CS130: Software Engineering

Lecture 6

https://forms.gle/u5h7AHWUtzNVMLuP8

A word: How's your team doing?
A tweet: Which is hardest and why: writing the implementation of a function, writing a unit test for that function, or writing an integration test that exercises the new behavior?

DI Frameworks
Testing State vs Interaction
Debugging
Integration Tests

Assignment 2

Assignment 2

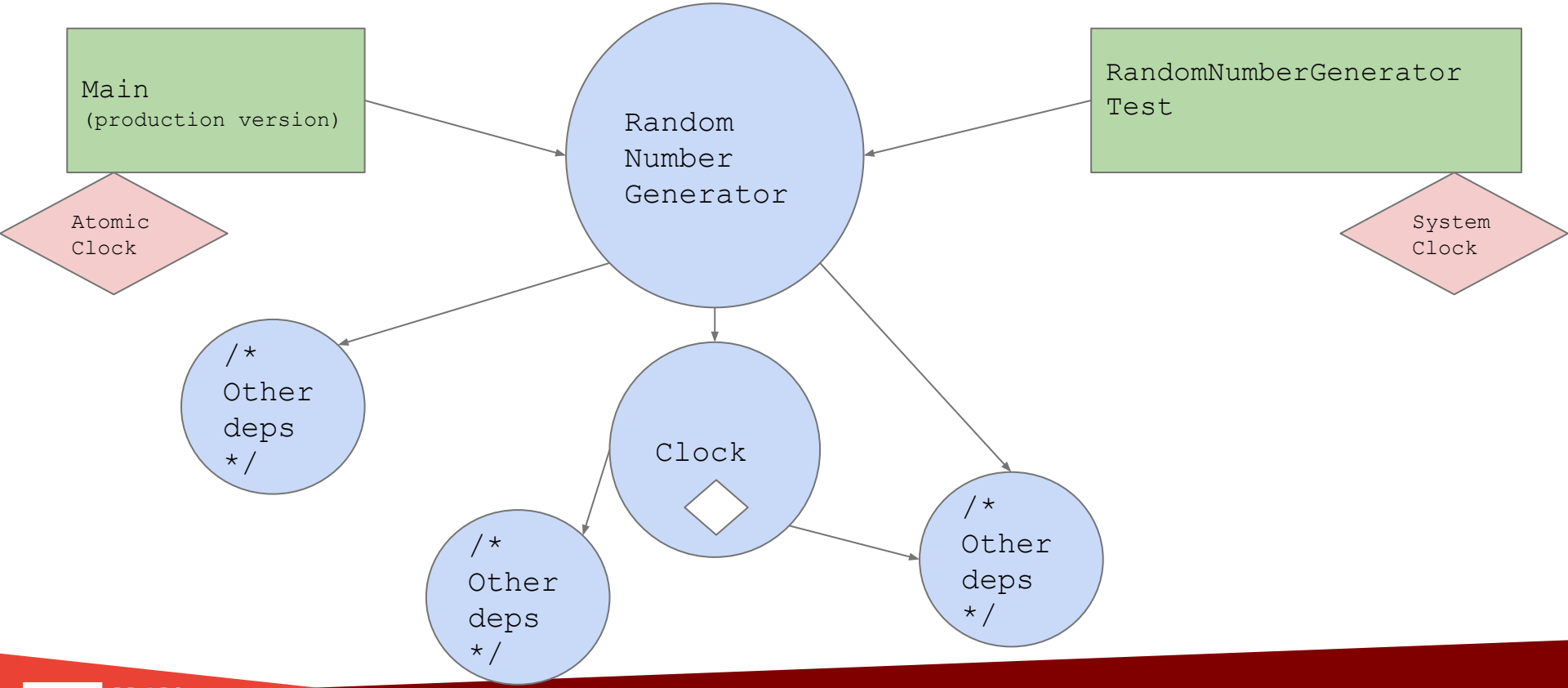
Featured:

- C++ / Boost
- Networking
- Docker
- Google Cloud Build
- Google Compute Engine

- What went well?
- What did not go well?
- What could be improved?

Dependency Injection Frameworks

Remember

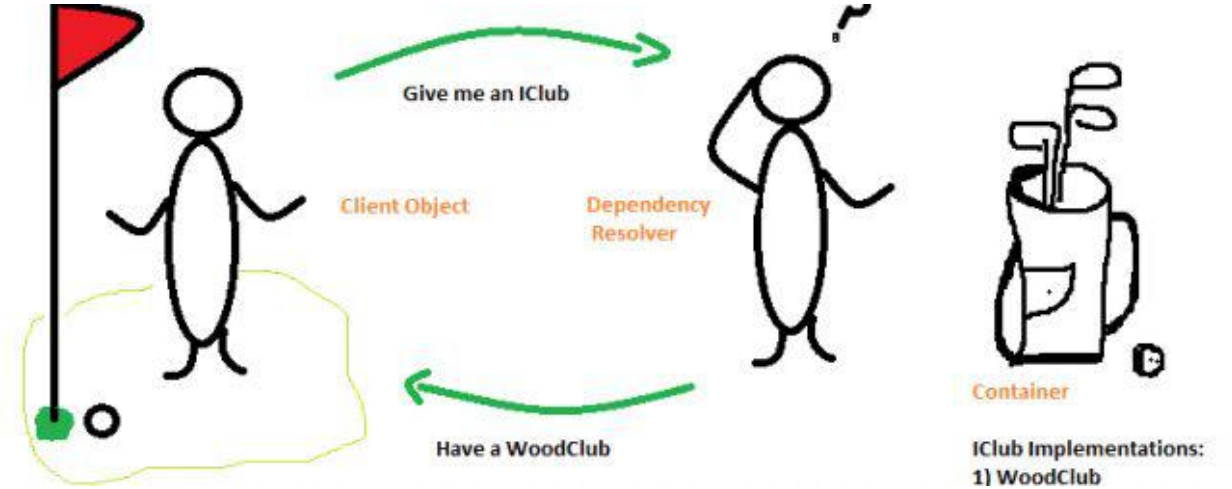

Main
(production version)
Atomic
Clock
Random
Number
Generator
RandomNumberGenerator
Test
System
Clock
/*
Other
deps
*/
Clock
/*
Other
deps
*/
/*
Other
deps
*/

Remember

```
public static void main() {
  RandomNumberGenerator rng =
    new RandomNumberGenerator(new SystemClock());

 /*
 * Instantiate 100s of concrete implementations???
 */
}
```

Dependency Injection

- Design pattern in which your objects **ask** for what they need instead of **retrieving** what they need
- In this way, your objects will depend solely on interfaces, and will be **implementation agnostic**



Dependency Injection: Practical application

```
void main() {
  Car car = new Car(
      new Engine(
          new Pistons(),
          new SparkPlug()),
      new Wheels(new Tires()),
      // and more ...
}
```

- Who calls new in your application?

