web root directory*, which is set in the Tangelo configuration file, or by the `-r` (or `--root`) flag when Tangelo is launched (see Setup and Administration). For example, if the web root directory is set to `/srv/tangelo/root`, visiting http://localhost:8080/ would serve content from that directory, and visiting http://localhost:8080/foobar would serve content from `/srv/tangelo/root/foobar`, etc.

Plugin content directories. The URLs rooted at http://localhost:8080/plugin refer to web content served by any active Tangelo plugins. Each active plugin can have static content associated with it, and such content is served from a directory particular to each plugin. For information about how Tangelo plugins work, see *Tangelo Plugins*. In partciular, this means that if there is a subdirectory of the web root directory named `plugin`, *Tangelo will not be able to serve any content from this directory*.

The foregoing examples demonstrate how Tangelo associates URLs to directories and files in the filesystem. URLs referencing particular files will cause Tangelo to serve that file immediately. URLs referencing a directory behave according to the following rules:

1. If the directory contains a file named `index.html`, that file will be served.
2. If Tangelo was launched with the `--list-dir` option, Tangelo will generate and serve a directory listing for the directory. This listing will include hyperlinks to the files contained therein.
3. Tangelo will serve a `403 Forbidden` error indicating that directory listing is disabled.

Furthermore, any URL referring to a Python script, but lacking the final `.py`, names a *web service*; such URLs do not serve static content, but rather run the referred Python script and serve the results (see Tangelo Web Services).

The following table summarizes Tangelo's URL types:

URL type	Example	Behavior
Web root	http://localhost:8080/holodeck3/status.html	serve `/srv/tangelo/root/holodeck3/status.html`
Indexed directory	http://localhost:8080/tenforward	serve `/srv/tangelo/root/tenforward/index.html`
Unindexed directory	http://localhost:8080/warpdrive	serve `403 Forbidden` error, or directory listing for `/srv/tangelo/root/warpdrive`
Web service	http://localhost:8080/lcars/lookup	serve result of executing `run()` function of `/srv/tangelo/lcars/lookup.py`
Plugin	http://localhost:8080/plugin/foobar/	serve content from `foobar` plugin

2.3.2 HTTP Authentication

Tangelo supports HTTP Digest Authentication to password protect web directories. The process to protect a directory is as follows:

1. Go to the directory you wish to protect:

```
cd /srv/engineering/DilithiumChamberStats
```

   The idea is, this directory (which is accessible on the web as http://localhost:8080/DilithiumChamberStats) contains sensitive information, and should be restricted to just certain people who have a password.

2. Create a file there called `.htaccess` and make it look like the following example, customizing it to fit your needs:

```
AuthType digest
AuthRealm USS Enterprise NCC-1701-D
AuthPasswordFile /home/laforge/secret/dilithiumpw.txt
```

   This file requestes digest authnetication on the directory, sets the *authentication realm* to be the string "USS Enterprise NCC-1701-D", and specifies that the acceptable usernames and passwords will be found in the file `/home/laforge/secret/dilithiumpw.txt`.

   Currently, the only supported authentication type is *digest*. The realm will be displayed to the user when prompted for a username and password.

3. Create the password file, using the `tangelo-passwd` program (see tangelo-passwd):

```
$ tangelo-passwd -c ~laforge/secret/dilithiumpw.txt "USS Enterprise NCC-1701-D" picard
Enter password for picard@USS Enterprise NCC-1701-D: <type password here>
Re-enter password: <retype password here>
```

   This will create a new password file. If you inspect the file, you will see a user `picard` associated with an md5 hash of the password that was entered. You can add more users by repeating the command without the `-c` flag, and changing the username.

   At this point, the directory is password protected - when you visit the page, you will be prompted for a username and password, and access to the page will be restricted until you provide valid ones.

2.4 Tangelo Web Services

Tangelo's special power lies in its ability to run user-created *web services* as part of a larger web application. Essentially, each Python file in Tangelo's web space is associated to a URL; requesting this URL (e.g., by visiting it in a browser) will cause Tangelo to load the file as a Python module, run a particular function found within it, and return the output as the content for the URL.

In other words, **Tangelo web services** mean that **Python code** can become **web resources**. Python is a flexible and powerful programming language with a comprehensive standard library and a galaxy of third-party modules providing access to all kinds of APIs and software libraries.

2.4.1 General Services

Here is a simple example of a web service. Suppose `/home/riker/tangelo_html/calc.py` reads as follows:

```
import tangelo

allowed = ["add", "subtract", "multiply", "divide"]
```

```
@tangelo.types(a=float, b=float)
def run(operation, a=None, b=None):
    if a is None:
        return "Parameter 'a' is missing!"
    elif b is None:
        return "Parameter 'b' is missing!"

    try:
        if operation == "add":
            return a + b
        elif operation == "subtract":
            return a - b
        elif operation == "multiply":
            return a * b
        elif operation == "divide":
            return a / b
        else:
            return "Unsupported operation: %s\nAllowed operations are: %s" % (operation, ", ".join(al
    except ValueError:
        return "Could not %s '%s' and '%s'" % (operation, a, b)
    except ZeroDivisionError:
        return "Can't divide by zero!"
```

This is a Python module named `calc`, implementing a very rudimentary four-function calculator in the `run()` function. Tangelo will respond to a request for the URL http://localhost:8080/examples/calculator/calc/add?a=33&b=14 (**without** the trailing `.py`) by loading `calc.py` as a Python module, executing its `run()` function, and returning the result - in this case, the string `47` - as the contents of the URL.

The `run()` function takes three arguments: a positional argument named `operation`, and two keyword arguments named `a` and `b`. Tangelo maps the positional arguments to any "path elements" found after the name of the script in the URL (in this case, `add`), while keyword arguments are mapped to query parameters (`33` and `14` in this case). In other words, the example URL is the equivalent of running the following short Python script:

```
import calc
print calc.run("add", "33", "14")
```

Note that *all arguments are passed as strings.* This is due to the way URLs and associated web technologies work - the URL itself is simply a string, so it is chunked up into tokens which are then sent to the server. These arguments must therefore be cast to appropriate types at run time. The *`tangelo.types()`* decorator offers a convenient way to perform this type casting automatically, but of course you can do it manually within the service itself if it is necessary.

Generally speaking, the web endpoints exposed by Tangelo for each Python file are not meant to be visited directly in a web browser; instead, they provide data to a web application using Ajax calls to retrieve the data. Suppose we wish to use `calc.py` in a web calculator application, which includes an HTML file with two fields for the user to type inputs into, and four buttons, one for each arithmetic operation. An associated JavaScript file might have code like the following:

```
function do_arithmetic(op) {
    var a_val = $("#input-a").val();
    var b_val = $("#input-b").val();

    $.ajax({
        url: "calc/" + op,
        data: {
            a: a_val,
            b: b_val
        },
        dataType: "text",
```

```
        success: function (response) {
            $("#result").text(response);
        },
        error: function (jqxhr, textStatus, reason) {
            $("#result").html(reason);
        }
    });
}

$("#plus").click(function () {
    do_arithmetic("add");
});

$("#minus").click(function () {
    do_arithmetic("subtract");
});

$("#times").click(function () {
    do_arithmetic("multiply");
});

$("#divide").click(function () {
    do_arithmetic("divide");
});
```

The `do_arithmetic()` function is called whenever the operation buttons are clicked; it contains a call to the JQuery `ajax()` function, which prepares a URL with query parameters then retrieves data from it. The `success` callback then takes the response from the URL and places it on the webpage so the user can see the result. In this way, your web application front end can connect to the Python back end via Ajax.

Return Types

The type of the value returned from the `run()` function determines how Tangelo creates content for the associated web endpoint. Since web server communication occurs via textual data, all values returned by web services must eventually be converted to strings. By default, Tangelo accomplishes this by considering all such values to be JSON-encoded. For example, in the calculator example, the `run()` function returns Python `int` value `47`; Tangelo takes this value and applies the standard function `json.dump()` to it, resulting in the string `"47"`, which is delivered to the client for further processing. Similarly, a service that returns a Python `dict` value will be converted to a general JSON-object, making it easy to return structured information from any given web service.

This means that, in the most general case, you can create your own types, equipped with methods for JSON encoding them, and use those are direct return values (see the Python documentation for information on custom JSON encoding). Attempting to return a type that is not JSON-serializable results in a 400 error.

The only exception to the default conversion behavior is that if the service returns a string directly, this value will not be JSON encoded (which entails surrounding it with double-quotes), but simply passed along unchanged. This "escape hatch" enables a service to return any kind of data by encoding it as a string. The `tangelo.content_type()` utility function can be used to specify the intended type of the returned data. For instance, `tangelo.content_type("text/plain")` followed by `return "hello, world"` will result in a text result being sent to the client. More complex types are also possible; e.g., a service might compute a PNG image, then send the PNG data back as a string after calling `tangelo.content_type("application/png")`.

Specifying a Custom Return Type Converter

Similarly to the *tangelo.types()* decorator mentioned above, services can specify a custom return type via the *tangelo.return_type()* decorator. It takes a single argument, a function to convert the object returned from the service function to a string or JSON-serializable value (see *Return Types*):

```
import tangelo

def excited(s):
    return s + "!!!"

@tangelo.return_type(excited)
def run(name):
    return "hello %s" % (name)
```

Given `Data` as an input, this service will return the string `Hello Data!!!` to the client.

A more likely use case for this decorator is special-purpose JSON converters, such as Pymongo's `bson.json_util.dumps()` function, which can handle certain non-standard objects such as Python `datetime` objects when converting to JSON text.

2.4.2 HTTP Status Codes

When something goes wrong during execution of a web service, you may wish to signal to the client what happened. The `tangelo.http_status()` function can be used to set the status code to indicate the class of problem. For instance, if the service invocation does not include the proper required arguments, the service might signal the error by the following:

```
tangelo.http_status(400, "Required Argument Missing")
```

Many HTTP status codes have standard meanings, including default titles (e.g., the default title for 400 is "Bad Request"); invoking `tangelo.http_status()` with only a numerical code will use such a default title. Otherwise, you may include a second string argument to provide a more specific description.

Errors are generally signaled with *4xx* and *5xx* codes. In these cases, the response body may be useful for providing specific information about the error to the client. Such information can be provided as JSON, plain text, HTML, or any other feasible format. Just make sure to call `tangelo.content_type()` to specify the MIME type of the response before using `return` to prepare and send the response.

2.4.3 RESTful Services

Tangelo also supports the creation of REST services. Instead of placing functionality in a `run()` function, such a service has one function per desired REST verb. For example, a rudimentary service to manage a collection of databases might look like the following:

```
import tangelo
import lcarsdb

@tangelo.restful
def get(dbname, query):
    db = lcarsdb.connect("enterprise.starfleet.mil", dbname)
    if not db:
        return None
    else:
        return db.find(query)
```

```
@tangelo.restful
def put(dbname):
    connection = lcarsdb.connect("enterprise.starfleet.mil")
    if not connection:
        return "FAIL"
    else:
        success = connection.createDB(dbname)
        if success:
            return "OK"
        else:
            return "FAIL"
```

The *tangelo.restful()* decorator is used to explicitly mark the functions that are part of the RESTful interface so as to avoid (1) restricting REST verbs to just the set of commonly used ones and (2) exposing every function in the service as part of a REST interface (since some of those could simply be helper functions).

Bear in mind that a function named `run()` will always take precedence over any functions marked with `@tangelo.restful`. This is because `run()` is meant to be agnostic to the HTTP method that was used to invoke it, and as such, has higher precedence when Tangelo is looking for a function to invoke.

2.4.4 Configuring Web Services

You can optionally include a configuration file alongside the service itself. For instance, suppose the following service is implemented in *autodestruct.py*:

```
import tangelo
import starship

def run(officer=None, code=None, countdown=20*60):
    config = tangelo.config()

    if officer is None or code is None:
        return {"status": "failed",
                "reason": "missing officer or code argument"}

    if officer != config["officer"]:
        return {"status": "failed",
                "reason": "unauthorized"}
    elif code != config["code"]:
        return {"status": "failed",
                "reason": "incorrect code"}

    starship.autodestruct(countdown)

    return {"status": "complete",
            "message": "Auto destruct in %d seconds!" % (countdown)}
```

Via the *tangelo.config()* function, this service attempts to match the input data against credentials stored in the module level configuration, which is stored in *autodestruct.yaml* a YAML file containing an associative array (i.e., a key-value store) at its top level:

```
officer: picard
code: echo november golf alpha golf echo four seven enable
```

The two files must have the same base name (*autodestruct* in this case) and be in the same location. Any time the module for a service is loaded, the configuration file will be parsed and loaded as well. The `tangelo.config()` function returns a copy of the configuration dictionary, to prevent an errant service from updating the configuration in

a persistent way. For this reason, it is advisable to only call this function once, capturing the result in a variable, and retrieving values from it as needed.

If the *watch* plugin is enabled (easily done by specifying `--watch` when running Tangelo), changing either file will cause the module to be reloaded the next time it is invoked.

2.4.5 Persistent Storage for Web Services

In contrast to the read-only service configuration, each service also has access to a *persistent data store* that remembers changes made to it from invocation to invocation. This may be accessed by invoking `tangelo.store()` within a service function. Like `tangelo.config()`, the store is a Python dictionary, but anything stored in it will be accessible from a subsequent invocation of the service.

A very simple example would increment `tangelo.store()["count"]` on each invocation, allowing the service to "know" how many times it has been invoked before.

2.5 Tangelo Plugins

Tangelo's capabilities can be extended by creating *plugins* to serve custom content and services from a canonical URL, extend Tangelo's Python runtime environment, and perform specialized setup and teardown actions to support new behaviors. Tangelo ships with several bundled plugins that implement useful features and provide examples of how the plugin system can add value to your Tangelo setup.

Plugins are loaded in the order listed in the tangelo configuration. If one plugin is dependent on another plugin being loaded before it can be initialized, be sure to place the dependent plugin after the plugin it requires.

2.5.1 Structure and Content

A plugin is simply a directory containing a mix of content, documentation, and control directives. Together, these elements determine what services and features the plugin provides to a Tangelo server instance, and how those services and features are prepped and cleaned up. A configuration file supplied to Tangelo at startup time controls which plugins are loaded.

An example plugin's file contents might be as follows:

```
foobar/
    control.py
    config.yaml
    requirements.txt
    info.txt
    python/
        __init__.py
        helper.py
    web/
        foobar.js
        foobar.py
        example/
            index.html
            index.js
```

We can examine the contents piece by piece.

Web Content

The directory `foobar/web` behaves much like any other static and dynamic content served by Tangelo. Content in this directory is served from a base URL of `/plugin/foobar/` (where, `foobar` is the name of this plugin; however, see *Plugin Configuration*). For example, `/plugin/foobar/foobar.js` refers to the file of the same name in the `web` directory; this URL could be used by a web application to include this file in a `<script>` tag, etc. (see *Serving Web Content* for more information).

Dynamic web services also behave as elsewhere: the URL `/plugin/foobar/foobar` will cause Tangelo to run the code found in `foobar.py` and return it to the client, etc. (see *Tangelo Web Services* for more information).

Python Content

A plugin may also wish to export some Python code for use in web services. The `python` directory or a `python.py` file within a plugin is imported as a module in the `tangelo.plugin` namespace with the plugin's name.

In the foobar plugin example, such content appears in `foobar/python/__init__.py`. This file, for example, might contain the following code:

```
import helper

def even(n):
    return n % 2 == 0
```

When the foobar plugin is loaded by Tangelo, the contents of `python/__init__.py` are placed in a virtual package named `tangelo.plugin.foobar`. This enables web services to use the `even()` functions as in the following example:

```
import tangelo
# It isn't necessary to explicitly import tangelo.plugin.foobar, as it is
# added to the tangelo.plugin namespace when tangelo starts.

def run(n):
    tangelo.content_type("text/plain")
    return "even" if tangelo.plugin.foobar.even(n) else "odd"
```

To export "submodules" that will appear in the `tangelo.plugin.foobar` namespace, note that `__init__.py` uses the `import` statement to cause the `helper` module to appear within its scope; this module can now be addressed with `tangelo.plugin.foobar.helper`, and any functions and data exported by `helper` will become available for use in web services as well.

