Principles of Software Construction: Objects, Design, and Concurrency

Design for Robustness: Distributed Systems

Christian Kästner **Vincent Hellendoorn**

Carnegie Mellon University
School of Computer Science

institute for
SOFTWARE
RESEARCH

Outline

- Intro to distributed systems
- Robustness and Failures
- Testing large/distributed systems
    - Mocks, Stubs

Where we are

	Small scale: One/few objects	*Mid scale:* Many objects	*Large scale:* Subsystems
Design for understanding, change/ext., reuse, robustness, ...	Subtype Polymorphism ✓	Domain Analysis ✓	GUI vs Core ✓
	Information Hiding, Contracts ✓	Inheritance & Del. ✓	Frameworks and Libraries ✓, APIs ✓
	Immutability ✓	Responsibility Assignment, Design Patterns, Antipattern ✓	Module systems, microservices
	Types	Promises/ Reactive P. ✓	**Testing for Robustness**
	Unit Testing ✓	Integration Testing ✓	CI ✓, DevOps, Teams

Recap: Designing for Robustness

- Single-threaded, local systems:
  - Problems are (usually) deterministic
  - Checked vs. unchecked exceptions
- Key ideas:
  - ???

Recap: Designing for Robustness

- Single-threaded, local systems:
  - Problems are (usually) deterministic
  - Checked vs. unchecked exceptions
- Key ideas:
  - Provide explicit control-flow for normal and abnormal execution
    - Error handling and recovery for the latter
  - Unit testing to increase confidence
    - Cover both typical and atypical/boundary paths

Recap: Designing for Robustness

- Concurrent, local systems:
  - Non-determinism from thread ordering, asynchronous returns
  - Errors can occur at any shared mutable state
- Key ideas:
  - ???

Recap: Designing for Robustness

- Concurrent, local systems:
  - Non-determinism from thread ordering, asynchronous returns
  - Errors can occur at any shared mutable state
- Key ideas:
  - Reduce mutable state
    - Use atomicity, synchronization everywhere else
  - Organize asynchrony with promises
    - Especially natural in a single-threaded environment

Designing for Robustness

- Key ideas:
  - Provide explicit control-flow for normal and abnormal execution
    - Error handling and recovery for the latter
  - Test normal and abnormal execution

Designing for Robustness

- Key ideas:
  - Provide explicit control-flow for normal and abnormal execution
    - Error handling and recovery for the latter
  - Test normal and abnormal execution
- Until now, most of the program was under our control
  - What if something goes wrong and it's not our fault?
  - What if the system is too big to test?

What is a distributed system?

“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”

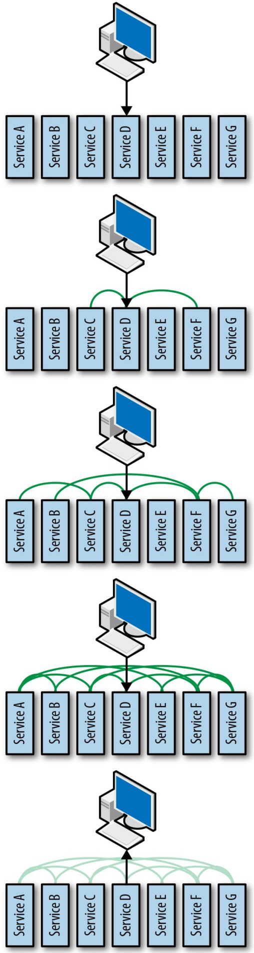
-- Leslie Lamport

What is a distributed system?

- Multiple system components (computers) communicating via some medium (the network) to achieve some goal
- “Concurrent” (shared-memory multiprocessing) vs. Distributed
  - **Agents**: Threads vs. Processes
    - Processes typically spread across multiple computers
    - Can put them on one computer for testing
  - **Communication**: changes to Shared Objects vs. Network Messages

Distributed systems

- A collection of autonomous systems working together to form a single system
  - Enable scalability, availability, resiliency, performance, etc …



Designing for Robustness

- Concurrent, distributed systems:
    - Non-determinism risks almost everywhere
        - Left-pad gone? Better not rebuild your apps.
        - DB busy? Queries could time out.
        - Use any API? Prepare for down-time
    - Errors can occur at any external call
- Key ideas:
    - ???

What will you do if

- An API your data plugin uses is temporarily down?
    - Or returns a surprising error code

Retry!

- Maybe wait a bit.
  - How Long? How often?

Retry!