Dependency Injection: Implementation

```
class Injector {
  Map<Interface, Implementation>
      bindings;

  T getInstanceOf(Interface key) {
      return bindings.get(key);
  }
}
```

- Injector class maintains map of interface to implementation
- Ask the injector for instances of objects instead of creating them yourself

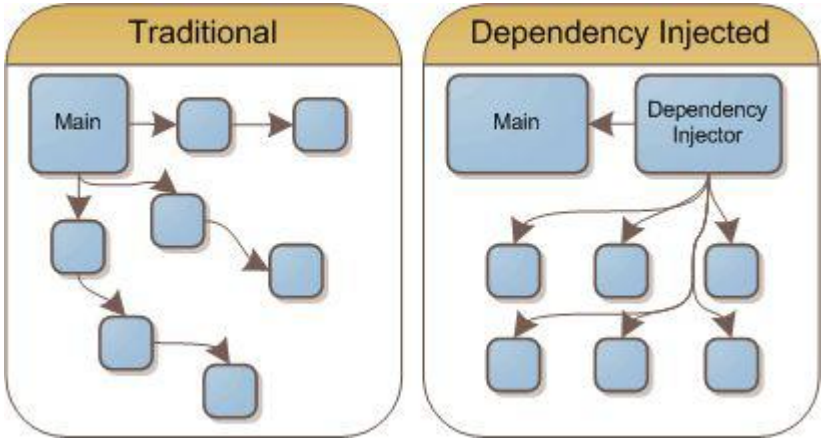
Dependency Injection: Use a Framework!

```
int main() {
  Injector injector = new Injector();
  injector->bindToImpl(Engine,
      InternalCombustionEngine);
  injector->bindToFactory(Wheels,
      createCoolWheels);
  injector->bindToValue(SparkPlug,
      localSparkPlug);
...
  Car myRealCar = injector.get(Car);
}
```

- Use a framework!
- Injectors are simple conceptually, but expensive to implement
- Lots of good frameworks to handle the tough stuff for you

One weird thing...

- If you do decide to adopt a DI framework, your code will look pretty different.
- Trying to understand parts of the Google codebase → no usages of `new` anywhere!



One weird thing... How are connection IDs generated?

- You see code like this:

```
Connection::Connection(IdGenerator idGenerator) {
  this.idGenerator = idGenerator;
}
```

- IdGenerator is an interface, so you look to where the Connection is used to find what implementation is being used:

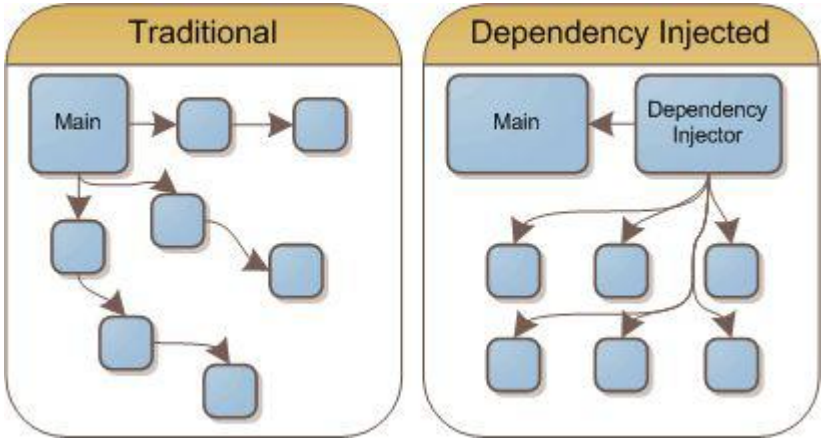
```
ConnectionHandler::ConnectionHandler(Connection connection) {
  this.connection = connection
}
```

- Uh oh...

One weird thing...

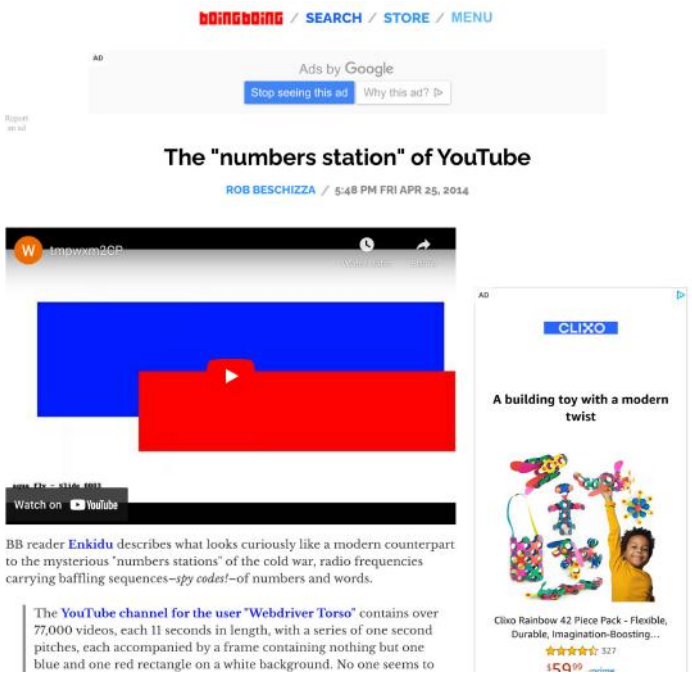
- What you're actually looking for may lie in a Module

```
ConnectionModule::init() {
  injector->bind(IdGenerator, DistributedIdGenerator);
  injector->bind(Connection, SecuredTcpConnection);
}
```



Testing

Testing : Webdriver Torso



"Webdriver also never sleeps, uploading about 400 videos on most days, every day Monday to Sunday."

We're never gonna give you uploading that's slow or loses video quality, and we're never gonna let you down by playing YouTube in poor video quality. That's why we're always running tests like Webdriver Torso."

Official Google Response

Mocks vs Fakes

Disclaimer

- We’ve seen different naming conventions for these types of objects
- We’re going to define them in one canonical way for the sake of this class (i.e. on the midterm)
- Focus more on the behavior than the terminology

Test Objects: Mocks, Fakes, etc..

Real object

- Same implementation you'd use in production

Real airplane

Test Objects: Mocks, Fakes, etc.

