Testing Apache Modules with Python and ctypes

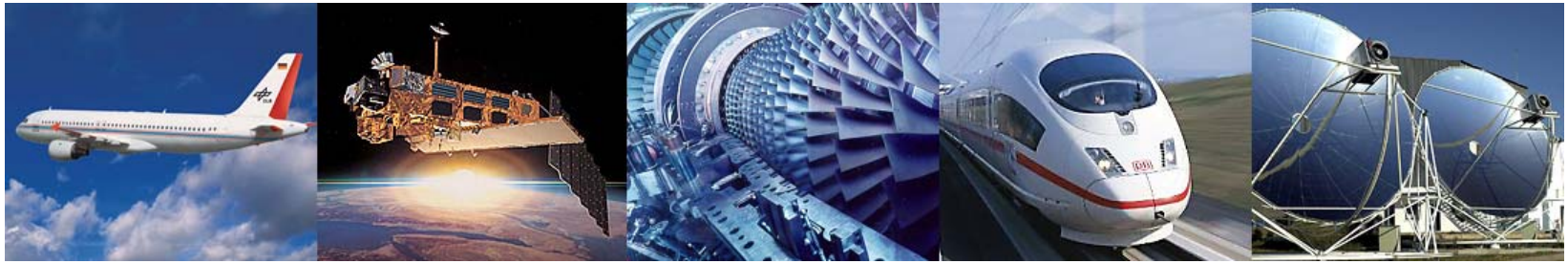
ApacheCon US 2009

Markus Litz - 06.11.2009

Agenda for today

- Why?
- Introduction to ctypes
- Preparing the apache
- Creating tests
- Demo

DLR
German Aerospace Center



- Research Institution
- Research Areas
  - Aeronautics
  - Space
  - Transport
  - Energy
- Space Agency

Locations and employees

6200 employees across
29 research institutes and
facilities at

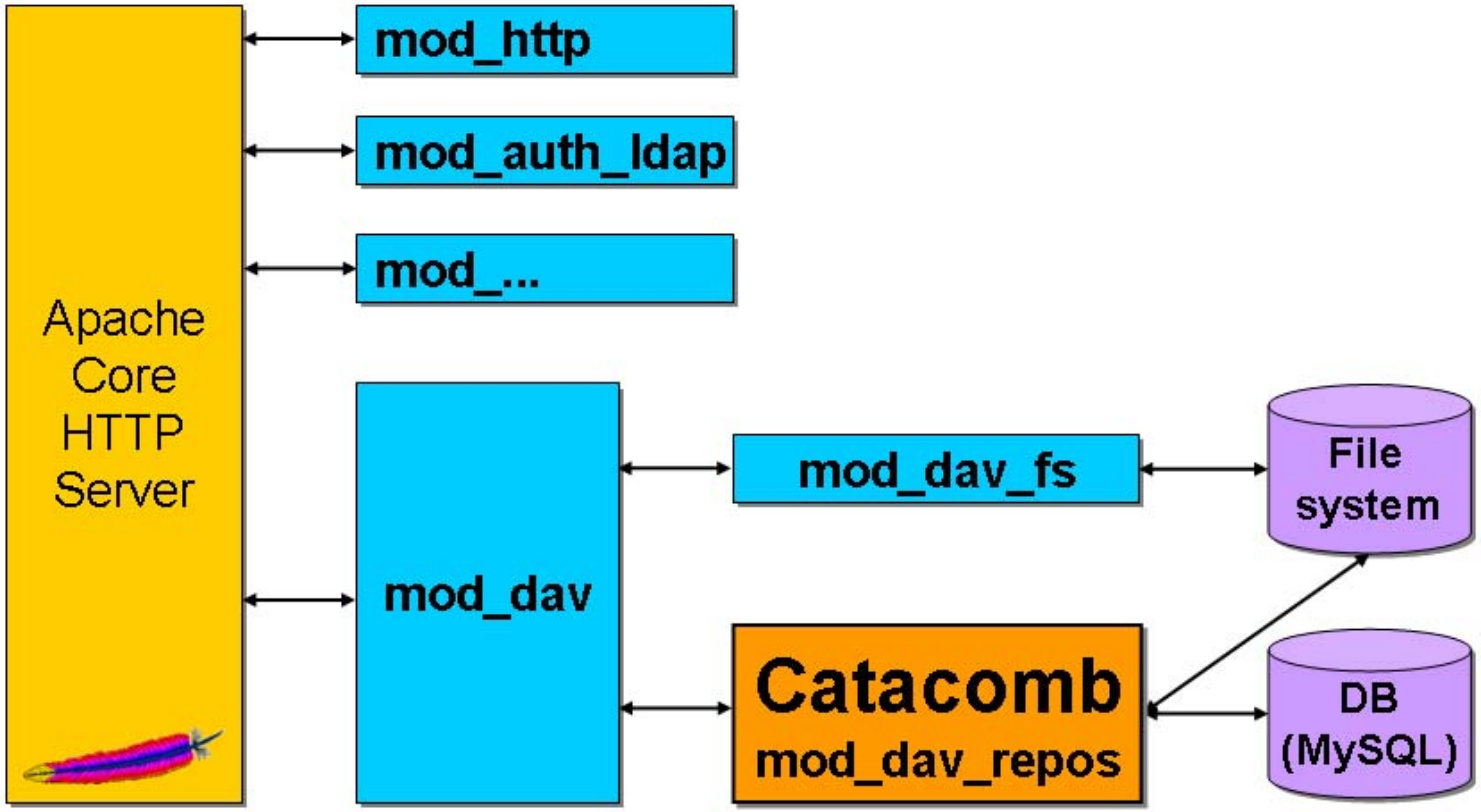
- 13 sites.