The bundled *bokeh* plugin contains an example of exporting a decorator function using this technique.

2.5.2 Setup and Teardown

The file `foobar/control.py` defines *setup* and *teardown* actions for each plugin. For example, the contents of that file might be as follows:

```
import tangelo

def setup(config, store):
    tangelo.log("FOOBAR", "Setting up foobar plugin!")

def teardown(config, store):
    tangelo.log("FOOBAR", "Tearing down foobar plugin!")
```

Whenever Tangelo loads (unloads) the foobar plugin, it will import `control.py` as a module and execute any `setup()` (`teardown()`) function it finds, passing the configuration and persistent storage (see *Plugin Configuration*) to it as arguments. If during setup the function raises any exception, the exception will be printed to the log, and Tangelo will abandon loading the plugin and move to the next one.

The `setup()` function can also cause arbitrary CherryPy applications to be mounted in the plugin's URL namespace. `setup()` can optionally return a list of 3-tuples describing the applications to mount. Each 3-tuple should contain a CherryPy application object, an optional configuration object associated with the application, and a string describing where to mount the application. This string will automatically be prepended with the base URL of the plugin being set up. For instance:

```
import tangelo.plugin.foobar

def setup(config, store):
    app = tangelo.plugin.foobar.make_cherrypy_app()
    appconf = tangelo.plugin.foobar.make_config()

    return [(app, appconf, "/superapp")]
```

When the `foobar` plugin is loaded, the URL `/plugin/foobar/superapp` will serve the CherryPy application implemented in `app`. Any such applications are also unmounted when the plugin is unloaded.

2.5.3 Plugin Configuration

Plugin configuration comes in two parts: specifying which plugins to load, and specifying particular behavior for each plugin.

Enabling Plugins

The Tangelo configuration file supports an option `plugin` that specifies a plugin configuration. The option's value should be a YAML expression consisting of a list of objects, one for each plugin under consideration. The objects themselves are relatively simple:

```
- name: foobar
  path: /path/to/foobar/plugin

- name: quux
  path: path/to/quux

- name: docs
```

Each contains a required `name` property and an optional `path` property describing where to find the plugin materials (i.e., the example directory shown above).

Note that you can enable a bundled plugin (see *Bundled Plugins*) by omitting the `path` property. In this case, Tangelo searches for a plugin by the given name in the plugins that come bundled with Tangelo. In the example above, the *docs* plugin will be enabled. This is useful for enabling a "standard" plugin without having to know where Tangelo keeps it.

The `plugins` option can simply be omitted when you do not wish to load any plugins.

When Tangelo is started with a `plugins` option in its configuration file, each plugin listed will be loaded before Tangelo begins serving content to the web. Because it is assumed that any plugins specified are necessary for the Tangelo application being launched, any error in loading any of the plugins will result in aborting the startup process (logging errors as they occur).

Inversely, when Tangelo is shut down, each plugin will be unloaded in turn (enabling, e.g., cleanup actions such as flushing buffers to disk, committing pending database transactions, closing connections, etc.). In this case, if a plugin cannot be unloaded for any reason, Tangelo's shutdown will continue, and you should clean up after the faulty plugin manually.

Plugin Setup

Some plugins may need to be set up before they can be properly used. Plugin setup consists of two steps: installing Python dependencies, if any, and consulting any informational messages supplied by the plugin.

In the example *foobar* plugin, note that the root directory includes a `requirements.txt` file. This is simply a pip requirements file declaring what Python packages the plugin needs to run. You can install these with a command similar to `pip install -r foobar/requirements.txt`.

Secondly, some plugins may require some other action to be taken before they work. The plugin authors can describe any such instructions in the `info.txt` file. After installing the requirements, you should read this file to see if anything else is required. For instance, a package may need to bootstrap after it's installed by fetching further resources or updates from the web; in this case, `info.txt` would explain just how to accomplish this bootstrapping.

These steps constitute a standard procedure when retrieving a new plugin for use with your local Tangelo installation. For instance, if the *foobar* plugin resides in a GitHub repository, you would first find a suitable location on your local computer to clone that repository. Then you would invoke `pip` to install the required dependencies, then take any action specified by `info.txt`, and finally create an entry in the Tangelo plugin configuration file. When Tangelo is started (or when the plugin registry is refreshed), the new plugin will be running.

Configuring Plugin Behavior

The file `foobar/config.yaml` describes a YAML associative array representing the plugin's configuration data. This is the same format as web service configurations (see *Configuring Web Services*), and can be read with the function `tangelo.plugin_config()`.

Similarly, plugins also have a editable persistent store, accessed with the `tangelo.plugin_store()` function.

Both the configuration and the persistent store and passed as arguments to `setup()` and `teardown()` in the control module.

2.5.4 Loading and Unloading

When plugins are loaded or unloaded, Tangelo takes a sequence of particular steps to accomplish the effect.

Loading a Plugin

Loading a plugin consists of the following actions:

1. The configuration is loaded from `config.yaml`.
2. An empty persistent store is created.
3. Any python content is set up by creating a virtual package called `tangelo.plugin.<pluginname>`, and exporting the contents of `python/__init__.py` to it.
4. The `control.py` module is loaded, and `control.setup()` is invoked, passing the configuration and fresh persistent store to it.
5. If `setup()` returns a result, the list of CherryPy apps expressed in the `"apps"` property of it are mounted.

Steps 3, 4, and 5 are not taken if the corresponding content is not present. If any of those steps raises an exception, the error will be logged and the Tangelo startup process will abort.

Unloading a Plugin

Unloading a plugins consists of the follow actions (which serve to undo the corresponding setup actions):

1. Any python content present in `tangelo.plugin.<pluginname>` is torn down by deleting the virtual package from the runtime.
2. Any CherryPy applications are unmounted.
3. If the control module contains a `teardown()` function, it is invoked, passing the configuration and persistent store to it.

If an exception occurs during step 3, the `teardown()` function will not finish executing, but Tangelo shutdown will continue with the unloading of the rest of the plugins and eventual exiting of the Tangelo process.

CHAPTER 3

Tutorials

3.1 Building a Tangelo Web Application from Scratch

This tutorial will go through the steps of building a working, albeit simple, Tangelo application from the ground up. Most Tangelo applications consist of (at least) three parts: an HTML document presenting the form of the application as a web page, a JavaScript file to drive dynamic content and behavior within the web page, and a Python service to perform serverside processing. The tutorial application will have one of each to demonstrate how they will fit together.

3.1.1 String Reverser

The tutorial application will be a *string reverser*. The user will see a form where a word can be entered, and a button to submit the word. The word will then make a trip to the server, where it will be reversed and returned to the client. The reversed word will then be displayed in the web page.

Preparing the Stage

Tangelo will need to be running in order for the application to work. The quickstart instructions will be sufficient (see *Getting Started*):

```
tangelo
```

This should launch Tangelo on *localhost*, port 8080 (also known as http://localhost:8080/).

First we need a place to put the files for the application. We will serve the application out of a specialized directory to contain several Tangelo applications. It is a good practice to house each application in its own subdirectory - this keeps things organized, and allows for easy development of web applications in source control systems such as GitHub:

```
cd ~
mkdir tangelo_apps
cd tangelo_apps
mkdir reverser
```

Next, we need to serve this directory to the web. We'd also like to be able to see the directory contents and Python source as we edit our application, since we're in "developer mode":

```
tangelo --list-dir --show-py
```

By default, Tangelo serves files from the current directory. It would also be possible to use the web root option by adding `--root ~/tangelo_apps` to make the desired root directory explicit.

Visiting http://localhost:8080/reverser in a web browser should at this point show you a directory listing of no entries. Let's fix that by creating some content.

HTML

The first step is to create a web page. In a text editor, open a file called `index.html` and copy in the following:

```
1  <!DOCTYPE html>
2  <title>Reverser</title>
3
4  <!-- Boilerplate JavaScript -->
5  <script src=http://code.jquery.com/jquery-1.11.0.min.js></script>
6  <script src=/js/tangelo.js></script>
7
8  <!-- The app's JavaScript -->
9  <script src=myapp.js></script>
10
11 <!-- A form to submit the text -->
12 <h2>Reverser</h2>
13 <div class=form-inline>
14     <input id=text type=text>
15     <button id=go class="btn btn-class">Go</a>
16 </div>
17
18 <!-- A place to show the output -->
19 <div id=output></div>
```

This is a very simple page, containing a text field (with ID `text`), a button (ID `go`), and an empty div element (ID `output`). Feel free to reload the page in your browser to see if everything worked properly.

Next we need to attach some behaviors to these elements.

JavaScript

We want to be able to read the text from the input element, send it to the server, and do something with the result. We would like to do this whenever the "Go" button is clicked. The JavaScript to accomplish this follows - place this in a file named `myapp.js` (to reflect the script tag in line 9 of `index.html`):

```
1  $(function () {
2      $("#go").click(function () {
3          var text = $("#text").val();
4          $.getJSON("myservice?text=" + encodeURIComponent(text), function (data) {
5              $("#output").text(data.reversed);
6          });
7      });
8  });
```

Several things are happening in this short bit of code, so let's examine them one by one. Line 1 simply makes use of the jQuery `$()` function, which takes a single argument: a function to invoke with no arguments when the page content is loaded and ready.

Line 2 uses the "CSS selector" variant of the `$()` function to select an element by ID - in this case, the "go" button - and attach a behavior to its "click" callback.

Line 3 - the first line of the function executed on button click - causes the contents of of the text input field to be read out into the variable `text`.

Line 4 uses the jQuery convenience function `$.getJSON()` to initiate an ajax request to the URL http://localhost:8080/reverser/myservice, passing in the text field contents as a query argument. When the server has a response prepared, the function passed as the second argument to `$.getJSON()` will be called, with the response as the argument.

Line 5 makes use of this response data to place some text in the blank div. Because `$.getJSON()` converts the server response to a JSON object automatically, we can simply get the reversed word we are looking for in `data.reversed`. The output div in the webpage should now be displaying the reversed word.

The final component of this application is the server side processing itself, the service named `myservice`.

Python

The Python web service will perform a reversal of its input. The following Python code accomplishes this (save it in a file named `myservice.py`, again, to reflect the usage of that name in the `myapp.js` above):

```
def run(text=""):
    return {"reversed": text[::-1]}
```

This short Python function uses a terse array idiom to reverse the order of the letters in a string. Note that a string goes into this function from the client (i.e., the call to `$.getJSON` is line 4 of `myapp.js`), and a Python dict comes out. The dict is automatically converted to JSON-encoded text, which the `$.getJSON()` function automatically converts to a JavaScript object, which is finally passed to the anonymous function on line 4 of `myapp.js`.

Tying it All Together

The application is now complete. Once more refresh the page at http://localhost:8080/reverser/, type in your favorite word, and click the "Go" button. If all goes well, you should see your favorite word, reversed, below the text input field!

Discussion

Of course, we did not need to bring the server into this particular example, since JavaScript is perfectly suited to reversing words should the need arise. However, this example was meant to demonstrate how the three pieces - content, dynamic clientside behavior, and serverside processing - come together to implement a full, working web application.

Now imagine that instead of reversing the word, you wanted to use the word as a search index in a database, or to direct the construction of a complex object, or to kick off a large, parallel processing job on a computation engine, or that you simply want to use some Python library that has no equivalent in the JavaScript world. Each of these cases represents some action that is difficult or impossible to achieve using clientside JavaScript. By writing Tangelo web services you can enrich your application by bringing in the versatility and power of Python and its libraries.

3.2 Driving a Visualization with SQLAlchemy

This tutorial demonstrates how you might set up a SQL database to serve data to a visualization application using Tangelo. This example demonstrates how an application-specific approach to building and using a database can provide flexibility and adaptability for your Tangelo application.

In this tutorial we will

- obtain *Star Trek: The Next Generation* episode data
- create an SQLite database from it

- establish some object-relational mapping (ORM) classes using SQLAlchemy
- visualize the data using Vega

To begin this tutorial, create a fresh directory somewhere where we can build a new project:

```
mkdir tng
cd tng
```

Here, we will create a database, along with appropriate ORM infrastructure; write some web services to be used as runtime data sources to pull requested data from the database; and a simple web frontend made from HTML and JavaScript, using the Vega visualization library.

For convenience, you can download and unpack a ZIP archive of the entire web application as well: tng.zip. However, downloading and inspecting the files as we go, or writing them by hand from the listings below, may encourage a deeper understanding of what's going on.

3.2.1 Getting the Data

The episode data, gleaned from Memory Alpha by hand, is in these two CSV files:

- episodes.csv
- people.csv

If you take a look in these files, you'll see some basic data about episodes of *Star Trek: The Next Generation*. *episodes.csv* contains one row per episode, indicating its overall episode number, season/episode, title, airdate, a link to the associated Memory Alpha article, and numeric indices into *people.csv* to indicate who wrote each teleplay, who developed each story, and who directed each episode.

3.2.2 Creating the Database

SQLAlchemy is a Python library that provides a programmatic API for creating, updating, and accessing SQL databases. It includes an *object-relational mapping (ORM)* component, meaning it provides facilities for writing Python classes that transparently maintain a connection to the database, changing it as the object is updated, etc.

To install SQLAlchemy, you can use the Python package manager `pip` as follows:

```
pip install sqlalchemy==0.9.8
```

(The version specifier may not be necessary, but this tutorial was designed using SQLAlchemy 0.9.8.)

Establishing ORM Classes

The first step in this visualization project is to establish our data model by writing some ORM classes, then using those classes to read in the CSV files from above and flow them into an SQLite database. The file startrek.py has what we need. Let's analyze it, section by section.

First, we need some support from `SQLAlchemy`:

```
from sqlalchemy import create_engine
engine = create_engine("sqlite:///tngeps.db", echo=True, convert_unicode=True)

from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()

from sqlalchemy.orm import sessionmaker
DBSession = sessionmaker(bind=engine)
```

`create_engine` simply gives us a handle to a database - this one will exist on disk, in the file `tngeps.db`. The `echo` keyword argument controls whether the behind-the-scenes translation to SQL commands will appear on the output when changes occur. For now it may be a good idea to leave this set to `True` for the sake of education.

`declarative_base` is a base class that provides the foundation for creating ORM classes to model our data, while the `sessionmaker` function creates a function that can be used to establish a connection to the database.

Next we need to import some types for the columns appearing in our data:

```
from sqlalchemy import Column
from sqlalchemy import Date
from sqlalchemy import Integer
from sqlalchemy import String
```

`Date` and `String` will be used to store actual data values, such as airdates, and names of people and episodes. `Integer` is useful as a unique ID field for the various objects we will be storing.

Now, we finally get to the data modeling:

```
from sqlalchemy import Table
from sqlalchemy import ForeignKey
from sqlalchemy.orm import relationship

episode_teleplays = Table("episode_teleplays", Base.metadata,
                          Column("episode_id", Integer, ForeignKey("episodes.id")),
                          Column("teleplay_id", Integer, ForeignKey("people.id")))

episode_stories = Table("episode_stories", Base.metadata,
                        Column("episode_id", Integer, ForeignKey("episodes.id")),
                        Column("story_id", Integer, ForeignKey("people.id")))

episode_directors = Table("episode_directors", Base.metadata,
                          Column("episode_id", Integer, ForeignKey("episodes.id")),
                          Column("director_id", Integer, ForeignKey("people.id")))


class Episode(Base):
    __tablename__ = "episodes"

    id = Column(Integer, primary_key=True)
    season = Column(Integer)
    episode = Column(Integer)
    title = Column(String)
    airdate = Column(Date)
    teleplay = relationship("Person", secondary=episode_teleplays, backref="teleplays")
    story = relationship("Person", secondary=episode_stories, backref="stories")
    director = relationship("Person", secondary=episode_directors, backref="directors")
    stardate = Column(String)
    url = Column(String)

    def __repr__(self):
        return (u"Episode('%s')" % (self.title)).encode("utf-8")


class Person(Base):
    __tablename__ = "people"

    id = Column(Integer, primary_key=True)
    name = Column(String)

```

```
58      def __repr__(self):
59          return (u"Person('%s')" % (self.name)).encode("utf-8")
```

First take a look at the classes `Episode` and `Person`. These make use of SQLAlchemy's `declarative_base` to establish classes whose structure reflect the semantics of a table in the database. The `__tablename__` attribute gives the name of the associated table, while the other named attributes give the names of the columns appearing in it, along with the data type.