Fake object

- Trivial implementation of the interface that satisfies all contracts, but relies on memory only (e.g. `FakeDatabase` performs read/mutate operations as expected, but on an array in memory instead of using a SQL database)

Flight simulator

Test Objects: Mocks, Fakes, etc.

Mock object

- Placeholder test object that can be set up to respond to specific stimuli and report back how it was interacted with

Counterfeit watch

Test Objects: Mocks, Fakes, etc.

- Real objects should be preferred for testing
- Why?
  - Any mocks or fakes need to be written. Who wants to write more code than you need to?
  - Any mocks or fakes could have bugs. If you create mock or fake objects that break assumptions or contracts made by the real object, your test might miss bugs introduced in your code
  - Mocks and fakes introduce more maintenance cost. Since they have to mirror the behavior of the real object, any time the real object is updated you have to update your mocks and fakes

Test Objects: **Mocks**, Fakes, etc.

- Mock objects are also an option
- Pros:
  - Only need to define the behavior you care about in your test (e.g. if your service only uses the read operations in MyStorageService, no need to define mutates for MockStorageService)
  - Mocks can confirm some API contracts
- Cons:
  - Maintenance cost -- must be defined in every single test that has a dependency on a given object
  - Easy to misuse -- since we have these nice verify methods, might be tempted to test interaction instead of testing state

Test Objects: Mocks, **Fakes**, etc.

- Fake objects are usually next best thing to real objects
- Pros:
  - Only need to define it once. Service owner should also own and maintain the fake for their service.
  - Fake can be tested independently to verify its behavior
- Cons:
  - Maintenance cost -- must be updated in lockstep with the real thing
  - Sometimes not trivial to satisfy the service's contract in memory. Think about how you would write the minimal implementation of Socket that satisfied its API contracts

Testing state vs testing interaction

Testing state

```
public void testSortNumbers() {
     NumberSorter numberSorter = new NumberSorter(quicksort, bubbleSort);

     // Verify that the returned list is sorted. It doesn't matter which sorting
     // algorithm is used, as long as the right result is returned.
     assertEquals(
          new ArrayList(1, 2, 3), numberSorter.sortNumbers(new ArrayList(3, 1, 2)));
}
```

Testing state vs testing interaction

Testing interaction

```
public void testSortNumbers_quicksortIsUsed() {

    // Pass in mocks to the class and call the method under test.
    NumberSorter numberSorter = new NumberSorter(mockQuicksort, mockBubbleSort);
    numberSorter.sortNumbers(new ArrayList(3, 1, 2));

    // Verify that numberSorter.sortNumbers() used quicksort. The test should
    // fail if mockQuicksort.sort() is never called or if it's called with the
    // wrong arguments (e.g. if mockBubbleSort is used to sort the numbers).
    verify(mockQuicksort).sort(new ArrayList(3, 1, 2));
}
```

Testing state vs testing interaction

Testing interaction

```
public void testOpenConnection() {
     // Pass in mocks to the class and call the method under test.
     Connection connection = new Connection(mockSocket);
     connection.instantiate(80);

     verify(mockSocket).open(80);
}
```

Testing state vs testing interaction

Testing interaction

```
public void testOpenConnection() {
     // Pass in mocks to the class and call the method under test.
     Connection connection = new Connection(mockSocket);
     connection.instantiate(80);

     verify(mockSocket).open(80); // Fails!!!
}
```

- A zero day is discovered in the socket library you're using!
- socket.open() is immediately deprecated, and everyone must switch to socket.secureOpen() immediately!
- Your Connection shouldn't care about this implementation detail, but the test will start failing when you migrate to the new API even though behavior is unchanged.

Testing state vs testing interaction

- Quick note: one best practice to make sure you're testing state is to **test your public APIs**
- Your public API should have a clear contract that defines state that should be returned by your service, so your API documentation can serve as a blueprint of what to test

Testing state vs testing interaction

- A bad code smell is testing package private methods/making a method visible specifically for testing
- This almost always indicates you're testing some sort of interaction, since your test is now relying on implementation details of your implementation

Testing readability

Be sure to structure your tests as:

```
void test() {
  setupTestConditions();
  doSomething();
  expectResult();
}
```

- Tests are not tested, so err on the side of repetitive and verbose test
  - Refactoring into concise helper methods is best practice in production code, but can introduce untested programming errors when used in tests
- Testing is often great documentation
  - How often do you look for usages of an API to figure out how it works?

Testing readability

DRY (Don't repeat yourself)

```
void test1() {
  setupAccounts();
  expectThat(account.getBalance())
    .equals(1000)
}

void test2() {
  setupAccounts();
  account.withdraw(50);
  expectThat(account.getBalance())
    .equals(950);
}
```

DAMP (Descriptive and meaningful phrases)

```
void testDepositAndWithdraw() {
  account = new Account();
  account.open();
  account.deposit(1000);
  account.withdraw(50);
  expectThat(account.getBalance())
    .equals(950);
}
```

Integration tests

Integration tests

- We've refactored all of our code to make unit testing easy
- We've stubbed out all of the really hairy stuff (network, disk, etc.)
- How do we test the connections we've stubbed out?

Integration tests

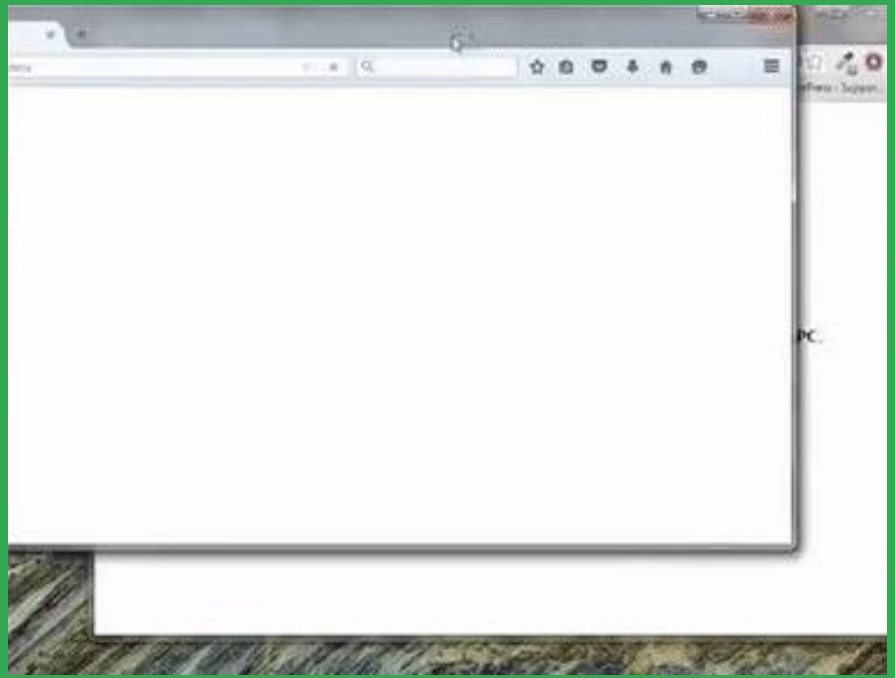
- Test code end to end
- These tests tend to be VERY expensive
  - Expensive to write (how do I connect my test code to all of these real production components?)
  - Expensive to keep green (so many dependencies, makes it very likely something messes up your configuration)
  - Expensive to run (since we're using resources like network, disk, tests tend to take a long time to run)