Offices in Brussels,
Paris and Washington.

Background

- DataFinder – a application for scientific data management
  - Storing and managing huge amounts of data
  - Search through the resource content and metadata
  - Various ways to store data, for example
    - ftp, network share, offline stores
  - Metadata management with the WebDAV protocol
    - Two supported WebDAV Server:
      - Tamino XML Server & Catacomb

Catacomb – A WebDAV Server Module for Apache



Catacomb – The Difference to mod_dav_fs

- Saving the resources
  - mod_dav_fs save content and properties in files on the filesystem
  - mod_dav_fs creates for every resource, and also for every collection, their own property file

- Consequence:
  - A single query of server side searching needs to open many files
  - Implementation of complex queries is difficult
  - Full text search is expensive

Catacomb – A WebDAV Server Module for Apache

- WebDAV repository module for mod_dav
- Catacomb uses relational databases to store the metadata
  - Strong search performance through SQL statements
- Catacomb is:
  - Good for Content management
  - Good for Collaborated web authoring
    - Support locks, avoid the “lost update” problem
  - Capable of searching (DASL) and versioning (DeltaV) resources

Catacomb – History and Current State

- Initial development at the University of California under the chair of Jim Whitehead
- Open Source project since 2002
- DeltaV and DASL implementation
- Since 2006 contribution of the DLR
  - ACP support
  - Database abstraction using mod_dbd
  - License changed to ASL2.0

Why testing your code?

- Development is faster and easier
- Code is more robust
- Code is more maintainable
- Code is more reliable

Why testing with Python and ctypes?

- Writing tests is easy
- No need to start an apache instance every time
- Tests could be automatically done with various Apache versions

What is ctypes

- ctypes is a wrapper for C-librarys for python
- ctypes allows to call functions in dlls/shared libraries from python code
- It is possible to implement C callback function
- Since Python 2.5.x, ctypes is in the standard library

How to use ctypes

- `from ctypes import *`
- Loading dynamic link libraries
  - `libc = cdll.msvcr`
  - `libc = CDLL("libc.so.6")`
- Calling functions
  - `print libc.time(None)`

Fundamental data types

- Good support for many primitive C compatible data types:

C		Python
`char`	→	`c_char`
`int`	→	`c_int`
`long`	→	`c_long`
`void*`	→	`c_void_p`

Fundamental data types - usage

- All these types can be created by calling them with an optional initializer of the correct type and value:

```
i = c_int(42)
print i.value      # „42“
i.value = -1
print i.value      # „-1“

num = c_double(3.14)
libc.printf("Number: %f\n“, num)
                                  # „Numner: 3.14“
```

Using pointers

- `byref()` passes parameters by reference
  - `libc.sscanf("1 3.14 Hello", "%d %f %s", byref(i), byref(f), s)`

- Creating a pointer

```
i = c_int(42)
pi = pointer(i)
```

Return types

- Default return type: int

```
strcat = libc.strcat
strcat("abc", "def"))          # „8059983“

strcat.restype = c_char_p
strcat("abc", "def"))          # „abcdef“
```