Note that the `teleplay`, `story`, and `director` attributes of `Episode` are a bit more complex than the others. These are fields that cross-reference into the "people" table: each `Episode` may have multiple writers and story developers [1], each of which is a `Person`. Of course, a particular Person may also be associated with multiple Episodes, so a special "many-to-many" relationship exists between `Episode` and `Person` when it comes to the `teleplay`, `story`, and `director` columns. These are expressed in the "association table" declarations appearing in lines 25, 29, and 33. Such tables simply contain one row for each unique episode-person connection; they are referenced in the appropriate column declaration (lines 46-48) to implement the many-to-many relation.

The effect of these ORM classes is that when, e.g., an `Episode` object is queried from the database, its `teleplay` property will contain a list of the correct `Person` objects, having been reconstructed by examining the "episode_teleplays" table in the database.

Creating the Database

Now it just remains to use the ORM infrastructure to drive the parsing of the raw data and creation of the actual database. The file build-db.py contains a Python script to do just this. Let's examine this script, section by section. First, as always, we need to import some standard modules:

```
1  import csv
2  import datetime
3  import sys
```

Next, we need some stuff from *startrek.py*:

```
5  from startrek import Base
6  from startrek import engine
7  from startrek import DBSession
8  from startrek import Episode
9  from startrek import Person
```

Now, let's go ahead and slurp in the raw data from the CSV files:

```
12 try:
13     with open("episodes.csv") as episodes_f:
14         with open("people.csv") as people_f:
15             episodes = list(csv.reader(episodes_f))
16             people = list(csv.reader(people_f))
17 except IOError as e:
18     print >>sys.stderr, "error: %s" % (e)
19     sys.exit(1)
```

We now have two lists, `episodes` and `people`, containing the row data. Before we continue, we need to "activate" the ORM classes, and connect to the database:

```
22 Base.metadata.create_all(engine)
23 session = DBSession()
```

Now let's add the rows from *people.csv* to the database:

[1] And even directors, though this only happened in one episode when the original director was fired and a replacement brought on.

```
26 people_rec = {}
27 for i, name in people[1:]:
28     i = int(i)
29
30     p = Person(id=i, name=name.decode("utf-8"))
31
32     people_rec[i] = p
33     session.add(p)
```

This loop simply runs through the rows of the people data (excluding the first "row," which is just the header descriptors), saving each in a table indexed by ID: line 30 creates a `Person` object, while line 33 causes a row in the appropriate tables to be created in the database (using ORM magic). We want the saved table so we can reference `Person` objects later.

Now that we have all the people loaded up, we can put the episode data itself into the database:

```
37 for i, season, ep, title, airdate, teleplay, story, director, stardate, url in episodes[1:]:
38     # Extract the fields that exist more or less as is.
39     i = int(i)
40     season = int(season)
41     ep = int(ep)
42
43     # Parse the (American-style) dates from the airdate field, creating a Python
44     # datetime object.
45     month, day, year = airdate.split("/")
46     month = "%02d" % (int(month))
47     day = "%02d" % (int(day))
48     airdate = datetime.datetime.strptime("%s/%s/%s" % (month, day, year), "%m/%d/%Y")
49
50     # Create lists of writers, story developers, and directors from the
51     # comma-separated people ids in these fields.
52     teleplay = map(lambda writer: people_rec[int(writer)], teleplay.split(","))
53     story = map(lambda writer: people_rec[int(writer)], story.split(","))
54     director = map(lambda writer: people_rec[int(writer)], director.split(","))
55
56     # Construct an Episode object, and add it to the live session.
57     ep = Episode(id=int(i), season=season, episode=ep, title=title.decode("utf-8"), airdate=airdate,
58     session.add(ep)
```

This is a loop running through the rows of the episode data, pulling out the various fields, possibly converting them to appropriate Python types. The `teleplay`, `story`, and `director` columns in particular are converted to lists of `Person` objects (by looking up the appropriate `Person` in the table we created earlier). Line 58 adds the newly created `Person` to the database. The many-to-many relationships we established earlier will be invoked here, updating the association tables according to the `Person` objects present in each `Episode` object's fields.

Finally, we must commit the accumulated database operations:

```
61 session.commit()
62 sys.exit(0)
```

If you have downloaded the data files, *startrek.py*, and *build-db.py* all to the same directory, you should be able to build the database with this command:

```
python build-db.py
```

Because we directed the database engine to echo its activity to the output, you should see SQL commands fly by as they are generated by the various calls to `session.add()`. This should result in a new file being created, *tngeps.db*. This is an SQLite database file, and should contain all of the data and relationships established in the raw data files.

3.2.3 Writing Data Services

Now we have a database and some ORM classes to query it. The next step is to write a web service that can pull out some data that we need. We are going to use Vega to create some basic charts of the episode data, and Vega visualizations rely on data presented as a list of JSON objects, one per data point. As a starting point for a visualization project on Star Trek episode data, let's tally up the number of episodes written or developed by each person in the *people* table, and use Vega to render a bar chart. To do so, we need to query the database and count how many episodes each person is associated to. We can use the ORM classes to accomplish this. Let's analyze the file writers.py to see how. First, module imports:

```
1  import json
2  import tangelo
3
4  from startrek import DBSession
5  from startrek import Episode
```

Now, the meat of the service, the `run()` function:

```
8  @tangelo.types(sort=json.loads)
9  def run(sort=False):
```

The function signature says that the sort parameter, if present, should be a query argument in JSON-form, defaulting to `False`. We will use this parameter to sort the list of episode writers by the number of episodes worked on (since this may be an interesting thing to look into). Next we need a connection to the database:

```
10     session = DBSession()
```

and some logic to aggregate writers' episode counts ():

```
12     count = {}
13     episodes = session.query(Episode)
14     for ep in episodes:
15         seen = set()
16         for writer in ep.teleplay:
17             count[writer] = count.get(writer, 0) + 1
18             seen.add(writer)
19
20         for writer in ep.story:
21             if writer not in seen:
22                 count[writer] = count.get(writer, 0) + 1
```

This retrieves a list of `Episode` objects from the database (line 13), then loops through them, incrementing a count of writers in a dictionary (being careful not to double count writers listed under both *teleplay* and *story* for a given episode).

Now we convert the dictionary of collected counts into a list of objects suitable for a Vega visualization:

```
24     results = [{"name": r.name, "count": count[r]} for r in sorted(count.keys(), key=lambda x: x.id)]
25     if sort:
26         results.sort(key=lambda x: x["count"], reverse=True)
27
28     return results
```

This line converts each `Person` object into a Python dictionary after sorting by the numeric ID (which, because of how the data was collected, roughly corresponds to the order of first involvement in writing for *Star Trek: The Next Generation*). If the `sort` parameter was `True`, then the results will be sorted by descending episode count (so that the most frequent writers will appear first, etc.). And finally, of course, the function returns this list of results.

With this file written we have the start of a web application. To see how things stand, you can launch Tangelo to serve this directory to the web,

```
tangelo --root .
```

and then visit http://localhost:8080/writers?sort=false to see the list of JSON objects that results.

3.2.4 Designing a Web Frontend

The final piece of the application is a web frontend. Ours will be relatively simple. Here is the webpage itself, in index.html:

```
1  <!doctype html>
2  <title>Star Trek: The Next Generation Episode Writers</title>
3
4  <script src=http://trifacta.github.io/vega/lib/d3.v3.min.js></script>
5  <script src=http://trifacta.github.io/vega/vega.js></script>
6  <script src=index.js></script>
7
8  <div id=chart></div>
```

This is a very simple HTML file with a `div` element (line 9), in which we will place a Vega visualization.

Next, we have some simple JavaScript to go along with this HTML file, in index.js:

```
1  window.onload = function () {
2      d3.json("barchart.json", function (spec) {
3          vg.parse.spec(spec, function (chart) {
4              chart({
5                  el: "#chart"
6              }).update();
7          });
8      });
9  };
```

This simply parses a Vega visualization specification into a JavaScript object, which it then passes to `vg.parse.spec()`, which in turn renders it into the `#chart` element of the web page [2].

The final piece of the puzzle is the Vega specification itself, in barchart.json.:

```
1  {
2      "name": "barchart",
3      "width": 4000,
4      "height": 500,
5      "data": [
6          {
7              "name": "table",
8              "url": "writers?sort=true"
9          }
10     ],
11     "scales": [
12         {
13             "name": "y",
14             "type": "linear",
15             "range": "height",
16             "domain": {
17                 "data": "table",
18                 "field": "data.count"
19             }
20         },
```

[2] For more information on how Vega works, and what you can do with it, see the Vega website at http://trifacta.github.io/vega.

```
        {
            "name": "x",
            "type": "ordinal",
            "range": "width",
            "domain": {
                "data": "table",
                "field": "data.name"
            }
        }
    ],
    "axes": [
        {
            "type": "x",
            "scale": "x",
            "values": []
        },
        {
            "type": "y",
            "scale": "y",
            "grid": false
        }
    ],
    "marks": [
        {
            "type": "rect",
            "from": {
                "data": "table"
            },
            "properties": {
                "enter": {
                    "x": {
                        "scale": "x",
                        "field": "data.name",
                        "offset": -1
                    },
                    "width": {
                        "scale": "x",
                        "band": true,
                        "offset": -1
                    },
                    "y": {
                        "scale": "y",
                        "field": "data.count"
                    },
                    "y2": {
                        "scale": "y",
                        "value": 0},
                    "fill": {
                        "value": "steelblue"
                    }
                },
                "update": {
                    "fill": {
                        "value": "steelblue"
                    }
                },
                "hover": {
                    "fill": {
```

```
                        "value": "firebrick"
                    }
                }
            }
        },
        {
            "type": "text",
            "from": {
                "data": "table"
            },
            "properties": {
                "enter": {
                    "x": {
                        "scale": "x",
                        "field": "data.name"
                    },
                    "dx": {
                        "value": 5
                    },
                    "y": {
                        "value": 505
                    },
                    "angle": {
                        "value": 45
                    },
                    "fill": {
                        "value": "black"
                    },
                    "text": {
                        "field": "data.name"
                    },
                    "font": {
                        "value": "Helvetica Neue"
                    },
                    "fontSize": {
                        "value": 15
                    }
                }
            }
        }
    ]
}
```

This specification describes a data-driven bar chart. You may wish to experiment with this file (for example, changing the colors used, or the width and height of the visualization, or by setting the `sort` parameter in the `url` property to `false`), but as-is, the specification will deliver a bar chart of *Star Trek: The Next Generation* writers, ordered by most episodes worked on.

3.2.5 Putting It All Together

Your web application is complete! If Tangelo is not running, start it with

```
tangelo --root .
```

and then visit http://localhost:8080. You should see a bar chart appear, in which the trekkies out there will surely recognize some of the names.

In summary, we performed the following actions to write a Tangelo application driven by a database:

1. Got some data we wanted to visualize.
2. Developed some ORM infrastructure to model the data, using SQLAlchemy.
3. Imported the data into a new database, using the data and the ORM models.
4. Developed a web service using SQLAlchemy to retrieve some of the data and then shape it into a form we needed for Vega.
5. Developed a Vega specification that can take the web service results and render it as a bar chart.
6. Developed a simple web application to give Vega a place to work and display its results.

Of course, this is just a simple example of what you can do. With Python's power, flexibility, and interfaces to many kinds of databases and visualization systems, you can develop a Tangelo application that is suited to whatever problem you happen to be working on.

3.3 Fiddling with the Bundled Examples

The previous tutorials have shown how to develop various types of Tangelo applications, but you might also want to simply fiddle around with the example applications that come bundled with Tangelo. Since the bundled examples are treated in a somewhat special manner by Tangelo, this tutorial explains how you can make a copy of the example applications, set them up with a Tangelo configuration file, and then experiment and see the results yourself.

The examples can be viewed at http://localhost:8080 by launching Tangelo in example mode:

```
tangelo --examples
```

The index page contains links to different examples, each of which is served as the web content of one plugin or another. For example, http://localhost:8080/plugin/vis/examples/barchart/ serves an example of the barchart that is part of the *vis* plugin.

3.3.1 The Example Web Applications

The example web applications are bundled in the Tangelo Python package with the following directory structure:

```
tangelo.ico
web/
    ...
plugin/
    ...
```

The file `tangelo.ico` is served by Tangelo statically as the default favicon, while the `web` directory contains the example site's front page (including the Tangelo Sunrise, and the menu of links to the individual examples). The examples themselves are contained within the `plugin` directory. As an example, the directory for the *Bokeh* plugin looks like this:

```
plugin/
    bokeh/
        requirements.txt
        python/
            __init__.py
        web/
            bokeh.js
            examples/
                iris/
                    index.html
                    iris.py
```

This plugin contains a Python component and a clientside component, as well as the Iris example application, which demonstrates how the pieces fit together (for information about how plugins work, see *Tangelo Plugins*). The web application content in the `iris` directory would be a good place to play around to discover for yourself how this plugin works.

Therefore, we will want to make a private copy of the `web` and `plugin` directories in order to experiment with the contents of the example web applications.

3.3.2 Making a Copy

To actually play with the examples, we'd like to set up our own sandbox, copy these materials into it, configure Tangelo to run with the appropriate plugins, and finally serve our own version of the example applications.

Step 1: Create a Sandbox

The examples are bundled as *package data* with the Tangelo Python package, meaning they will be found within the `tangelo/pkgdata` subdirectory of the `site-packages` directory of the Python installation that contains Tangelo. On a typical Linux Python installation, this directory might be `/usr/lib/python2.7/site-packages/tangelo/pkgdata`. Because different Python setups may behave differently with respect to where such files are kept, Tangelo includes a program `tangelo-pkgdata` that simply reports the full path to the `pkgdata` directory. Using this program, the following sequence of shell commands will create an area where we can safely modify and otherwise experiment with the examples:

```
$ cd ~
$ mkdir tangelo-examples
$ cd tangelo-examples
$ cp -r `tangelo-pkgdata` .
```

(Enclosing a command in backticks causes the shell to run the enclosed program and substitute its output in the original command. You can also run `tangelo-pkgdata` manually, inspect the output, and copy it into your own manual shell command as well.)

Step 2: Configure the Plugins

We will want to have Tangelo serve the `web` directory, while loading the appropriate plugins from the `plugin` directory. For the latter, we will need a configuration file to declare the plugins:

This very simple configuration simply names the plugins we need, together with relative paths stating where the plugins can be found. Create a file `config.yaml` (in the `tangelo-examples` directory) and copy the configuration into it.

Step 3: Launch Tangelo

Now that we have web materials, plugins, and a configuration, we just need to start Tangelo:

```
$ tangelo --root web --config config.yaml
```

Tangelo should begin serving the example site at http://localhost:8080 (if you get an error about port 8080 not being free, try again with the `--port` option to select a different port).

Step 4: Fiddle!

Now you can go into the various `web` subdirectories of the plugin paths, make changes, and observe them live. If you find things don't update as expected, you can try restarting the server (certain features of plugins can only be instantiated when Tangelo first starts up).

Try changing the data values in the *mapping* plugin examples, or changing how some of the web services retrieve, process, or format their output data. With a safe, hands-on approach, you can learn a lot about how Tangelo operates.