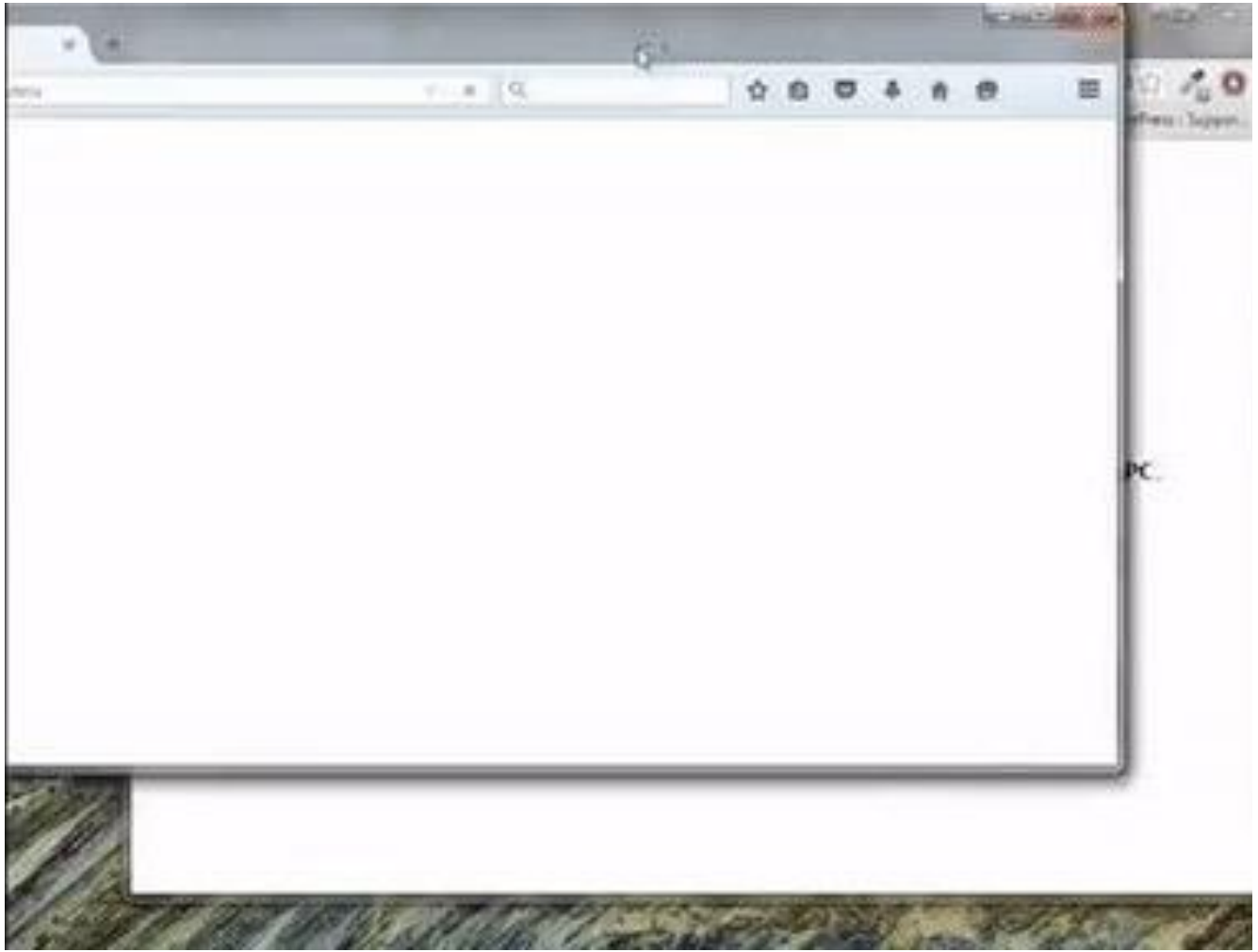
Integration tests

- How to pick your integration tests
  - Since the cost of integration tests is so high, the benefit better be high too
  - Pick a small handful of CRITICAL use cases for your application, and test those
  - Want to be 100% sure that if key functionality in your app is broken (by you or by any of your dependencies) that you know about it

Integration tests: Web Driver

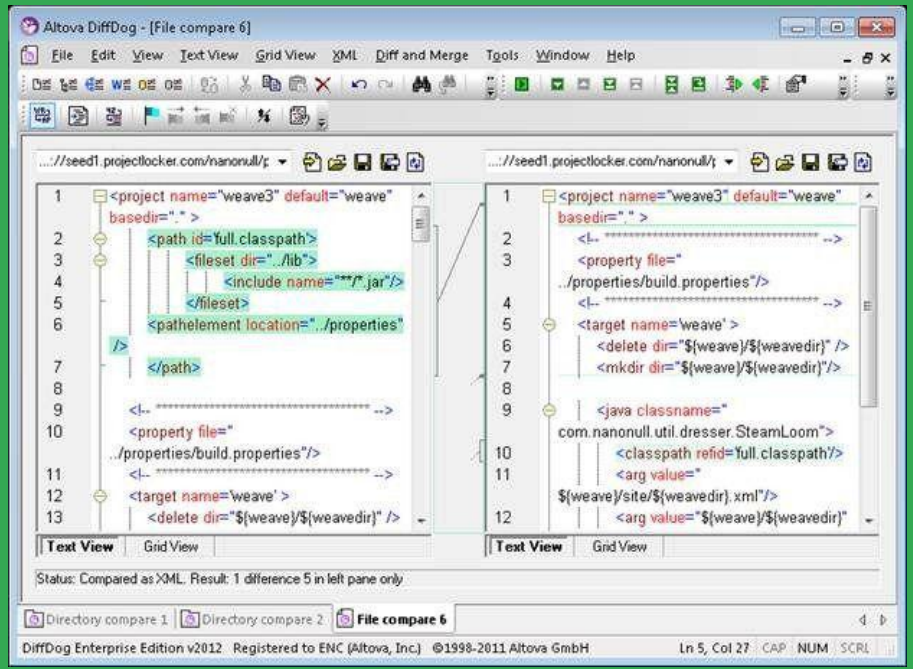
- Spins up a browser, navigates to web pages, interacts with the page
- Can verify structure of DOM as the script interacts with the page
- Takes many minutes to execute (build everything, start browser, wait for network calls)