Arrays

- Create an array-type
  - `TenIntsArrayType = c_int * 10`

- Create an array-instance
  - `array1 = TenIntegers()`
  - `array2 = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`

- Using arrays
  - `Array1[3]` → “0”
  - `Array2[3]` → “4”

Structures and unions

```
class POINT(Structure):
    _fields_ = [("x", c_int),
                ("y", c_int)]

point = POINT(10, 20)
print point.x, point.y          ➔   „10 20“
```

UnitTesting Apache Modules

- The problem
  - (Most) functions of a module could only be tested with a running apache
  - Module-functions could not be called directly
- The solutions
  - Starting and stopping an apache on each test
  - Test functions from the module directly using ctypes

Calling module functions directly

- Causes a exception stops execution
  - On runtime, ctypes tries to resolve all dynamic symbols
  - All apache specific methods and data structures are not available
- Solution:
  - Building Apache as a shared core

Building-kernel apache as a share core

- Building the apache kernel as shared module
  - On apache 1.x
    - `--enable-rule=SHARED_CORE`
  - On apache 2.x build infrastructure doesn't seem to know this anymore

Compiling Apache

- Compiling apache

  - `make clean`
  - `CFLAGS='-D SHARED_CORE -fPIC ' ./configure`
  - `make`

Linking the Shared Core

- After compiling, the make command links apache
  - `libtool ... -mode=link gcc ... -o httpd ..`
- Linking command for a shared core
  - `libtool ... -mode=link gcc ... -shared -o libhttpd.so ..server/exports.o`

Modifications of the Module

- Module must be linked against the shared core
  - `LDFLAGS = -lhttpd -L </…/libhttpd.so>`
  - Could be an extra make-target

Apache Data Structures in Python

```
class apr_allocator_t(Structure):

class apr_memnode_t(Structure):

class apr_pool_t(Structure):

class cleanup_t(Structure):
```

Setting Up Data Structures – apt_pool_t

```
class apr_pool_t(Structure):
    _fields_ = [("cleanups",POINTER(cleanup_t)),
                ("free_cleanups",POINTER(cleanup_t)),
                ("allocator",POINTER(apr_allocator_t)),
                ("subprocesses",POINTER(process_chain)),
                ("abort_fn",c_void_p),
                ("user_data",c_void_p),
                ("tag",c_char_p),
                ("active",POINTER(apr_memnode_t)),
                ("self",POINTER(apr_memnode_t)),
                ("self_first_avail",c_char_p),
                ("parent",POINTER(apr_pool_t)),
                ("child",POINTER(apr_pool_t)),
                ("sibling",POINTER(apr_pool_t)),
                ("ref",POINTER(POINTER(apr_pool_t)))]
```

Setting Up Data Structures – GCC

- Ctypes code generator – modified version of GCC
- Looks for declarations in C header files. Generates python codes for:
  - enums, structs, unions, function declarations, com interfaces, and preprocessor definitions
- Very early stage

Unit Test Framwork (nose)

- Simple structure, one class for each testing object
  - `Setup_class()`
  - `Test1()`
  - …
  - `TestX()`
  - `TearDown_class()`

Setting up the Test Environment

```
def setup (self) :
    self.catacomb = CDLL("/apachecon/libmod_dav_repos.so")
    self.httpd = CDLL("/apachecon/libhttpd.so")
    self.apr = CDLL("/apachecon/lib/libapr-1.so")

    self.pool = c_void_p()
    self.allocator = c_void_p()

    self.apr.apr_initialize()
    self.apr.apr_allocator_create(byref(self.allocator))
    self.apr.apr_pool_create_ex(byref(self.pool), None,
                                None, self.allocator)
```

Writing the Test

```
def testSomething(self):
    assert self.catacomb.function_to_test(arg1,
                        byref(arg2)) == “true”
```

Shutting down the Test Environment

```
def teardown(self):
    self.apr.apr_pool_destroy(self.pool)
    self.apr.apr_allocator_destroy(self.allocator)
    self.apr.apr_terminate()
```

Summary of Steps

- Compile Apache as a shared core
- Link own module against shared core
- Define the data structures you need
- Write the tests
- Run the test

Conclusion

- Powerful possibility to create tests with no need of a running Apache.
- Tests could be made in an easy language with possibility to easily make moc-objects.
- Writing a test is in most cases less than writing 10 lines of code.
- Tests are easily portable to other systems/apache-versions.

Demonstration

- Before the demo:
  - Thanks to **Steven Mohr**