CHAPTER 4

Command Line Utilities

4.1 tangelo

tangelo [-h] [-c FILE] [-nc] [-a] [-na] [-p] [-np]
[–hostname HOSTNAME] [–port PORT] [-u USERNAME]
[-g GROUPNAME] [-r DIR] [–vtkpython FILE] [–verbose] [–quiet]
[–version] [–key FILE] [–cert FILE]

Start a Tangelo server.

Optional argument	Effect
-h, –help	show this help message and exit
-c FILE, –config FILE	specifies configuration file or json string to use
-nc, –no-config	skips looking for and using a configuration file
-a, –access-auth	enable HTTP authentication (i.e. processing of .htaccess files) (default)
-na, –no-access-auth	disable HTTP authentication (i.e. processing of .htaccess files)
-p, –drop-privileges	enable privilege drop when started as superuser (default)
-np, –no-drop-privileges	disable privilege drop when started as superuser
-s, –sessions	enable server-side session tracking (default)
-ns, –no-drop-privileges	disable server-side session tracking
–hostname HOSTNAME	overrides configured hostname on which to run Tangelo
–port PORT	overrides configured port number on which to run Tangelo
-u USERNAME, –user USERNAME	specifies the user to run as when root privileges are dropped
-g GROUPNAME, –group GROUPNAME	specifies the group to run as when root privileges are dropped
-r DIR, –root DIR	the directory from which Tangelo will serve content
–examples	serve the Tangelo example applications
–verbose, -v	display extra information as Tangelo runs
–quiet, -q	reduce the amount of information displayed
–version	display Tangelo version number
–key FILE	the path to the SSL key. You must also specify –cert to serve content over https.
–cert FILE	the path to the SSL certificate. You must also specify –key to serve content over https.

4.1.1 Example Usage

To start a Tangelo server with the default configuration, serving from the current directory:

```
tangelo
```

This starts Tangelo on port 8080.

To serve the example applications that come bundled with Tangelo:

```
tangelo --examples
```

To control particular options, such as the port number (overriding the value specified in the config) file:

```
tangelo --port 9090
```

4.2 tangelo-passwd

tangelo-passwd [-h] [-c] passwordfile realm user

Edit .htaccess files for Tangelo

Positional argument	Meaning
passwordfile	Password file
realm	Authentication realm
user	Username

Optional argument	Effect
-h, –help	Show this help message and exit
-c, –create	Create new password file

4.2.1 Example Usage

To create a new password file:

```
tangelo-passwd -c secret.txt romulus tomalak
```

(Then type in the password as prompted.)

To add a user to the file:

```
tangelo-passwd secret.txt Qo\'noS martok
```

(Again, type in password.)

To overwrite a new password file on top of the old one:

```
tangelo-passwd -c secret.txt betazed troi
```

CHAPTER 5

The Tangelo API

5.1 Python Web Service API

The web service API is a collection of Python functions meant to help write web service scripts in as "Pythonic" a way as possible. The functionality is divided into severul areas: core services for generally useful utilities; HTTP interaction, for manipulating request headers, retrieving request bodies, and formatting errors; and web service utilities to supercharge Python services.

5.1.1 Core Services

`tangelo.`**`log`**`(`[*context*]`, `*msg*[`, `*lvl=loglevel*]`)`

Writes a message *msg* to the log file. The optional *context* is a descriptive tag that will be prepended to the message within the log file (defaulting to "TANGELO" if omitted). Common context tags used internally in Tangelo include "TANGELO" (to describe startup/shutdown activities), and "ENGINE" (which describes actions being taken by CherryPy). This function may be useful for debugging or otherwise tracking a service's activities as it runs. The optional logging level `lvl` is one of the python logging constants. By default, `logging.INFO` is used.

Generally you should use one of the variants of this function listed below, but if you want to write a logging message in the terminal's default color, you can use this function, specifying the log level you need.

`tangelo.`**`log_debug`**`(`[*context*]`, `*msg*`)`

Variant of *`tangelo.log()`* that writes out messages in blue, at level `logging.DEBUG`. These messages can be used to diagnose, e.g., plugins or services in development. By default, these messages are hidden - you can increase the verbosity to see them.

`tangelo.`**`log_info`**`(`[*context*]`, `*msg*`)`

Variant of *`tangelo.log()`* that writes out messages in purple at level `logging.INFO`. Informational messages describe what the system is doing at the moment. For example, when a plugin is about to perform initialization, a call like `tangelo.log_info("FOOBAR", "About to initialize...")` may be appropriate.

`tangelo.`**`log_warning`**`(`[*context*]`, `*msg*`)`

Variant of *`tangelo.log()`* that writes out messages in yellow at level `logging.WARNING`. Warnings are messages indicating that something did not work out as expected, but the requested action will still be accomplished (perhaps differently than was expected by the user). For example, if a plugin needs to fall back on a default method for doing something because the requested method was not available, this might be reported by a warning message.

`tangelo.`**`log_error`**`(`[*context*]`, *msg*)`
Variant of *tangelo.log()* that writes out messages in red at level `logging.ERROR`. Errors describe instances when Tangelo or a plugin fails to perform a requested task, without compromising Tangelo's ability to continue running. For example, if a plugin requires a file that is missing, it might report an error condition using this function.

`tangelo.`**`log_critical`**`(`[*context*]`, *msg*)`
Variant of *tangelo.log()* that writes out messages in bright, bold red at level `logging.CRITICAL`. Critical errors describe conditions that immediately prevent the further functioning of the system, generally leading to Tangelo halting. Generally, you will not need to call this function.

5.1.2 HTTP Interaction

`tangelo.`**`content_type`**`(`[*type*]`)`
Returns the content type for the current request, as a string. If *type* is specified, also sets the content type to the specified string.

`tangelo.`**`http_status`**`(*code*`[`, *message*`]`)
Sets the HTTP status code for the current request's response. *code* should be an integer; optional *message* can give a concise description of the code. Omitting it results in a standard message; for instance, `tangelo.http_status(404)` will send back a status of `404 Not Found`.

This function can be called before returning, e.g., a `dict` describing in detail what went wrong. Then, the response will indicate the general error while the body contains error details, which may be informational for the client, or useful for debugging.

`tangelo.`**`header`**`(*header_name*`[`, *new_value*`]`)
Returns the value associated to *header_name* in the HTTP headers, or `None` if the header is not present.

If *new_value* is supplied, the header value will additionally be replaced by that value.

`tangelo.`**`request_header`**`(*header_name*)`
Returns the value associated to *header_name* in the request headers, or `None` if the header is not present.

`tangelo.`**`request_path`**`()`
Returns the path of the current request. This is generally the sequence of path components following the domain and port number in a URL.

`tangelo.`**`request_body`**`()`
Returns a filelike object that streams out the body of the current request. This can be useful, e.g., for retrieving data submitted in the body for a POST request.

`tangelo.`**`session`**`(*key*`[`, *value*`]`)
Returns the value currently associated to the session key *key*, or *None* if there is no such key. If *value* is given, it will become newly associated to *key*.

`tangelo.`**`redirect`**`(*path*`[`, *status_code*`]`)
Used to signal the browser that a web service wants to perform an HTTP redirect to a different *path*. The optional *status_code* should be a value in the `3xx` range indicating the type of redirect desired; it defaults to `303`.

In the following example service,

```
import tangelo

def run():
    return tangelo.redirect("other/path/content.html")
```

Tangelo will direct the client to the URL shown, resulting in that file being served instead of the service itself.

`tangelo.`**`internal_redirect`**(*path*)
Used to signal the server to serve content from a different path in place of the current service; similar to *`tangelo.redirect()`* but without informing the client of the redirection.

The example above will look very similar using this function instead:

```
import tangelo

def run():
    return tangelo.internal_redirect("other/path/content.html")
```

When this internal redirection occurs, the browser's displayed URL, for example, will not change to reflect the requested path.

`tangelo.`**`file`**(*path*[, *content_type="application/octet-stream"*])
Used to signal the server to serve content from the file at *path* as if it were being served statically. By using appropriate absolute or relative paths, this function can be used to serve content from a Tangelo service that is not otherwise available from Tangelo's web root. It can be used as follows:

```
import tangelo

def run():
    return tangelo.file("/some/crazy/path/to/content.txt", content_type="text/plain")
```

5.1.3 Web Services Utilities

`tangelo.`**`paths`**(*paths*)
Augments the Python system path with the list of web directories specified in `paths`. Each path must be **within the web root directory** or **within a user's web home directory** (i.e., the paths must be legal with respect to `tangelo.legal_path()`).

This function can be used to let web services access commonly used functions that are implemented in their own Python modules somewhere in the web filesystem.

After a service calling this function returns, the system path will be restored to its original state. This requires calling `tangelo.paths()` in every function wishing to change the path, but prevents shadowing of expected locations by modules with the same name in other directories, and the uncontrolled growth of the `sys.path` variable.

`tangelo.`**`config`**()
Returns a copy of the service configuration dictionary (see *Configuring Web Services*).

`@tangelo.`**`restful`**
Marks a function in a Python service file as being part of that service's RESTful API. This prevents accidental exposure of unmarked support functions as part of the API, and also enables the use of arbitrary words as REST verbs (so long as those words are also valid Python function names). An example usage might look like the following, which uses a both a standard verb ("GET") and a custom one ("NORMALIZE").

```
import tangelo

@tangelo.restful
def get(foo, bar, baz=None):
    pass

@tangelo.restful
def normalize():
    pass
```

Note that Tangelo automatically converts the verb used by the web client to all lowercase letters before searching the Python module for a matching function to call.

`@tangelo.types`(*arg1=type1, ..., argN=typeN*)

Decorates a service by converting it from a function of several string arguments to a function taking typed arguments. Each argument to `tangelo.types()` is a function that converts strings to some other type - the standard Python functions `int()`, `float()`, and `json.loads()` are good examples. The functions are passed in as keyword arguments, with the keyword naming an argument in the decorated function. For example, the following code snippet

```
import tangelo

def stringfunc(a, b):
    return a + b

@tangelo.types(a=int, b=int)
def intfunc(a, b):
    return a + b

print stringfunc("3", "4")
print intfunc("3", "4")
```

will print:

```
34
7
```

`stringfunc()` performs string concatentation, while `intfunc()` performs addition on strings that have been converted to integers.

Though the names of the built-in conversion functions make this decorator look like it accepts "types" as arguments, any function that maps strings to any type can be used. For instance, a string representing the current time could be consumed by a function that parses the string and returns a Python `datetime` object, or, as mentioned above, `json.loads()` could be used to convert arbitrary JSON data into Python objects.

If an exception is raised by any of the conversion functions, its error message will be passed back to the client via a `tangelo.HTTPStatusCode` object.

`@tangelo.return_type`(*type*)

Similarly to how *`tangelo.types()`* works, this decorator can be used to provide a function to convert the return value of a service function to some other type or form. By default, return values are converted to JSON via the standard `json.dumps()` function. However, this may not be sufficient in certain cases. For example, the `bson.dumps()` is a function provided by PyMongo that can handle certain types of objects that `json.dumps()` cannot, such as `datetime` objects. In such a case, the service module can provide whatever functions it needs (e.g., by `import`ing an appropriate module or package) then naming the conversion function in this decorator.

5.2 Tangelo JavaScript Library

The Tangelo clientside library (*tangelo.js*) contains functions to help work with Tangelo, including basic support for creating web applications. These functions represent basic tasks that are widely useful in working with web applications; for advanced functionality and associated JavaScript/Python functions, see *Bundled Plugins*.

`tangelo.version`()

Return type **string** – the version string

Returns a string representing Tangelo's current version number. See *A Note on Version Numbers* for more information on Tangelo version numbers.

`tangelo.`**`ensurePlugin`**(*pluginName*)

Arguments

- **`string`** (*pluginName*) – The name of the plugin whose existence to ensure

Creates a Tangelo plugin namespace for *pluginName*, if it does not yet exist. If it already exists, this function does nothing.

This is the standard way to ensure that a plugin has been created. For instead, if `foobar.js` introduces the *foobar* clientside plugin, it may contain code like this:

```
tangelo.ensurePlugin("foobar");

var plugin = tangelo.plugin.foobar;

plugin.awesomeFunction = function () { ... };
plugin.greatConstant = ...;
```

As demonstrated here, once the existence of plugin *foobar* is guaranteed by the call to *`tangelo.ensurePlugin()`*, it can thereafter be referenced by the name `tangelo.plugin.foobar`.

`tangelo.`**`getPlugin`**(*pluginName*)

Deprecated since version 0.10: See *`tangelo.ensurePlugin()`* for the replacement.

Arguments

- **`string`** (*pluginName*) – The name of the plugin to retrieve

Return type **object** – the contents of the requested plugin

Returns an object containing the plugin contents for *pluginName*. If *pluginName* does not yet exist as a plugin, the function first creates it as an empty object.

This is a standard way to create and work with plugins. For instance, if `foobar.js` introduces the *foobar* clientside plugin, it may contain code like this:

```
var plugin = tangelo.getPlugin("foobar");

plugin.awesomeFunction = function () { ... };

plugin.greatConstant = ...
```

The contents of this example plugin would hereafter be accessible via `tangelo.plugin.foobar`.

`tangelo.`**`pluginUrl`**(*plugin*[, **pathComponents*])

Arguments

- **`api`** (*string*) – The name of the Tangelo plugin to construct a URL for.
- **`*pathComponents`** (*string*) – Any extra path components to be appended to the constructed URL.

Return type **string** – the URL corresponding to the requested plugin and path

Constructs and returns a URL for the named *plugin*, with optional trailing path components listed in the remaining arguments to the function.

For example, a call to `tangelo.pluginUrl("stream", "next", "a1b2c3d4e5")` will return the string `"/plugin/stream/next/a1b2c3d4e5"`. This function is useful for calls to, e.g., `$.ajax()` when engaging a Tangelo plugin.

`tangelo.`**`queryArguments`**`()`

Return type **object** – the query arguments as key-value pairs

Returns an object whose key-value pairs are the query arguments passed to the current web page.

This function may be useful to customize page content based on query arguments, or for restoring state based on configuration options, etc.

`tangelo.`**`absoluteUrl`**`(`*webpath*`)`

Arguments

- **`webpath`** (*string*) – an absolute or relative web path

Return type **string** – an absolute URL corresponding to the input webpath

Computes an absolute web path for *webpath* based on the current location. If *webpath* is already an absolute path, it is returned unchanged; if relative, the return value has the appropriate prefix computed and prepended.

For example, if called from a page residing at `/foo/bar/index.html`, `tangelo.absoluteUrl("../baz/qux/blah.html")` would yield `/foo/baz/qux/blah.html`, and `tangelo.absoluteUrl("/one/two/three")` would yield `/one/two/three`.

`tangelo.`**`accessor`**`(`[*spec*]`)`

Arguments

- **`object`** (*spec*) – The accessor specification

Return type **function** – the accessor function

Returns an *accessor function* that behaves according to the accessor specification *spec*. Accessor functions generally take as input a JavaScript object, and return some value that may or may not be related to that object. For instance, `tangelo.accessor({field: "mass"})` returns a function equivalent to:

```
function (d) {
    return d.mass;
}
```

while `tangelo.accessor({value: 47})` return a constant function that returns 47, regardless of its input.

As a special case, if *spec* is missing, or equal to the empty object `{}`, then the return value is the `undefined accessor`, which simply raises a fatal error when called.

For more information of the semantics of the *spec* argument, see *Accessor Specifications*.

5.3 Bundled Plugins

Tangelo ships with several bundled plugins that implement useful and powerful functionality, as well as providing examples of various tasks that plugins can perform. This page divides the set of bundled plugins into categories, demonstrating some of the styles of problems Tangelo can help solve.

5.3.1 Core Plugins

Although these "core plugins" are built using the same plugin system architecture available to any Tangelo user, these deliver services vital to any working Tangelo instance, and can therefore be considered integral parts of the Tangelo platform.