Integration tests: Replay

- Test your API by firing a variety of real requests
- Keep a copy of “golden” response for reach request, and make sure requests coming back match the golden in all ways that matter

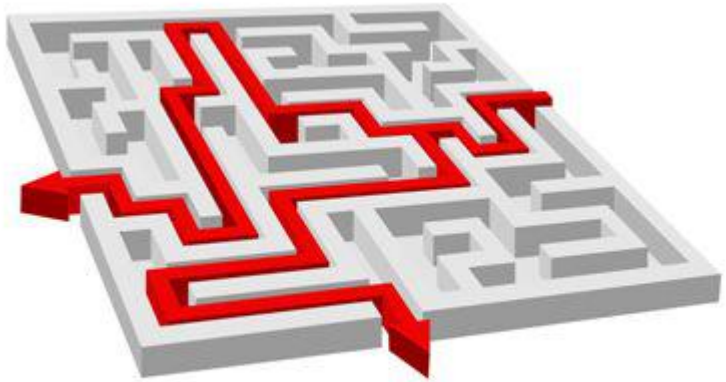


Integration tests: Screenshot

- Test your UI by rendering it (in browser, in Android emulator, etc) and taking a screenshot
- Keep a copy of “golden” screenshots for each request, and make sure screenshots you take match the golden
- Good for testing things like CSS changes
- Also great for code reviews, because any layout changes must also have new goldens to check in
    - Screenshots of your app are a lot easier to verify than a bunch of CSS changes

Debugging

What if you want to test the whole thing?



- Then you want an integration test
- This can manifest as a shell or python script, or a compiled program
- Naively, the script would start your web server, invoke a client, check the response
- The challenge of these types of tests typically related to:
  - How you define your “ends”
  - Flakiness from lots of moving parts

Integration tests

```
#!/bin/bash

expected=“Some special string”

# Start the webserver
./http_server --port 12345 &

# Run the test
if [ `curl http://localhost:12345 | grep $expected`x == ‘x’
];
then
    echo “FAILED”
else
    echo “SUCCESS”
fi

# Stop the webserver
kill %1
```

- This is a simplistic example*
- Allows you to test the “boilerplate” code to make sure it is still working.
- After all, if a test like this doesn’t work, then the whole application doesn’t really work

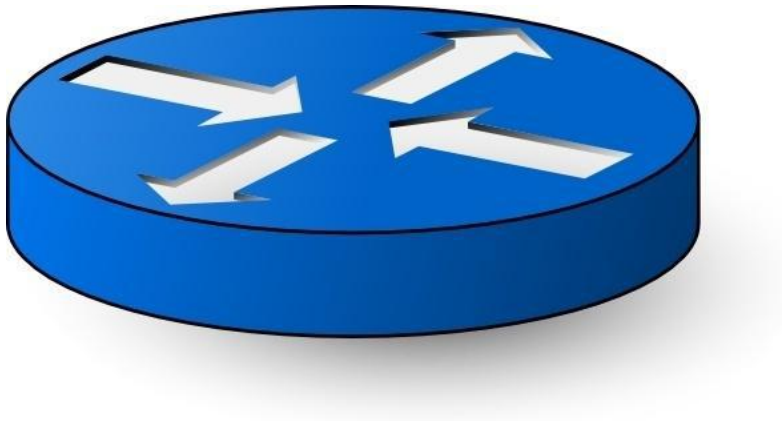
*read: this probably doesn’t work as written.

Web Server Components

Web Servers typically have a common internal structure that decomposes into:

- Request Handlers
- Request Container & Parser
- Response Container & Generator
- Mime Type Resolution
- and many more...

Request Handlers



- Encapsulates a certain type of request processing behavior
- Behavior is often modulated by the config and triggered by a particular URL path

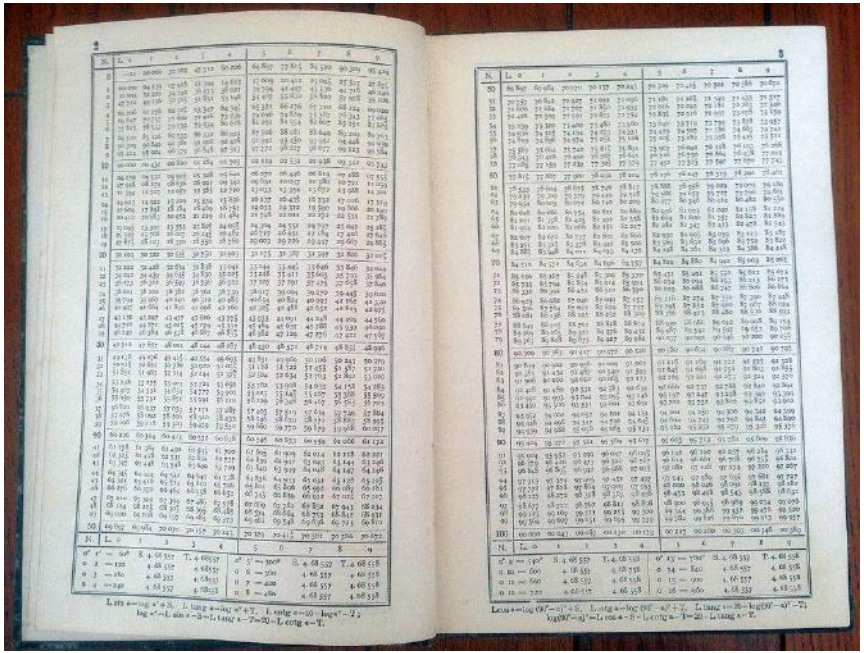
Request Container and Parser

- Helps parsing the request, in this case mediating HTTP
- Serves as a higher level representation of the logical request
- Could also handle things like HTTP 1.1 keep-alive