- Exponential Backoff
  - Retry, but wait exponentially longer each time
  - Assumes that failures are exponentially distributed
    - E.g., a 10h outage is extremely rare, a 10s one not so crazy
  - E.g.:

```
const delay = retryCount => new Promise(resolve =>
                   setTimeout(resolve, 10 ** retryCount));

const getResource = async (retryCount = 0, lastError = null) => {
  if (retryCount > 5) throw new Error(lastError);
  try {
    return apiCall();
  } catch (e) {
    await delay(retryCount);
    return getResource(retryCount + 1, e);
  }
```

https://www.bayanbennett.com/posts/retrying-and-exponential-backoff-with-promises/

Retry!

- Still need an exit-strategy
  - Learn HTTP response codes
    - Don’t bother retrying on a 403 (go find out why)
  - Use the API response, if any
    - Errors are often documented -- e.g., GitHub will send a “rate limit exceeded” message

```
const delay = retryCount => new Promise(resolve =>
                    setTimeout(resolve, 10 ** retryCount));

const getResource = async (retryCount = 0, lastError = null) => {
  if (retryCount > 5) throw new Error(lastError);
  try {
    return apiCall();
  } catch (e) {
    await delay(retryCount);
    return getResource(retryCount + 1, e);
  }
```

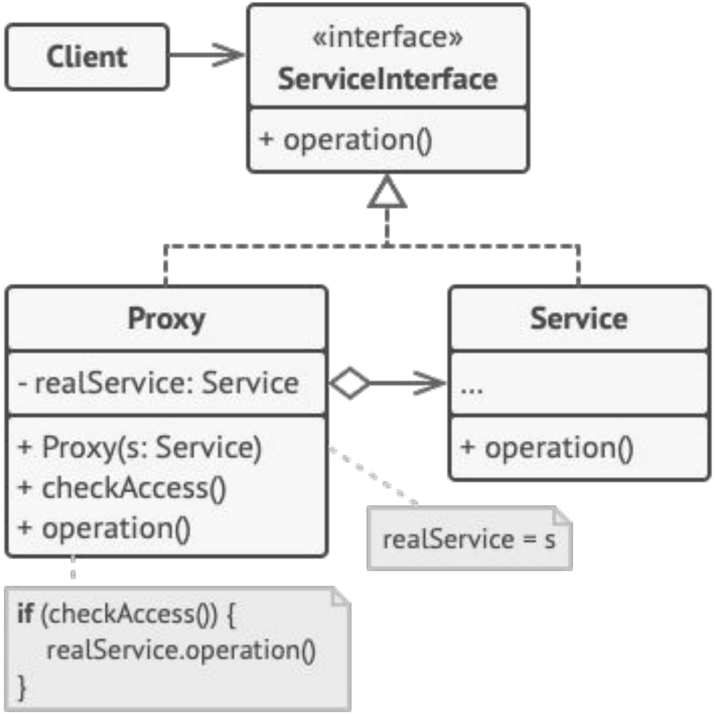
https://www.bayanbennett.com/posts/retrying-and-exponential-backoff-with-promises/

Handling Recovery

- We need a fallback plan
  - Can't just `e.printStackTrace()`
  - What *can* we do?

Proxy Design Pattern

- Local representative for remote object
  - Create expensive obj on-demand
  - Control access to an object
- Hides extra “work” from client
  - Add extra error handling, caching
  - Uses *indirection*



https://refactoring.guru/design-patterns/proxy

Example: Caching

```
interface FacebookAPI {
    List<Node> getFriends(String name);
}
class FacebookProxy implements FacebookAPI {
    FacebookAPI api;
    HashMap<String,List<Node>> cache = new HashMap…
    FacebookProxy(FacebookAPI api) { this.api=api;}

    List<Node> getFriends(String name) {
        result = cache.get(name);
        if (result == null) {
            result = api.getFriends(name);
            cache.put(name, result);
        }
        return result;
    }
}
```

Example: Caching and Failover

```
interface FacebookAPI {
    List<Node> getFriends(String name);
}
class FacebookProxy implements FacebookAPI {
    FacebookAPI api;
    HashMap<String,List<Node>> cache = new HashMap…
    FacebookProxy(FacebookAPI api) { this.api=api;}

    List<Node> getFriends(String name) {
        try {
            result = api.getFriends(name);
            cache.put(name, result);
            return result;
        } catch (ConnectionException c) {
            return cache.get(name);
        }
    }
```

Example: Redirect to Local Service

```
interface FacebookAPI {
    List<Node> getFriends(String name);
}
class FacebookProxy implements FacebookAPI {
    FacebookAPI api;
    FacebookAPI fallbackApi;
    FacebookProxy(FacebookAPI api, FacebookAPI f) {
        this.api=api; fallbackApi = f; }

    List<Node> getFriends(String name) {
        try {
            return api.getFriends(name);
        } catch (ConnectionException c) {
            return fallbackApi.getFriends(name);
        }
    }
```