Tangelo

The Tangelo plugin simply serves the Tangelo clientside library files `tangelo.js` and `tangelo.min.js`. It also includes a "version" web service that simply returns, as plain text, the running server's version number.

This is supplied as a plugin to avoid having to include the JavaScript files manually into every deployment of Tangelo. Instead, the files can be easily served directly from the plugin, thus retaining stable URLs across deployments.

Manifest

File	Description
`/plugin/tangelo/tangelo.js`	Unminified Tangelo library
`/plugin/tangelo/tangelo.min.js`	Minified Tangelo library
`/plugin/tangelo/version`	Version reporting service

Docs

The Docs plugin serves the Tangelo documentation (the very documentation you are reading right now!). Again, this is to simplify deployments. The index is served at `/plugin/docs` and from there the index page links to all pages of the documentation.

Stream

It may be necessary to return an immense (or even infinite) amount of data from a web service to the client. However, this may take up so much time and memory that dealing with it becomes intractable. In such situations, the Stream plugin may be able to help.

Generators in Python

Essentially, the plugin works by exposing Python's abstraction of generators. If a web service module includes a `stream()` function that uses the `yield` keyword instead of `return`, thus marking it as a generator function, then the Stream plugin can use this module to launch a *streaming service*. Here is an example of such a service, in a hypothetical file named `prime-factors.py`:

```
import math
import tangelo

def prime(n):
    for i in xrange(2, int(math.floor(math.sqrt(num)+1))):
        if n % i == 0:
            return False
    return True

@tangelo.types(n=int)
def stream(n=2):
    for i in filter(prime, range(2, int(math.floor(math.sqrt(num)+1)))):
        if n % i == 0
            yield i
```

The `stream()` function returns a *generator object* - an object that returns a prime divisor of its argument once for each call to its `next()` method. When the code reaches its "end" (i.e., there are no more values to `yield`), the `next()` method raises a `StopIteration` exception.

In Python this object, and others that behave the same way, are known as *iterables*. Generators are valuable in particular because they generate values as they are requested, unlike e.g. a list, which always retains all of its values and therefore

has a larger memory footprint. In essence, a generator trades space for time, then amortizes the time over multiple calls to `next()`.

The Stream plugin leverages this idea to create *streaming services*. When a service module returns a generator object from its `stream()` function, the plugin logs the generator object in a table, associates a key to it, and sends this key as the response. For example, an ajax request to the streaming API, identifying the `prime-factors` service above, might yield the following response:

```
{"key": "3dffee9e03cef2322a2961266ebff104"}
```

From this point on, values can be retrieved from the newly created generator object by further engaging the streaming API.

The Stream REST API

The streaming API can be found at `/plugin/stream/stream`. The API is RESTful and uses the following verbs:

- `GET /plugin/stream/stream` returns a list of all active stream keys.
- `GET /plugin/stream/stream/<stream-key>` returns some information about the named stream.
- `POST /plugin/stream/stream/start/<path>/<to>/<streaming>/<service>` runs the `stream()` function found in the service, generates a hexadecimal key, and logs it in a table of streaming services, finally returning the key.
- `POST /api/stream/next/<stream-key>` calls `next()` on the associated generator and returns a JSON object with the following form:

```
{
    "finished": false,
    "data": <value>
}
```

  The `finished` field indicates whether `StopIteration` was thrown, while the `data` field contains the value `yield`ed from the generator object. If `finished` is `true`, there will be no `data` field, and the stream key for that stream will become invalid.

- `DELETE /api/stream/<stream-key>` makes the stream key invalid, removes the generator object from the stream table, and returns a response showing which key was removed:

```
{"key": "3dffee9e03cef2322a2961266ebff104"}
```

  This is meant to inform the client of which stream was deleted in the case where multiple deletions are in flight at once.

JavaScript Support for Streaming

`/plugin/stream/stream.js` defines a clientside `stream` plugin that offers a clean, callback-based JavaScript API to the streaming REST service:

`tangelo.plugin.stream.`**`streams`**(*callback*)

Arguments

- **`callback`** (*function(keys)*) – Callback invoked with the list of active stream keys

Asynchronously retrieves a JSON-encoded list of all stream keys, then invokes *callback*, passing the keys in as a JavaScript list of strings.

`tangelo.plugin.stream.`**`start`**(*webpath*, *callback*)

Arguments

- **webpath** (*string*) – A relative or absolute web path, naming a stream-initiating web service
- **callback** (*function(key)*) – A function to call when the key for the new stream becomes available

Asynchronously invokes the web service at *webpath* - which should initiate a stream by returning a Python iterable object from its *run()* method - then invokes *callback*, passing it the stream key associated with the new stream.

This callback might, for example, log the key with the application so that it can be used later, possibly via calls to *tangelo.plugin.stream.query()* or *tangelo.plugin.stream.run()*:

```
tangelo.plugin.stream.start("myservice", function (key) {
    app.key = key;
});
```

tangelo.plugin.stream.**query**(*key*, *callback*)

Arguments

- **key** (*string*) – The key for the desired stream
- **error) callback** (*function(data,)* – The callback to invoke when results come back from the stream

Runs the stream keyed by *key* for one step, then invokes *callback* with the result. If there is an error, *callback* is instead invoked passing `undefined` as the first argument, and the error as the second.

tangelo.plugin.stream.**run**(*key*, *callback*[, *delay=100*])

Arguments

- **key** (*string*) – The key for the stream to run
- **callback** (*function(data)*) – The callback to pass stream data when it becomes available
- **delay** (*number*) – The delay in milliseconds between the return from a callback invocation, and the next stream query

Runs the stream keyed by *key* continuously until it runs out, or there is an error, invoking *callback* with the results each time. The *delay* parameter expresses in milliseconds the interval between when a callback returns, and when the stream is queried again.

The behavior of *callback* can influence the future behavior of this function. If *callback* returns a value, and the value is a

•**function**, it will replace *callback* for the remainder of the stream queries;

•**boolean**, it will stop running the stream if `false`;

•**number**, it will become the new delay, beginning with the very next stream query.

•**object**, it will have the **function** effect above if there is a key `callback`; the **boolean** effect above if there is a key `continue`; the **number** effect above if there is a key `delay` (in other words, this allows for multiple effects to be declared at once).

Other return types will simply be ignored.

tangelo.plugin.stream.**delete**(*key*[, *callback*])

Arguments

- **key** (*string*) – The key of the stream to delete

- **callback** (*function(error)*) – A callback that is passed an error object if an error occurs during deletion.

Deletes the stream keyed by *key*. The optional *callback* is a function that is invoked with an error object is something went wrong during the delete operation, or no arguments if the delete was successful.

VTKWeb

The VTKWeb plugin is able to run VTK Web programs and display the result in real time on a webpage. The interface is somewhat experimental at the moment and only supports running the program and interacting with it via the mouse. In a later version, the ability to call functions and otherwise interact with VTK Web in a programmatic way will be added.

In order to enable this funcationality, the plugin must be configured with a `vtkpython` option set to the full path to a `vtkpython` executable in a build of VTK.

The VTK Web REST API

The VTK Web API is found at */plugin/vtkweb/vtkweb*. The API is RESTful and uses the following verbs:

- `POST /plugin/vtkweb/vtkweb/full/path/to/vtkweb/script.py` launches the named script (which must be given as an absolute path) and returns a JSON object similar to the following:

```
{
    "status": "complete",
    "url": "ws://localhost:8080/ws/d74a945ca7e3fe39629aa623149126bf/ws",
    "key": "d74a945ca7e3fe39629aa623149126bf"
}
```

  The `url` field contains a websocket endpoint that can be used to communicate with the VTK web process. There is a *vtkweb.js* file (included in the Tangelo installation) that can use this information to hook up an HTML viewport to interact with the program, though for use with Tangelo, it is much simpler to use the JavaScript VTK Web library functions to abstract these details away. The `key` field is, similarly to the streaming API, a hexadecimal string that identifies the process within Tangelo.

  In any case, receiving a response with a `status` field reading "complete" means that the process has started successfully.

- `GET /plugin/vtkweb/vtkweb` returns a list of keys for all active VTK Web processes.
- `GET /plugin/vtkweb/vtkweb/<key>` returns information about a particular VTK Web process. For example:

```
{
    "status": "complete",
    "process": "running",
    "port": 52446,
    "stderr": [],
    "stdout": [
        "2014-02-26 10:00:34-0500 [-] Starting factory <vtk.web.wamp.ReapingWampServerFactor
        "2014-02-26 10:00:34-0500 [-] ReapingWampServerFactory starting on 52446\n",
        "2014-02-26 10:00:34-0500 [-] Log opened.\n",
        "2014-02-26 10:00:34-0500 [VTKWebApp,0,127.0.0.1] Client has reconnected, cancelling
        "2014-02-26 10:00:34-0500 [VTKWebApp,0,127.0.0.1] on_connect: connection count = 1\n
    ]
}
```

The `status` field indicates that the request for information was successful, while the remaining fields give information about the running process. In particular, the `stderr` and `stdout` streams are queried for any lines of text they contain, and these are delivered as well. These can be useful for debugging purposes.

If a process has ended, the `process` field will read `terminated` and there will be an additional field `returncode` containing the exit code of the process.

- `DELETE /plugin/vtkweb/vtkweb/<key>` terminates the associated VTK process and returns a response containing the key:

```
{
    "status": "complete",
    "key": "d74a945ca7e3fe39629aa623149126bf"
}
```

As with the streaming `DELETE` action, the key is returned to help differentiate which deletion has completed, in case multiple `DELETE` requests are in flight at the same time.

JavaScript Support for VTK Web

As with the Stream plugin's JavaScript functions, `/plugin/vtkweb/vtkweb.js` defines a clientside plugin providing a clean, callback-based interface to the low-level REST API:

`tangelo.plugin.vtkweb.`**`processes`**(*callback*)

Arguments

- **`callback`** (*function(keys)*) – The callback to invoke when the list of keys becomes available

Asynchronously retrieves a list of VTK Web process keys, and invokes *callback* with the list.

`tangelo.plugin.vtkweb.`**`info`**(*key*, *callback*)

Arguments

- **`key`** (*string*) – The key for the requested VTK Web process
- **`callback`** (*function(object)*) – The callback to invoke when the info report becomes available

Retrieves a status report about the VTK Web process keyed by *key*, then invokes *callback* with it when it becomes available.

The report is a JavaScript object containing a `status` field indicating whether the request succeeded ("complete") or not ("failed"). If the status is "failed", the `reason` field will explain why.

A successful report will contain a `process` field that reads either "running" or "terminated". For a terminated process, the `returncode` field will contain the exit code of the process.

For running processes, there are additional fields: `port`, reporting the port number the process is running on, and `stdout` and `stderr`, which contain a list of lines coming from those two output streams.

This function may be useful for debugging an errant VTK Web script.

`tangelo.plugin.vtkweb.`**`launch`**(*cfg*)

Arguments

- **`cfg.url`** (*string*) – A relative or absolute web path referring to a VTK Web script
- **`cfg.argstring`** (*string*) – A string containing command line arguments to pass to the launcher script

- **cfg.viewport** (*string*) – A CSS selector for the `div` element to serve as the graphics viewport for the running process
- **cfg.callback** (*function(key,error)*) – A callback that reports the key of the new process, or the error that occured

Attempts to launch a new VTK Web process by running a Python script found at *cfg.url*, passing *cfg.argstring* as commandline arguments to the launcher script. If successful, the streaming image output will be sent to the first DOM element matching the CSS selector given in *cfg.viewport*, which should generally be a `div` element.

After the launch attempt succeeds or fails, *callback* is invoked, passing the process key as the first argument, and the error object describing any errors that occurred as the second (or `undefined` if there was no error).

`tangelo.plugin.vtkweb.`**terminate**(*key*[, *callback*])

Arguments

- **key** (*string*) – The key of the process to terminate
- **callback** (*function(key,viewport,error)*) – A callback that will be invoked upon completion of the termination attempt

Attempts to terminate the VTK Web process keyed by *key*. If there is a *callback*, it will be invoked with the key of the terminated process, the DOM element that was the viewport for that process, and an error (if any). The key is passed to the callback in case this function is called several times at once, and you wish to distinguish between the termination of different processes. The DOM element is passed in case you wish to change something about the appearance of the element upon termination.

Girder

Girder is an open-source, high-performance data management platform. The Girder plugin mounts a working instance of Girder in the plugin namespace so that its web client and REST API become available for use with Tangelo web applications.

When the plugin is loaded, `/plugin/girder/girder` will serve out the web frontend to Girder, while `/plugin/girder/girder/api/v1` will point to the REST API documentation, as well as serving as the base URL for all API calls to Girder.

For more information about how to use Girder, see its documentation.

5.3.2 Utilities

These plugins do not represent core, substantive functionality, but rather utility functions that significantly ease the process of creating web applications.

Config

Many web applications need to change their behavior depending on external resources or other factors. For instance, if an application makes use of a Mongo database, a particular deployment of that application may wish to specify just which database to use. To this end, the Config plugin works to provide a simple way to configure the runtime behavior of applications, by using a file containing a JSON object as a key-value store representing the configuration.

The plugin provides a web service at `/plugin/config/config` that simply parses a JSON file and returns a JSON object representing the contents of the file. The API is as follows:

- `GET /plugin/config/config/<absolute>/<webpath>/<to>/<json>/<file>[?required]`

If the path specified does not point to a static file, or does not contain a valid JSON object, the call will result in an HTTP 4xx error, with the body expressing the particular reason for the error in a JSON response. Otherwise, the service will parse the file and return the configuration object in the "result" field of the response.

If the file does not exist, then the behavior of the service depends on the presence of absence of the `required` parameter: when the call is made *with* the parameter, this results in a 404 error; otherwise, the service returns an empty object. This is meant to express the use case where an application *can* use a configuration file if specified, falling back on defaults if there is none.

The plugin also supplies a JavaScript plugin via `/plugin/config/config.js`; like other JavaScript plugin components, it provides a callback-based function that engages the service on the user's behalf:

`tangelo.plugin.config.`**`config`**(*url*[, *required*], *callback*)

Arguments

- **`string`** (*url*) – An absolute or relative URL to the configuration file
- **`boolean`** (*required*) – A flag indicating whether the configuration file is required or not (default: `false`)
- **`function`** (*callback(config)*) – Callback invoked with configuration data when it becomes available

Engages the config service using the file specified by *url*, invoking *callback* with the configuration when it becomes available. The optional *required* flag, if set to `true`, causes *callback* to be invoked with `undefined` when the configuration file doesn't exist; when set to `false` or not supplied, a non-existent configuration file results in *callback* being invoked with `{}`.

UI

The UI plugin contains some JQuery plugins useful for building a user interface as part of a web application.

`$.`**`controlPanel`**`()`

Constructs a control panel drawer from a `<div>` element. The div can contain any standard HTML content; when this plugin is invoked on it, it becomes a sliding drawer with a clickable handle that will disappear into the bottom of the window when closed.

This plugin can be used to maintain, e.g., visualization settings that affect what is seen in the main window.

`$.`**`svgColorLegend`**(*cfg*)

Arguments

- **`cfg.legend`** (*string*) – CSS selector for SVG group element that will contain the legend
- **`cfg.cmap_func`** (*function*) – A colormapping function to create color patches for the legend entries
- **`cfg.xoffset`** (*integer*) – How far, in pixels, to set the legend from the left edge of the parent SVG element.
- **`cfg.yoffset`** (*integer*) – How far, in pixels, to set the legend from the top edge of the parent SVG element.
- **`cfg.categories`** (*string[]*) – A list of strings naming the categories represented in the legend.
- **`cfg.height_padding`** (*integer*) – How much space, in pixels, to place between legend entries.
- **`cfg.width_padding`** (*integer*) – How much space, in pixels, to place between a color patch and its associated label

- **cfg.text_spacing** (*integer*) – How far, in pixels, to raise text labels (used to vertically center text within the vertical space occupied by a color patch).
- **cfg.legend_margins** (*object*) – An object with (optional) fields *top*, *bottom*, *left*, and *right*, specifying how much space, in pixels, to leave between the edge of the legend and the entries.
- **cfg.clear** (*bool*) – Whether to clear out the previous contents of the element selected by *cfg.legend*.