Response Container and Generator

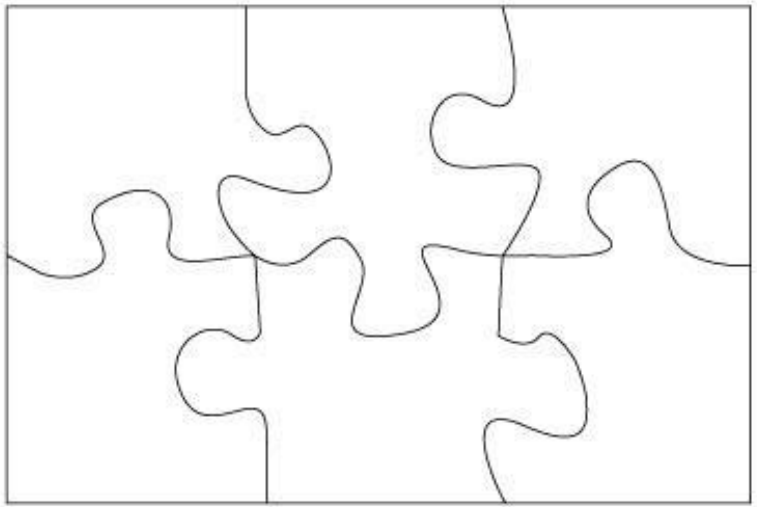
- Provides a place to incrementally build up a response
- Can encapsulate the details of actually translating the response into the HTTP protocol
- Might be responsible for handling streaming replies.

Mime Type Resolution



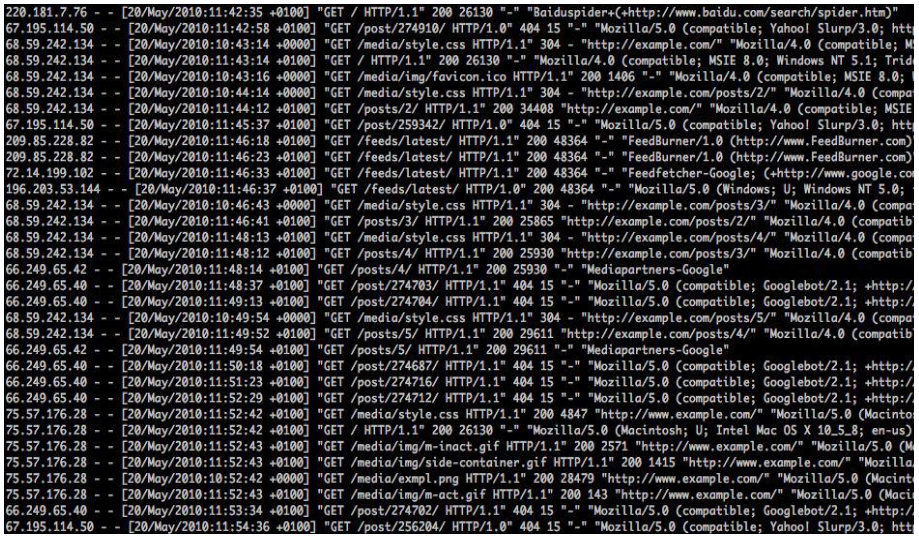
- For specialized tasks, like static file handling, you need to generate proper HTTP headers.
- For example, you need to tell the browser what type of file to expect in the body of the response.
- This object can encapsulate the mapping of a file's type on the filesystem to a HTTP header for the browser

Other Web Server Components



- There are many other specialized components.
- For inspiration, think about how you'd handle:
  - Streaming responses
  - Async responses
  - Caching

Debugging: Access Logging



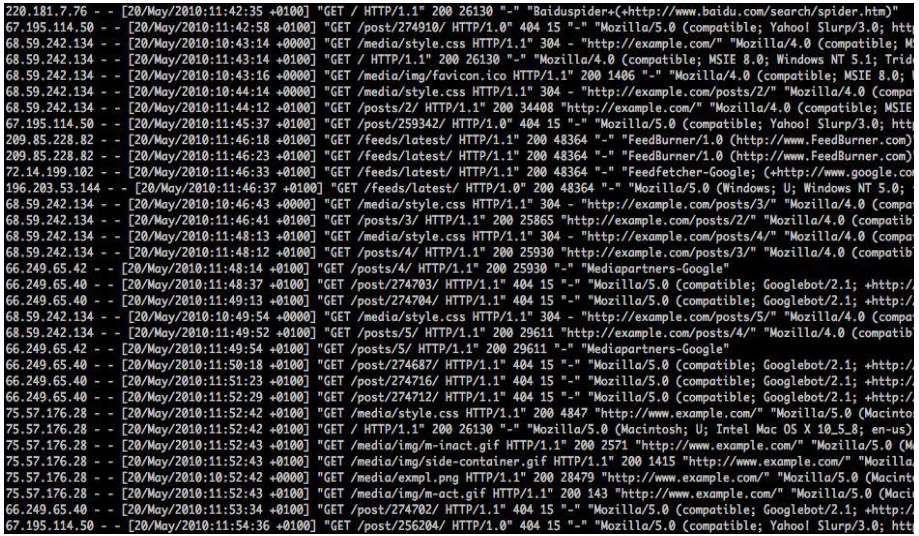
- Web servers usually have an access log that records
  - date, time
  - requestor IP
  - type of request (GET, POST, etc)
  - path
  - response code
- Helps after-the-fact bug archeology
- Typically one line / request, but can be augmented with other details