Constructs an SVG color legend in the `g` element specified by *cfg.legend*, mapping colors from the elements of *cfg.categories* through the function *cfg.cmap_func*.

Watch

The watch plugin monitors python files and will reload those files when they or any of their imported modules change based on file timestamps. In addition to adding this plugin in the list of plugins, it can be enabled with the Tangelo command-line option `--watch`, in which case it is the first plugin loaded.

Any module import *after* the watch plugin has been enabled will be monitored for changes, including library-based modules (but not built-in python modules).

When a Tangelo service is called, the plugin checks if that service script or any module it depends on has been changed. All modules that depend on the changed module will be reloaded.

Reloading modules is done via Python's `reload()` function. The module's dictionary of global variables is retained during a reload. Redefinitions override old definitions. This feature can be used to cache values between reloads. For instance:

```
try:
    cache
except NameError:
    cache = {}
```

The file times of python files are used to determine when a module has changed. For python files that are compiled or optimized into .pyc or .pyo files, if the uncompiled file exists (the .py file), then its time is used.

5.3.3 Data Management and Processing

To perform visualization, at some point it is necessary to deal with raw data. These plugins provide ways of storing, accessing, and tranforming data for use in your application.

Data

These functions provide transformations of common data formats into a common format usable by Tangelo plugins.

`tangelo.plugin.data.`**`tree`**(*spec*)

Converts an array of nodes with ids and child lists into a nested tree structure. The nested tree format with a standard *children* attribute is the required format for other Tangelo functions such as *`$.dendrogram()`*.

As an example, evaluating:

```
    var tree = tangelo.plugin.data.tree({
        data: [
            {name: "a", childNodes: [{child: "b", child: "c"}]},
            {name: "b", childNodes: [{child: "d"}]},
            {name: "c"},
```

```
            {name: "d"}
        ],
        id: {field: "name"},
        idChild: {field: "child"},
        children: {field: "childNodes"}
    });
```

will return the following nested tree (note that the original *childNodes* attributes will also remain intact):

```
    {
        name: "a",
        children: [
            {
                name: "b",
                children: [
                    {
                        name: "d"
                    }
                ]
            },
            {
                name: "c"
            }
        ]
    }
```

Arguments

- **spec.data** (*object*) – The array of nodes.
- **spec.id** (*Accessor*) – An accessor for the ID of each node in the tree.
- **spec.idChild** (*Accessor*) – An accessor for the ID of the elements of the children array.
- **spec.children** (*Accessor*) – An accessor to retrieve the array of children for a node.

tangelo.plugin.data.**distanceCluster**(*spec*)

Arguments

- **spec.data** (*object*) – The array of nodes.
- **spec.clusterDistance** (*number*) – The radius of each cluster.
- **spec.x** (*Accessor*) – An accessor to the x-coordinate of a node.
- **spec.y** (*Accessor*) – An accessor to the y-coordinate of a node.
- **spec.metric** (*function*) – A function that returns the distance between two nodes provided as arguments.

Groups an array of nodes together into clusters based on distance according to some metric. By default, the 2D Euclidean distance, $d(a, b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$, will be used. One can override the accessors to the x and y-coordinates of the nodes via the *spec* object. The algorithm supports arbitrary topologies with the presence of a custom metric. If a custom metric is provided, the *x*/*y* accessors are ignored.

For each node, the algorithm searches for a cluster with a distance *spec.clusterDistance*. If such a cluster exists, the node is added otherwise a new cluster is created centered at the node. As implemented, it runs in $\mathcal{O}(nN)$ time for n nodes and N clusters. If the cluster distance provided is negative, then the algorithm will be skipped and all nodes will be placed in their own cluster group.

The data array itself is mutated so that each node will contain a *cluster* property set to an array containing all nodes in the local cluster. For example, with clustering distance 5 the following data array

```
>>> data
[
    { x: 0, y: 0 },
    { x: 1, y: 0 },
    { x: 10, y: 0 }
]
```

will become

```
>>> data
[
    { x: 0, y: 0, cluster: c1 },
    { x: 1, y: 0, cluster: c1 },
    { x: 10, y: 0, cluster: c2 }
]
```

with

```
>>> c1
[ data[0], data[1] ]
>>> c2
[ data[2] ]
```

In addition, the function returns an object with properties *singlets* and *clusters* containing an array of nodes in their own cluster and an array of all cluster with more than one node, respectively. As in the previous example,

```
>>> tangelo.plugin.data.distanceCluster( { data: data, clusterDistance: 5 } )
{
    singlets: [ data[2] ],
    clusters: [ [ data[0], data[1] ] ]
}
```

`tangelo.plugin.data.`**`smooth`**(*spec*)

Arguments

- **`spec.data`** (*object*) – An array of data objects.
- **`spec.x`** (*Accessor*) – An accessor to the independent variable.
- **`spec.y`** (*Accessor*) – An accessor to the dependent variable.
- **`spec.set`** (*function*) – A function to set the dependent variable of a data object.
- **`spec.kernel`** (*string*) – A string denoting a predefined kernel or a function computing a custom kernel.
- **`spec.radius`** (*number*) – The radius of the convolution.
- **`spec.absolute`** (*bool*) – Whether the radius is given in absolute coordinates or relative to the data extent.
- **`spec.sorted`** (*bool*) – Whether the data is presorted by independent variable, if not the data will be sorted internally.
- **`spec.normalize`** (*bool*) – Whether or not to normalize the kernel to 1.

Performs 1-D smoothing on a dataset by convolution with a kernel function. The mathematical operation performed is as follows:

$$y_i \leftarrow \sum_{|x_i - x_j| < R} K(x_i, x_j)\, y_j$$

for $R =$ **spec.radius** and $K =$ **spec.kernel**. Predefined kernels can be specified as strings, these include:

- *box*: simple moving average (default),
- *gaussian*: gaussian with standard deviation **spec.radius**/3.

The function returns an array of numbers representing the smoothed dependent variables. In addition if **spec.set** was given, the input data object is modified as well. The set method is called after smoothing as follows:

```
set.call(data, y(data[i]), data[i], i),
```

and the kernel is called as:

```
kernel.call(data, x(data[i]), x(data[j]), i, j).
```

The default options called by

```
smooth({ data: data })
```

will perform a simple moving average of the data over a window that is of radius 0.05 times the data extent. A more advanced example

```
smooth({
    data: data,
    kernel: 'gaussian',
    radius: 3,
    absolute: true,
    sorted: false
})
```

will sort the input data and perform a gaussian smooth with standard deviation equal to 1.

`tangelo.plugin.data.`**`bin`**`(`*`spec`*`)`

Arguments

- **`spec.data`** (*object*) – An array of data objects.
- **`spec.value`** (*Accessor*) – An accessor to the value of a data object.
- **`spec.nBins`** (*integer*) – The number of bins to create (default 25).
- **`spec.min`** (*number*) – The minimum bin value (default data minimum).
- **`spec.max`** (*number*) – The maximum bin value (default data maximum).
- **`spec.bins`** (*object*) – User defined bins to aggregate the data into.

Aggregates an array of data objects into a set of bins that can be used to draw a histogram. The bin objects returned by this method look as follows:

```
{
    "min": 0,
    "max": 1,
    "count": 5
}
```

A data object is counted as inside the bin if its value is in the half open interval `[ min, max )`; however for the right most bin, values equal to the maximum are also included. The default behavior of this method is two construct a new array of equally spaced bins between data's minimum value and the data's maximum value. If `spec.bins` is given, then the data is aggregated into these bins rather than a new set being generated. In this case, the bin objects are mutated rather a new array being created. In addition, the counters are **not** reset to 0, so the user must do so manually if the bins are reused over multiple calls.

Examples:

```
>>> tangelo.plugin.data.bin({
        data: [{"value": 0}, {"value": 1}, {"value": 2}],
        nBins: 2
    })
[
    {"min": 0, "max": 1, "count": 1},
    {"min": 1, "max": 2, "count": 2}
]

>>> tangelo.plugin.data.bin({
        data: [{"value": 1}, {"value": 3}],
        nBins: 2,
        min: 0,
        max: 4
    })
[
    {"min": 0, "max": 2, "count": 1},
    {"min": 2, "max": 4, "count": 1}
]

>>> tangelo.plugin.data.bin({
        data: [{"value": 1}, {"value": 3}],
        bins: [{"min": 0, "max": 2, "count": 1}, {"min": 2, "max": 10, "count": 0}]
    })
[
    {"min": 0, "max": 2, "count": 2},
    {"min": 2, "max": 10, "count": 1}
]
```

Mongo

This plugin provides a service that connects to a Mongo database and retrieves results based on the requested query. The API looks as follows:

- `GET /plugin/mongo/mongo/<hostname>/<database>/<collection>?query=<query-string>&limit=`

query-string should be a JSON string describing a query object, while *field-string* should be a JSON string describing a list of fields to include in the results.

The service returns a JSON-encoded list of results from the database.

This plugin is under development, so the interface may change in the future in order to provide a more complete API.

5.3.4 Visualization

In order to help create vibrant visualization applications, the following plugins provide various services and widgets to visualize different kinds of data. These are meant to also offer a guideline on creating new visualization plugins as new data types and applications arise.

Vis

The Vis plugin provides several JQuery widgets for visualization particular types of data using basic chart types.

Dendrogram. `/plugin/vis/dendrogram.js` provides the following JQuery widget:

`$.dendrogram(spec)`

Arguments

- **spec.data** (*object*) – A nested tree object where child nodes are stored in the *children* attribute.
- **spec.label** (*accessor*) – The accessor for displaying tree node labels.
- **spec.id** (*accessor*) – The accessor for the node ID.
- **spec.nodeColor** (*accessor*) – The accessor for the color of the nodes.
- **spec.labelSize** (*accessor*) – The accessor for the font size of the labels.
- **spec.lineWidth** (*accessor*) – The accessor for the stroke width of the node links.
- **spec.lineColor** (*accessor*) – The accessor for the stroke color of the node links.
- **spec.nodeSize** (*accessor*) – The accessor for the radius of the nodes.
- **spec.labelPosition** (*accessor*) – The accessor for the label position relative to the node. Valid return values are *'above'* and *'below'*.
- **spec.expanded** (*accessor*) – The accessor to a boolean value that determines whether the given node is expanded or not.
- **spec.lineStyle** (*string*) – The node link style: *'curved'* or *'axisAligned'*.
- **spec.orientation** (*string*) – The graph orientation: *'vertical'* or *'horizontal'*.
- **spec.duration** (*number*) – The transition animation duration.
- **spec.on** (*object*) – An object of event handlers. The handler receives the data element as an argument and the dom node as *this*. If the function returns *true*, the default action is perfomed after the handler, otherwise it is prevented. Currently, only the *'click'* event handler is exposed.

Constructs an interactive dendrogram.

resize()
Temporarily turns transitions off and resizes the dendrogram. Should be called whenever the containing dom element changes size.

Correlation Plot. `/plugin/vis/correlationPlot.js` provides this widget:

`$.correlationPlot(spec)`
Constructs a grid of scatter plots that are designed to show the relationship between different variables or properties in a dataset.

Arguments

- **spec.variables** (*object[]*) – An array of functions representing variables or properties of the dataset. Each of these functions takes a data element as an argument and returns a number between 0 and 1. In addition, the functions should have a *label* attribute whose value is the string used for the axis labels.
- **spec.data** (*object[]*) – An array of data elements that will be plotted.
- **spec.color** (*accessor*) – An accessor for the color of each marker.

- **spec.full** (*bool*) – Whether to show a full plot layout or not. See the images below for an example. This value cannot currently be changed after the creation of the plot.

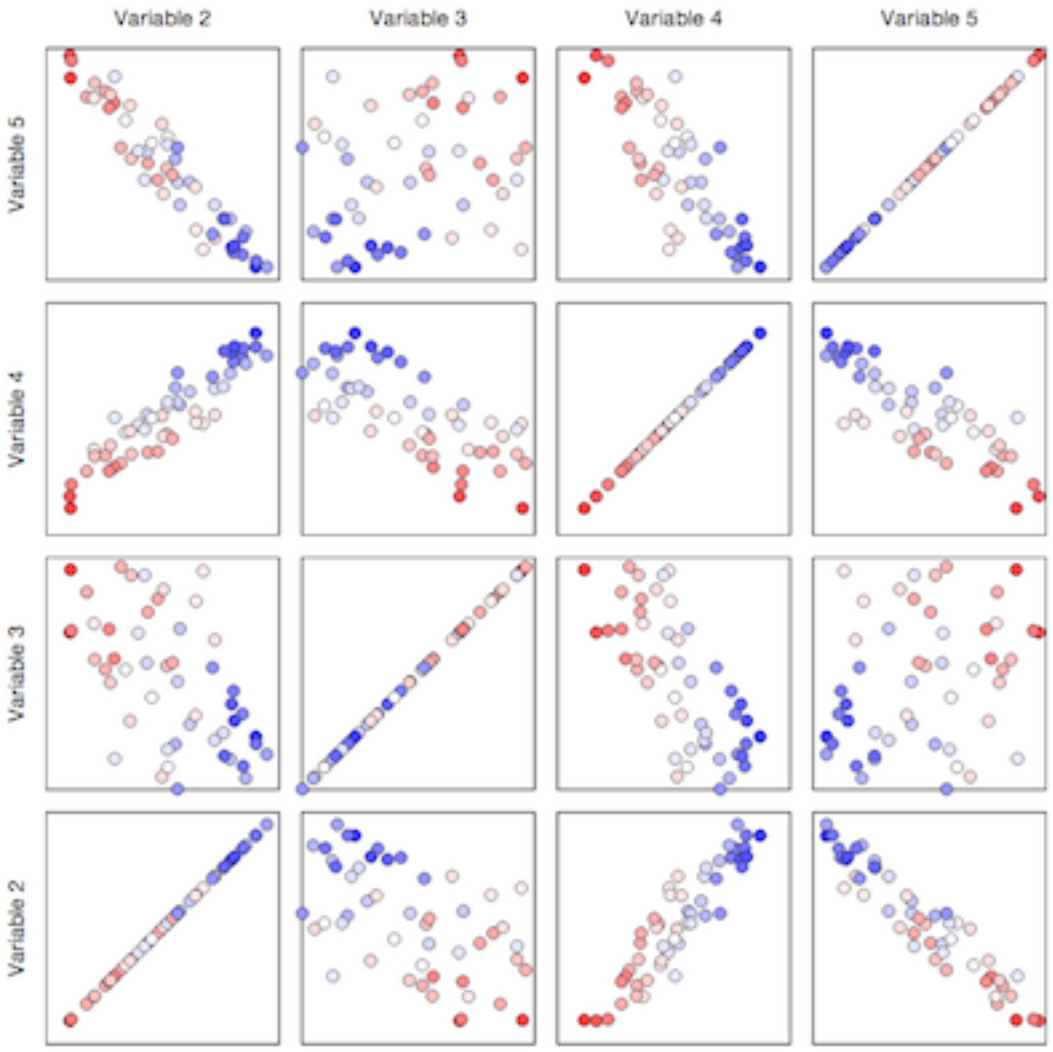


Fig. 5.1: An example of a full correlation plot layout. All variables are shown on the horizontal and vertical axes.

Timeline. `/plugin/vis/timeline.js` provides this widget:

`$.timeline`(*spec*)

Constructs a line plot with time on the x-axis and an arbitrary numerical value on the y-axis.

Arguments

- **spec.data** (*object[]*) – An array of data objects from which the timeline will be derived.
- **spec.x** (*accessor*) – An accessor for the time of the data.
- **spec.y** (*accessor*) – An accessor for the value of the data.
- **spec.transition** (*number*) – The duration of the transition animation in milliseconds, or false to turn off transitions.

xScale()

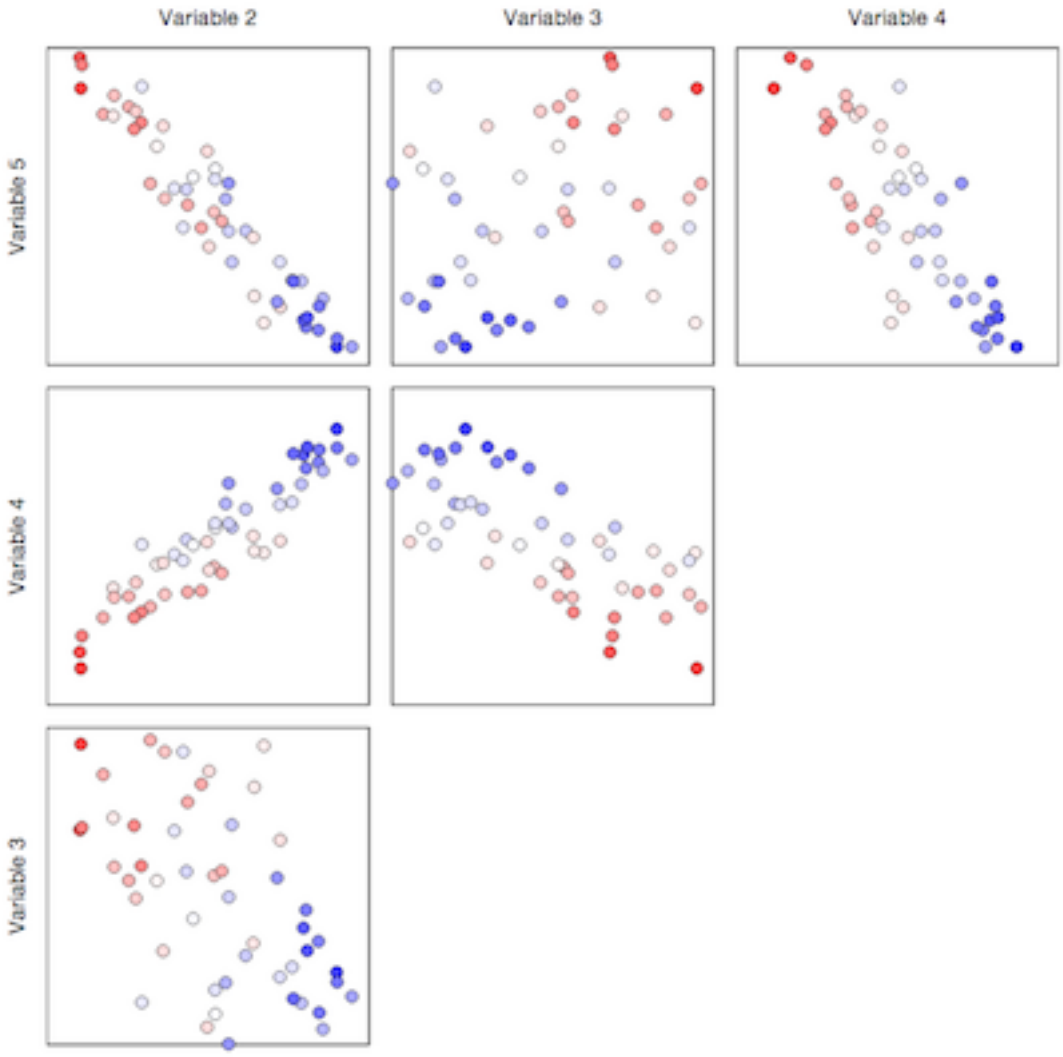
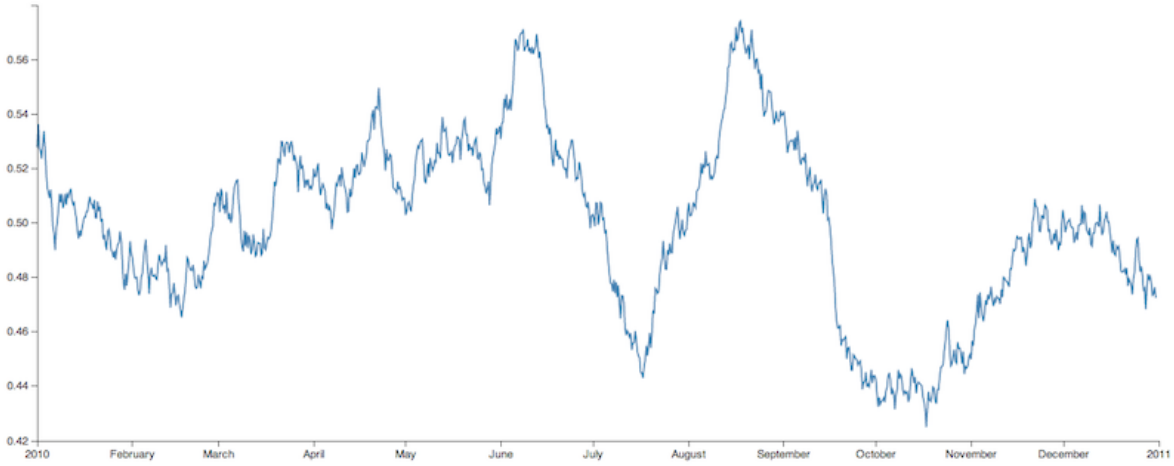


Fig. 5.2: An example of a half correlation plot layout. Only the upper left corner of the full layout are displayed.

yScale()

These return a d3 linear scale representing the transformation from plot coordinates to screen pixel coordinates. They make it possible to add custom annotations to the plot by appending an svg element to the *d3.select('.plot')* selection at the coordinates returned by the scales.



Node-link diagram. `/plugin/vis/nodelink.js` provides this widget:

`$.`**nodelink**(*spec*)

Arguments

- **spec.data** (*object*) – The node-link diagram data
- **spec.nodeSize** (*accessor*) – An accessor for the size of each node
- **spec.nodeColor** (*accessor*) – An accessor for the colormap category for each node
- **spec.nodeLabel** (*accessor*) – An accessor for each node's text label
- **spec.nodeCharge** (*accessor*) – An access for each node's simulated electrostatic charge
- **spec.linkSource** (*accessor*) – An accessor to derive the source node of each link
- **spec.linkTarget** (*accessor*) – An accessor to derive the target node of each link

Constructs an interactive node-link diagram. *spec.data* is an object with `nodes` and `links` fields, each of which is a list of objects. The `nodes` list objects specify the nodes' visual properties, while the `links` list simply specifies the nodes at the end of each link, as indices into the `nodes` list.

The accessors *spec.linkSource* and *spec.linkTarget* specify how to extract the source and target information from each link object, while *spec.nodeSize* and *spec.nodeColor* specify how to extract these visual properties from the node objects, much as in the *`$.geonodelink()`* plugin. *spec.nodeCharge* specifies the simulated electrostatic charge on the nodes, for purposes of running the interactive node placement algorithm (see the D3 documentation for more information). Finally, *spec.nodeLabel* is an accessor describing what, if any, text label should be attached to each node.

Mapping

In many cases, data has a geospatial component, for which some kind of map is a useful mode of visualization. The mapping plugin provides several options for visualization geolocation data, via the following JQuery widgets.

Geo dots. To plot location dots on a GeoJSON map, `/plugin/mapping/geodots.js` provides:

`$.`**geodots**(*spec*)

Arguments

- **spec.worldGeometry** (*string*) – A web path to a GeoJSON file
- **spec.latitude** (*accessor*) – An accessor for the latitude component
- **spec.longitude** (*accessor*) – An accessor for the longitude component
- **spec.size** (*accessor*) – An accessor for the size of each plotted circle
- **spec.color** (*accessor*) – An accessor for the colormap category for each plotted circle

Constructs a map from a GeoJSON specification, and plots colored SVG dots on it according to *spec.data*.

spec.worldGeometry is a web path referencing a GeoJSON file. *spec.data* is an array of JavaScript objects which may encode geodata attributes such as longitude and latitude, and visualization parameters such as size and color, while *spec.latitude*, *spec.longitude*, and *spec.size* are accessor specifications describing how to derive the respective values from the data objects. *spec.color* is an accessor deriving categorical values to put through a color mapping function.

For a demonstration of this plugin, see the geodots example.

Geo node-link diagram. To plot a node-link diagram on a GeoJSON map, `/plugin/mapping/geonodelink.js` provides:

`$.geonodelink(spec)`

Arguments

- **spec.data** (*object*) – The encoded node-link diagram to plot
- **spec.worldGeometry** (*string*) – A web path to a GeoJSON file
- **spec.nodeLatitude** (*accessor*) – An accessor for the latitude component of the nodes
- **spec.nodeLongitude** (*accessor*) – An accessor for the longitude component of the nodes
- **spec.nodeSize** (*accessor*) – An accessor for the size of each plotted circle
- **spec.nodeColor** (*accessor*) – An accessor for the colormap category for each plotted circle
- **spec.linkSource** (*accessor*) – An accessor to derive the source node of each link
- **spec.linkTarget** (*accessor*) – An accessor to derive the target node of each link

Constructs a map from a GeoJSON specification, and plots a node-link diagram on it according to *spec.data*. This plugin produces similar images as *$.geodots()* does.

spec.worldGeometry is a web path referencing a GeoJSON file.

spec.data is an object containing two fields: `nodes` and `links`. The `nodes` field contains an array of JavaScript objects of the exact same structure as the *spec.data* array passed to *$.geodots()*, encoding each node's location and visual properties.

The `links` field contains a list of objects, each encoding a single link by specifying its source and target node as an index into the `nodes` array. *spec.linkSource* and *spec.linkTarget* are accessors describing how to derive the source and target values from each of these objects.

The plugin draws a map with nodes plotted at their specified locations, with the specified links drawn as black lines between the appropriate nodes.

For a demonstration of this plugin, see the geonodelink example.

Map dots. To plot dots on a Google Map, `/plugin/mapping/mapdots.js` provides:

`$.mapdots`(*spec*)

This plugin performs the same job as *`$.geodots()`*, but plots the dots on an interactive Google Map rather than a GeoJSON map. To this end, there is no need for a "worldGeometry" argument, but the data format and other arguments remain the same.

For a demonstration of this plugin, see the mapdots example.

Arguments

- **spec.data** (*object[]*) – The list of dots to plot
- **spec.latitude** (*accessor*) – An accessor for the latitude component
- **spec.longitude** (*accessor*) – An accessor for the longitude component
- **spec.size** (*accessor*) – An accessor for the size of each plotted circle
- **spec.color** (*accessor*) – An accessor for the colormap category for each plotted circle

GeoJS Map. GeoJS is an open-source visualization-centric mapping library. Tangelo provides some JQuery plugins to replicate the above mapping use cases with GeoJS.

GeoJS map. To use a GeoJS map instance as a plugin, `/plugin/mapping/geojsMap.js` provides:

`$.geojsMap` (*spec*)

This plugin provides a low level interface to the GeoJS mapping library. For a simple example of using this plugin, see the geojsMap example.

Arguments

- **spec.zoom** (*integer*) – The initial zoom level of the map.

The widget also contains the following public methods for drawing on the map.

latlng2display (*points*)

Converts a point or points in latitude/longitude coordinates into screen pixel coordinates. This function

takes in either a *geo.latlng* object or an array of such objects. It always returns an array of objects with properties:

- *x* the horizontal pixel coordinate
- *y* the vertical pixel coordinate

Arguments

- **point** (*geo.latlng*) – The world coordinate(s) to be converted

display2latlng (*points*)
This is the inverse of *latlng2display* returning an array of *geo.latlng* objects.

Arguments

- **point** (*object*) – The world coordinate(s) to be converted

svg ()
Returns an svg DOM element contained in the geojs map. This element directly receives mouse events from the browser, so you can attach event handlers to svg elements as if the map were not present. You can call stopPropagation to customize user intaraction and to prevent mouse events from reaching the map.

map ()
Returns the geojs *map* object for advanced customization.

Users of this plugin should attach a handler to the *draw* event that recomputes the pixel coordinates and redraws the svg elements. The plugin will trigger this event whenever the map is panned, zoomed, or resized.

GeoJS dots. To plot dots on a GeoJS map, `/plugin/mapping/geojsdots.js` provides:

`$.`**geojsdots** (*spec*)

Arguments

- **spec.data** (*object[]*) – The list of dots to plot
- **spec.latitude** (*accessor*) – An accessor for the latitude component
- **spec.longitude** (*accessor*) – An accessor for the longitude component
- **spec.size** (*accessor*) – An accessor for the size of each plotted circle
- **spec.color** (*accessor*) – An accessor for the colormap category for each plotted circle

This plugin is similar to *`$.mapdots()`*, but plots the dots using the geojsMap plugin.

For a demonstration of this plugin, see the geojsdots example.

Bokeh

Bokeh is a Python plotting library that can display interactive graphics on the web. Tangelo provides seamless integration with Bokeh via the Bokeh plugin. This plugin provides a Python decorator for use with web service functions that invoke the Bokeh module to construct a visualization, and a JavaScript function to smoothly transition the results of such a service into a web application.

`@tangelo.plugin.bokeh.`**bokeh** (*plot_object*)

Parameters **plot_object** (*PlotObject*) – A Bokeh `PlotObject` instance representing the plot to be displayed

Return type dict – A Python `dict` containing a div and a script for embedded the plot in a webpage

This decorator transforms the output of a web service that computes a Bokeh plot to a form that can be handled by the browser. It works by converting the plot object into the web components necessary to render it. When the decorator is used, an ajax call to the service results in a `dict` of two fields: `script` and `div`. If the div is embedded in the DOM, and the script after it so that it executes, the plot will appear in the page.

Rather than perform the task of setting up the div and script manually, the following JQuery widget, found in `/plugin/bokeh/bokeh.js`, can help:

`$`**`.bokeh`**(*cfg*)

Arguments

- **`string`** (*cfg.url*) – The URL of a web service returning a PlotObject

When invoked on a DOM element, the URL is retrieved; the expected data should be in the format described by *`tangelo.plugin.bokeh.bokeh()`* above. The DOM element then receives both the div and script content returned by the service, causing the interactive Bokeh plot to begin running in the target DOM element.

An example application can be found at `/plugin/bokeh/examples/scatter/index.html`.

CHAPTER 6

Information for Developers

6.1 Coding Style Guidelines

This section concerns written code format in Tangelo, with the goal of clear, readable, and consistent code. The build process uses `jshint` and `jscs` to catch possible code and style errors. This document describes some of the coding practices employed to help make Tangelo more reliable.

6.1.1 Code Style Rules

Indentation

Indentation is used to provide visual cues about the syntactic scope containing particular line of code. Good indentation practice dramaticaly improves code readability.

Four-space indentation. Each additional indentation level shall be exactly four spaces.

Indentation policy. The following structures shall require incrementing the indentation level:

Statements belonging to any block.

Chained function calls:

```
obj.call1()
    .call2()
    .call3();
```

Properties in a literal object:

```
obj = {
    prop1: 10,
    prop2: 3
};
```

Curly bracket placement. The left curly bracket that introduces a new indentation level shall appear at the end of the line that uses it; the right curly bracket that delimits the indented statements shall appear on the line following the last indented statement, at the decremented indentation:

```
[some statement...] {
                    ^
    .
    .
    .
```

```
}
^
```

Naming

Use `camelCase` for visualization, property, method, and local variable names.

Curly brackets

JavaScript uses curly brackets to delimit blocks. Blocks are required by the language when functions are defined. They are also required for executing more than one statement within control flow constructs such as `if` and `while`. While the option exists *not* to use curly brackets for a single statement in such cases, that practice can lead to errors (when, e.g., a new feature requires the single statement to be replaced by several statements).

Always use blocks in control flow statements. Every use of control flow operators (`if`, `while`, `do`) shall use curly brackets for its associated statement block, even if only a single statement appears therein.