Debugging: Access Logging

```
220.181.7.76 - - [20/May/2010:11:42:35 +0100] "GET / HTTP/1.1" 200 26130 "-" "Baiduspider+(+http://www.baidu.com/search/spider.htm)"
67.195.114.50 - - [20/May/2010:11:42:58 +0100] "GET /post/274910/ HTTP/1.0" 404 15 "-" "Mozilla/5.0 (compatible; Yahoo! Slurp/3.0; http://help.yahoo.com/he
68.59.242.134 - - [20/May/2010:10:43:14 +0000] "GET /media/style.css HTTP/1.1" 304 - "http://example.com/" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5
68.59.242.134 - - [20/May/2010:11:43:14 +0100] "GET / HTTP/1.1" 200 26130 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2)"
68.59.242.134 - - [20/May/2010:10:43:16 +0000] "GET /media/img/favicon.ico HTTP/1.1" 200 1406 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Tride
68.59.242.134 - - [20/May/2010:10:44:14 +0000] "GET /media/style.css HTTP/1.1" 304 - "http://example.com/posts/2/" "Mozilla/4.0 (compatible; MSIE 8.0; Wind
68.59.242.134 - - [20/May/2010:11:44:12 +0100] "GET /posts/2/ HTTP/1.1" 200 34408 "http://example.com/" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
67.195.114.50 - - [20/May/2010:11:45:37 +0100] "GET /post/259342/ HTTP/1.0" 404 15 "-" "Mozilla/5.0 (compatible; Yahoo! Slurp/3.0; http://help.yahoo.com/he
209.85.228.82 - - [20/May/2010:11:46:18 +0100] "GET /feeds/latest/ HTTP/1.1" 200 48364 "-" "FeedBurner/1.0 (http://www.FeedBurner.com)"
209.85.228.82 - - [20/May/2010:11:46:23 +0100] "GET /feeds/latest/ HTTP/1.1" 200 48364 "-" "FeedBurner/1.0 (http://www.FeedBurner.com)"
72.14.199.102 - - [20/May/2010:11:46:33 +0100] "GET /feeds/latest/ HTTP/1.1" 200 48364 "-" "Feedfetcher-Google; (+http://www.google.com/feedfetcher.html; 9
196.203.53.144 - - [20/May/2010:11:46:37 +0100] "GET /feeds/latest/ HTTP/1.0" 200 48364 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.0; fr; rv:1.9.2.3) Gecko
68.59.242.134 - - [20/May/2010:10:46:43 +0000] "GET /media/style.css HTTP/1.1" 304 - "http://example.com/posts/3/" "Mozilla/4.0 (compatible; MSIE 8.0; Wind
68.59.242.134 - - [20/May/2010:11:46:41 +0100] "GET /posts/3/ HTTP/1.1" 200 25865 "http://example.com/posts/2/" "Mozilla/4.0 (compatible; MSIE 8.0; Windows
68.59.242.134 - - [20/May/2010:11:48:13 +0100] "GET /media/style.css HTTP/1.1" 304 - "http://example.com/posts/4/" "Mozilla/4.0 (compatible; MSIE 8.0; Wind
68.59.242.134 - - [20/May/2010:11:48:12 +0100] "GET /posts/4/ HTTP/1.1" 200 25930 "http://example.com/posts/3/" "Mozilla/4.0 (compatible; MSIE 8.0; Windows
66.249.65.42 - - [20/May/2010:11:48:14 +0100] "GET /posts/4/ HTTP/1.1" 200 25930 "-" "Mediapartners-Google"
66.249.65.40 - - [20/May/2010:11:48:37 +0100] "GET /post/274703/ HTTP/1.1" 404 15 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.h
66.249.65.40 - - [20/May/2010:11:49:13 +0100] "GET /post/274704/ HTTP/1.1" 404 15 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.h
68.59.242.134 - - [20/May/2010:10:49:54 +0000] "GET /media/style.css HTTP/1.1" 304 - "http://example.com/posts/5/" "Mozilla/4.0 (compatible; MSIE 8.0; Wind
68.59.242.134 - - [20/May/2010:11:49:52 +0100] "GET /posts/5/ HTTP/1.1" 200 29611 "http://example.com/posts/4/" "Mozilla/4.0 (compatible; MSIE 8.0; Windows
66.249.65.42 - - [20/May/2010:11:49:54 +0100] "GET /posts/5/ HTTP/1.1" 200 29611 "-" "Mediapartners-Google"
66.249.65.40 - - [20/May/2010:11:50:18 +0100] "GET /post/274687/ HTTP/1.1" 404 15 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.h
66.249.65.40 - - [20/May/2010:11:51:23 +0100] "GET /post/274716/ HTTP/1.1" 404 15 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.h
```

What might you want to debug with this? Would this information be enough?

Debugging: Verbose Logging



- Consider adding (optional) verbose logging.
- Perhaps could be triggered by certain conditions (bonus points for configuring those conditions at runtime).
- Can help export state throughout the process as a request is being processed.

Debugging: Logging Libraries

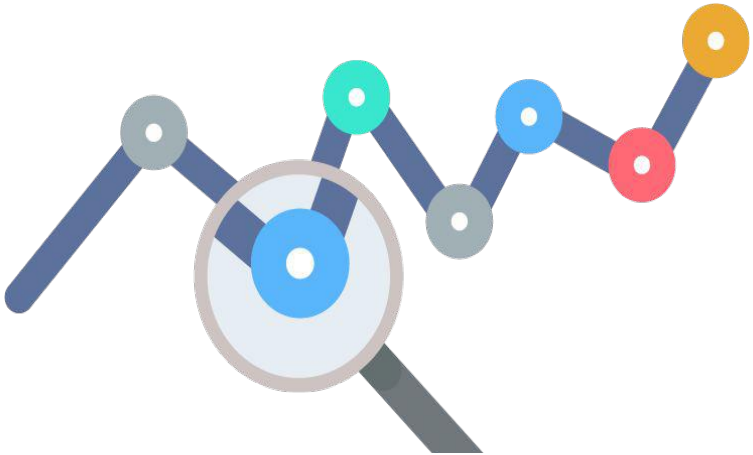
```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>

int main(int argc, char* argv[]) {
    // Initialize Boost logging library...

    [...]

    sources::severity_logger<int> lg;

    BOOST_LOG(lg) << “got request: ” << req.ToString();
    BOOST_LOG_SEV(lg, 1) << “this is the verbose request: ”
                         << req.VerboseString()

    [...]
}
```

- An example is the Boost logging library
- Several log streams (info, error, warning) (`boost::sources::severity_logger`)
- Can use `boost.StackTrace` for nice stack traces when you get `SIGSEGV` or other errors.

Debugging: Status Pages

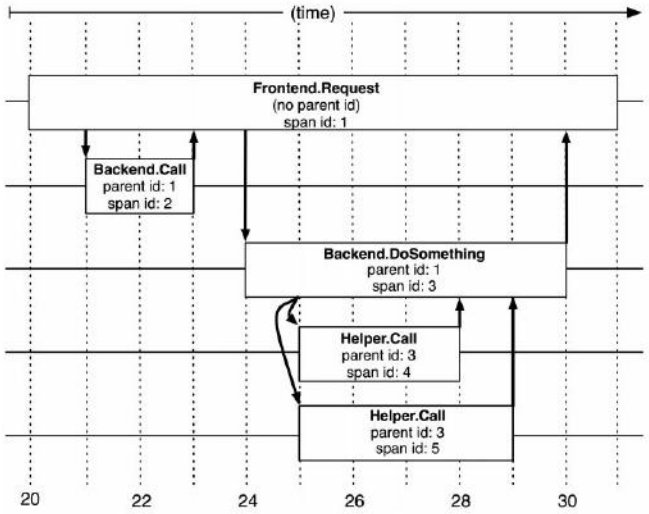
- For a live web server, might as well expose a debugging surface through a web page!
- Can render the log or present other statistics about the current process
  - Uptime
  - Number of requests handled
  - Mean request latency
  - etc...
- Collected stats can hook into monitoring, a topic we'll discuss in a later lecture

Debugging: Operation tracking



- Collect ops by type, e.g., those that invoke the same handler.
- Keep a distribution of various properties (latency, mem usage, etc.).
- Typically you need to sample the ops, so it is especially useful to track those that end in an error.