Space placement

Parentheses are required in several places in JavaScript. Proper space placement can help make such constructs more readable.

Keyword-condition separation. A single space shall appear in the following situations.

Between a control-flow operator and its parenthesized condition:

```
if (condition...) {
  ^
```

Between a parenthesized condition and its following curly bracket:

```
if (condition...) {
                  ^
```

Between a function argument list and its following curly bracket:

```
function foobar(x, y, z) {
                         ^
```

Between the `function` *keyword and the argument list, in anonymous functions:*

```
f = function (a, b) {
            ^
```

After every comma.

On either side of a binary operator:

```
x = 3 + 4;
     ^ ^
```

Extraneous spaces. The last character in any given line shall not be a space.

Blank lines. Blank lines should be used to set apart sequences of statements that logically belong together.

Chained if/else-if/else statements

A common programming pattern is to test a sequence of conditions, selecting a single action to take when one of them is satisfied. In JavaScript, this is accomplished with an `if` block followed by a number of `else if` blocks, followed by an `else` block. `try catch` blocks have a similar syntax.

Single line `else if`, `else`, **and** `catch`. The `else if`, `else`, and `catch` keyword phrases shall appear on a single line, with a right curly bracket on their left and a left curly bracket on their right:

```
if (condition) {
    action();
} else if {
    other_action();
} else {
    default_action();
}
```

`new Array` and `new Object`

The `new` keyword is problematic in JavaScript. If it is omitted by mistake, the code will run without error, but will not do the right thing. Furthermore, built in constructors like `Array` and `Object` can be reimplemented by other code.

Use `[]` **and** `{}`. All construction of arrays and objects shall use the literal `[]` and `{}` syntax. The sequence of statements `x = [];`, then `x.length = N;` shall replace `new Array(N)`.

6.1.2 Code structure

This section concerns the structure of functions and modules, how constructs at a larger scale than individual statements and expressions should be handled.

JSLint directives

JSLint reads two special comments appearing at the top of files it is working on. The first appears in the following form:

```
/*jslint browser: true */
```

and specifies options to JSLint. Because Tangelo is a web project, every JavaScript file should have the comment that appears above as the first line. The other recognized directive in the global name list:

```
/*globals d3, $, FileReader */
```

This directive prevents JSLint from complaining that the listed names are global variables, or undefined. It is meant for valid names, such as standard library objects or linked libraries used in the file.

Lexical scopes

JavaScript has only two scope levels: *global* and *function*. In particular, blocks following, e.g., `for` and `if` statements *do not introduce an inner scope*. Despite this fact, JavaScript allows for variables to be declared within such blocks, causing seasoned C and C++ programmers to assume something false about the lifetime of such variables.

Single `var` **declaration.** Every function shall contain a single `var` declaration as its first statement, which shall list every local variable used by that function, listed one per line.

```
function foobar(){
    var width,
        height,
        i;
    .
    .
    .
}
```

This declaration statement shall **not** include any initializers (this promotes clearer coding, as the "initializers" can be moved below the declaration, and each one can retain its own comment to explain the initialization).

Global variables. Global variables shall **not** be used, unless as a namespace-like container for variables and names that would otherwise have to be global. When such namespace-like containers are used in a JavaScript file, they shall appear in the JSLint global name specifier.

Strict Mode

JavaScript has a "strict mode" that disallows certain actions technically allowed by the language. These are such things as using variables before they are defined, etc. It can be enabled by including `"use strict";` as the first statement in any function:

```
function foobaz() {
    "use strict";

    .
    .
    .
}
```

Strict mode functions. All functions shall be written to use strict mode.

6.1.3 A note on `try...catch` blocks

JSLint complains if the exception name bound to a `catch` block is the same as the exception name bound to a previous `catch` block. This is due to an ambiguity in the ECMAScript standard regarding the semantics of `try...catch` blocks. Because using a unique exception name in each `catch` block just to avoid errors from JSLint seems to introduce just as much confusion as it avoids, the current practice is **not** to use unique exception names for each `catch` block.

Use `e` for exception name. `catch` blocks may all use the name `e` for the bound exception, to aid in scanning over similar messages in the JSLint output. **This rule is subject to change in the future.**

6.1.4 A note on *"eval is evil"*

JSLint claims that `eval` is evil. However, it is actually *dangerous*, and not evil. Accordingly, `eval` should be kept away from most JavaScript code. However, to take one example, one of Tangelo's main dependencies, Vega, makes use of compiler technology that generates JavaScript code. `eval`ing this code is reasonable and necessary in this instance.

eval is misunderstood. If a JavaScript file needs to make use of `eval`, it shall insert an `evil: true` directive into the JSLint options list. All other JavaScript files shall **not** make use of `eval`.

6.2 Creating Tangelo Releases

Tangelo is developed on GitHub using the Git Flow work style. The main development branch is named `develop`, while all commits on `master` correspond to tagged releases. Topic, hotfix, and release branches are all used as described in the discussion in the link above.

This page documents the careful steps to take in creating a new release, meaning that a new commit is made on `master`, and a package is uploaded to the Python Package Index.

6.2.1 Release Procedure

Suppose for the sake of example that the last release's version number is *1.1*. The following procedure will produce a new release of Tangelo:

1. Merge all topic branches to develop. Be sure that `develop` contains the code from which you wish to create the new release.

2. Create a release branch. A release branch needs to be created off of develop:

```
git checkout -b release-1.2 develop
```

Note that the version number mentioned in the branch name is the version number of the release being created.

3. Bump the version number. Edit the file `package.json` in the top level of the repository, updating the version number to *1.2.0*. Be sure to use the *major.minor.patch* format.

Also edit `js/tests/tangelo-version.js`, `tests/tangelo-version.py`, and `tests/commandline-version.py` to bump the version numbers there manually (in each file, the expected version string is contained in a variable named `expected`). This is done by hand to ensure that the version tests are deployed correctly for step 6 below.

Finally, edit the file `CHANGELOG.md` in the root of the codebase. Change the `[Unreleased] - [unreleased]` line near the top of the file to something like `[1.2.0] - 2014-12-18`. Look over the specified changes. Edit these if necessary, making sure that the list is up to date.

4. Build Tangelo. Issue the following commands to create a fresh build of Tangelo from scratch:

```
grunt clean:all
npm install
grunt
```

This should result in a virtual environment with a newly built Tangelo. Bumping the version number in the previous step means that Grunt should have also updated the version string in all parts of the code that require it.

5. Commit. Make a commit on the release branch containing the version number update:

```
git commit -am "Bumping version number for release"
```

then launch Tangelo with

```
./venv/bin/tangelo --examples
```

and visit http://localhost:8080/plugin/tangelo/version to verify the version number there. Finally, load up any of the examples that uses *tangelo.js* (e.g., http://localhost:8080/plugin/vis/examples/barchart), and, in the console, issue `tangelo.version()` to verify the clientside version number as well.

6. Run the tests.

Issue this command to verify that the client and server side tests all pass:

```
grunt test
```

If any tests fail, fix the root causes, making commits and retesting as you go. In particular, the tests regarding Tangelo version numbers will fail if the version number bump or build process did not work properly for any reason.

7. Merge into master. Switch to the `master` branch and merge the release branch into it:

```
git checkout master
git merge --no-ff release-1.2
```

Do not omit the `--no-ff` flag! You can use the default merge commit message.

If you run into merge conflicts, carefully fix them and conclude the merge, then make sure to run the tests again.

8. Tag the release. Create a tag for the release as follows:

```
git tag -a v1.2
```

Use a commit message like "Release v1.2". Be sure to push the tag so it becomes visible to GitHub:

```
git push --tags
```

9. Upload the package to PyPI. Unpack the built package file, and then use the `upload` option to `setup.py`:

```
cd sdist
tar xzvf tangelo-1.2.0.tar.gz
../venv/bin/python setup.py sdist upload
```

10. Merge into develop. The changes made on the release branch must be merged back into `develop` as well, so that development may continue there:

```
git checkout develop
git merge release-1.2
```

This is one of the few times you should not use the `--no-ff` flag. We want both `master` and `develop` to thread through the release branch to simplify the graph view of the release. After the next step, this leaves both `master` and `develop` one commit ahead of the same, prepared release branch point.

11. Bump the version number again. The version number on the `develop` branch needs to be changed again, to add a *-dev* suffix. In our example, the version number will now be *1.2.0-dev*. This entails editing `package.json` once more, as well as `js/tests/tangelo-version.js`, `tests/tangelo-version.py`, and `tests/commandline-version.py`.

Also edit `CHANGELOG.md` again, reproducing a skeleton of a new changes section, copying the following:

```
## [Unreleased] - [unreleased]
### Added

### Changed

### Deprecated

### Removed

### Fixed

### Security
```

This will allow developers to update the appropriate section easily whenever a topic branch is merged to `develop`.

12. Test again. Run the tests one more time, to verify that the version number bump happened correctly, and to catch anything weird that may have happened as well.

13. Commit. Commit the change so that `develop` is ready to go:

```
git commit -am "Bumping version number"
```

14. Push. Push both `develop` and `master` to origin to bring the local and remote branches up to date.

15. Update the documentation. Log into https://readthedocs.org, go to the Tangelo documentation panel, go to the "version" link, and activate the documentation for v1.2. Log out and verify that the new documentation appears at https://tangelo.readthedocs.org.

6.2.2 Summary

You now have

- a new Tangelo package on PyPI. Installing with `pip install tangelo` will install the new version to the system.
- a new, tagged commit on `master` that corresponds exactly to the new release, and the new package in PyPI. Anyone who checks this out and builds it will have the same Tangelo they would have if installing via `pip` as above.
- a new commit on `develop` representing a starting point for further development. Be sure to create topic branches off of `develop` to implement new features and bugfixes.
- documentation for the new version, live on Read The Docs.

6.3 Developing Visualizations

6.3.1 Creating jQuery Widgets

Tangelo visualizations can be implemented as jQuery widgets. They extend the base jQuery UI widget class, but otherwise do not need to depend on anything else from jQuery UI.

6.3.2 Visualization Options

Basic Options

- *data* - The data associated with the visualization, normally an array.
- *width*, *height* - The width and height of the visualization, in pixels. If omitted, the visualization should resize to fit the DOM element.

Visualization Mapping Options

The following options are optional, but if your visualization is able to map data element properties to visual attributes like size, color, and label, you should use this standard naming convention. If you have multiple sets of visual elements (such as nodes and links in a graph), prefix these attributes as appropriate (e.g. *nodeSize*, *nodeStrokeWidth*).

- *size* - The size of the visual element as a number of pixels. For example, if drawing a square for each data element, the squares should have sizes equal to the square-root of what the *size* option returns for each data element.
- *color* - The main color of the visual element, specified as a CSS color string.

- *symbol* - The symbol to use for the visual element. This should use D3's standard set of symbol names.
- *label* - The label for the visual element (a string).
- *stroke* - The color of the stroke (outline) of the visual element specified in pixels.
- *strokeWidth* - The width of the stroke of the visual element in pixels.
- *opacity* - The opacity of the entire visual element, as a number between 0 to 1.

6.3.3 Accessor Specifications

AccessorSpec

Each visual mapping should take an *AccessorSpec* for a value. Accessor specifications work much like *DataRef* specs do in Vega, though they also allow programmatic ways to generate arbitrary accessors and scales.

- `function (d) { ... }` - The most general purpose way to generate a visual mapping. The argument is the data element and the return value is the value for the visual property.
- `{value: v}` - Sets the visual property to the same constant value *v* for all data elements.
- `{index: true}` - Evaluates to the index of the data item within its array.
- `{field: "dot.separated.name"}` - Retrieves the specified field or subfield from the data element and passes it through the visualization's default scale for that visual property. Unlike Vega, fields from the original data do not need to be prefixed by `"data."`. The special field name `"."` refers to the entire data element.
- `{field: "dot.separated.name", scale: ScaleSpec}` - Overrides the default scale using a scale specification. Set *scale* to `tangelo.identity` to use a field directly as the visual property.
- `{}` - The *undefined accessor*. This is a function that, if called, throws an exception. The function also has a property `undefined` set to *true*. This is meant as a stand-in for the case when an accessor must be assigned but there is no clear choice for a default. It is also used when creating Tangelo jQuery widgets to mark a property as being an accessor. Calling *`tangelo.accessor()`* with no arguments also results in an undefined accessor being created and returned.

ScaleSpec

A scale specification defines how to map data properties to visual properties. For example, if you want to color your visual elements using a data field *continent* containing values such as North America, Europe, Asia, etc. you will need a scale that maps North America to `"blue"`, Europe to `"green"`, etc. Vega has a number of built-in named scales that together define the *ScaleSpec*. In Tangelo, a *ScaleSpec* may also be an arbitrary function.

CHAPTER 7

Indices and tables

- genindex
- search

Symbols

$.bokeh() ($ method), 64
$.controlPanel() ($ method), 49
$.correlationPlot() ($ method), 55
$.dendrogram() ($ method), 55
$.geodots() ($ method), 58
$.geojsMap() ($ method), 62
$.geojsdots() ($ method), 63
$.geonodelink() ($ method), 59
$.mapdots() ($ method), 61
$.nodelink() ($ method), 58
$.svgColorLegend() ($ method), 49
$.timeline() ($ method), 56

D

display2latlng() (built-in function), 63

L

latlng2display() (built-in function), 62

M

map() (built-in function), 63

R

resize() (built-in function), 55

S

svg() (built-in function), 63

T

tangelo.absoluteUrl() (tangelo method), 42
tangelo.accessor() (tangelo method), 42
tangelo.config() (built-in function), 39
tangelo.content_type() (built-in function), 38
tangelo.ensurePlugin() (tangelo method), 41
tangelo.file() (built-in function), 39
tangelo.getPlugin() (tangelo method), 41
tangelo.header() (built-in function), 38
tangelo.http_status() (built-in function), 38
tangelo.internal_redirect() (built-in function), 38
tangelo.log() (built-in function), 37
tangelo.log_critical() (built-in function), 38
tangelo.log_debug() (built-in function), 37
tangelo.log_error() (built-in function), 37
tangelo.log_info() (built-in function), 37
tangelo.log_warning() (built-in function), 37
tangelo.paths() (built-in function), 39
tangelo.plugin.bokeh.bokeh() (built-in function), 63
tangelo.plugin.config.config() (tangelo.plugin.config method), 49
tangelo.plugin.data.bin() (tangelo.plugin.data method), 53
tangelo.plugin.data.distanceCluster() (tangelo.plugin.data method), 51
tangelo.plugin.data.smooth() (tangelo.plugin.data method), 52
tangelo.plugin.data.tree() (tangelo.plugin.data method), 50
tangelo.plugin.stream.delete() (tangelo.plugin.stream method), 45
tangelo.plugin.stream.query() (tangelo.plugin.stream method), 45
tangelo.plugin.stream.run() (tangelo.plugin.stream method), 45
tangelo.plugin.stream.start() (tangelo.plugin.stream method), 44
tangelo.plugin.stream.streams() (tangelo.plugin.stream method), 44
tangelo.plugin.vtkweb.info() (tangelo.plugin.vtkweb method), 47
tangelo.plugin.vtkweb.launch() (tangelo.plugin.vtkweb method), 47
tangelo.plugin.vtkweb.processes() (tangelo.plugin.vtkweb method), 47
tangelo.plugin.vtkweb.terminate() (tangelo.plugin.vtkweb method), 48
tangelo.pluginUrl() (tangelo method), 41
tangelo.queryArguments() (tangelo method), 41
tangelo.redirect() (built-in function), 38
tangelo.request_body() (built-in function), 38
tangelo.request_header() (built-in function), 38

tangelo.request_path() (built-in function), 38
tangelo.restful() (built-in function), 39
tangelo.return_type() (built-in function), 40
tangelo.session() (built-in function), 38
tangelo.types() (built-in function), 40
tangelo.version() (tangelo method), 40

X

xScale() (built-in function), 56

Y

yScale() (built-in function), 56