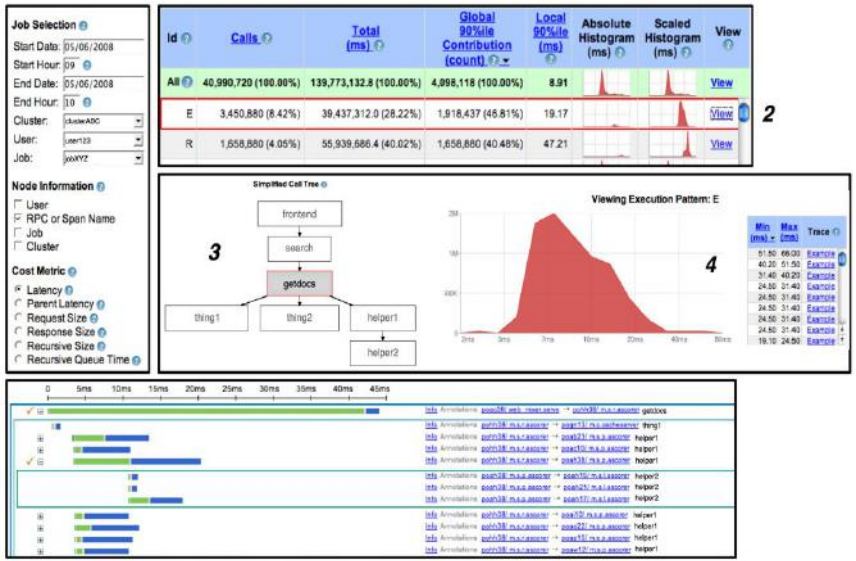
Debugging: Operation tracking + logging




```
const string& request = ...;
if (HitCache())
    TRACEPRINTF("cache hit for %s", request.c_str());
else
    TRACEPRINTF("cache miss for %s", request.c_str());
```

- Combining op tracking with logging is an even more powerful combination.
- You can attach a log to every op you perform so you can log messages running.
- Like before, each log entry should have a timestamp to help understand how long processing steps took.
- Helps determine what happened while processing a particular request.

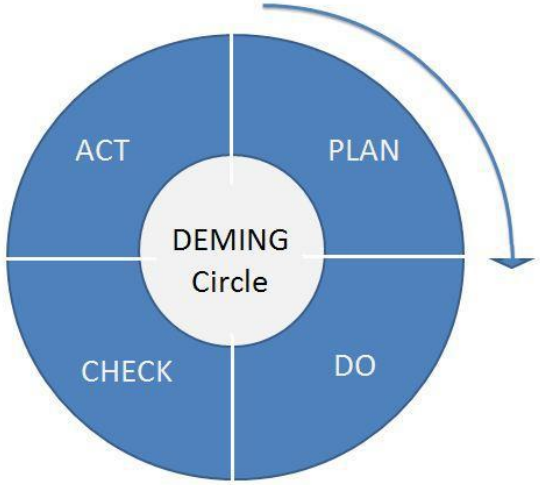
Debugging: Distributed Tracing



- The next logical step beyond op tracking
- Allows you to track a single logical op that crosses several servers in a distributed system
- Typically includes some interface to dig through the data since it is collected on several machines
- Provides a way to trigger such tracing on certain conditions (otherwise it is typically too expensive).

Continuous Build

Continuous Build (CI)

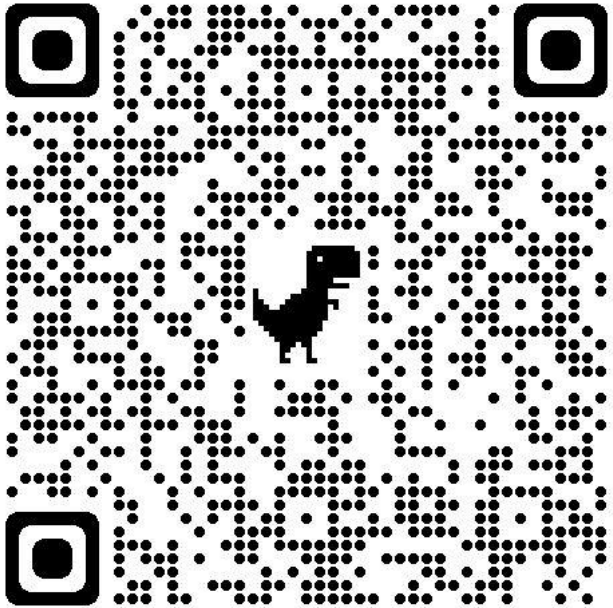


- Automate things you do yourself
  - Computer loves repetitive tasks
  - Computer loves repetitive tasks
  - Computer loves repeteteve tasks
- Ensures the project is moving from good state to good state
- Find errors quickly
- Facilitate collaboration

Continuous Build Example

https://bit.ly/3JKcaXN

A phrase: What is your test quest?
Tweet: A question you still have!
(the Rubber Duck will respond)



Appendix

Define:

```
interface IdGenerator { ... }
interface Connection { ... }

class RealIdGenerator implements IdGenerator { ... }
class FakeIdGenerator implements IdGenerator { ... }

class ApplicationModule extends AbstractModule {
  void configure() {
    bind(IdGenerator.class).to(RealIdGenerator.class);
    bind(Connection.class).to(FancyConnection.class);
  }
}

class TestApplicationModule extends AbstractModule {
  void configure() {
    bind(IdGenerator.class).to(FakeIdGenerator.class);
    bind(Connection.class).to(FancyConnection.class);
  }
}

class FancyConnection implements Connection {
  Connection(IdGenerator generator) {
    this.generator = generator;
  }
}
```

Use:

```
void main() {
  Injector injector = Injector.create(
      new ApplicationModule());
  Connection connection =
      injector.getInstance(Connection.class);
// connection is a FancyConnection
// with RealIdGenerator inside
}

void test_main() {
  Injector injector = Injector.create(
      new TestApplicationModule());
  Connection connection =
      injector.getInstance(Connection.class);
// connection is a FancyConnection
// with FakeIdGenerator inside
}
```