Principle: Delegating Recovery

- We need a fallback plan
  - Can't just `e.printStackTrace()`
  - What *can* we do?
- In case of failure, redirect
  - If at all plausible, hand work over to proxy
    - Local data(set), fallback service
  - If not, recruit clean-up service
    - Proces, display errors

What will you do if

- An API your data plugin uses is temporarily down?
  - Or returns a surprising error code
- Consider caching
  - E.g., store last Twitter feed, Target shopping card offline
  - Not cheap, select caching mechanism carefully
  - If user-facing: be transparent about offline status

What will you do if

- Your visualization plugin's latest version has a vulnerability?

Ever looked at NPM Install's output?

```
npm WARN deprecated babel-eslint@10.1.0: babel-eslint is now @babel/eslint-parser. This package will no longer receiv
updates.
npm WARN deprecated chokidar@2.1.8: Chokidar 2 will break on node v14+. Upgrade to chokidar 3 with 15x less dependenc
s.
npm WARN deprecated svgo@1.3.2: This SVGO version is no longer supported. Upgrade to v2.x.x.
npm WARN deprecated querystring@0.2.1: The querystring API is considered Legacy. new code should use the URLSearchPar
s API instead.
npm WARN deprecated @hapi/joi@15.1.1: Switch to 'npm install joi'
npm WARN deprecated rollup-plugin-babel@4.4.0: This package has been deprecated and is no longer maintained. Please u
 @rollup/plugin-babel.
npm WARN deprecated fsevents@1.2.13: fsevents 1 will break on node v14+ and could be using insecure binaries. Upgrade
o fsevents 2.
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in cert
n circumstances, which is known to be problematic.  See https://v8.dev/blog/math-random for details.
npm WARN deprecated querystring@0.2.0: The querystring API is considered Legacy. new code should use the URLSearchPar
s API instead.
npm WARN deprecated sane@4.1.0: some dependency vulnerabilities fixed, support for node < 10 dropped, and newer ECMAS
ipt syntax/features added
npm WARN deprecated flatten@1.0.3: flatten is deprecated in favor of utility frameworks such as lodash.
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated @hapi/bourne@1.3.2: This version has been deprecated and is no longer supported or maintained
```

Ever looked at NPM Install's output?

```
added 2110 packages from 770 contributors and audited 2113 packages in 141.9

158 packages are looking for funding
  run `npm fund` for details

found 27 vulnerabilities (8 moderate, 18 high, 1 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
```

Vulnerabilities in Distributed Systems

- A lot of software relies on vulnerable code somewhere deep down
  - Often not disclosed/discovered for quite a while
  - By then, it could be everywhere
- What can you do?
  - Routinely check using tools (e.g. dependabot, CI is great)
  - Upgrade/downgrade where possible, ditch bad packages otherwise
  - Area of active research


ars TECHNICA

NPM package with 3 million weekly downloads had a severe vulnerability

What will you do if

- Facebook withdraws its DNS routing information?

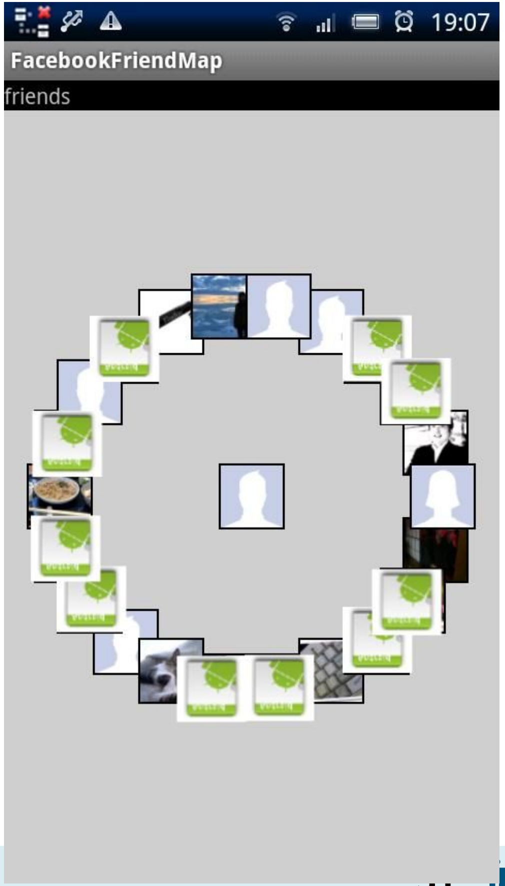
https://blog.cloudflare.com/october-2021-facebook-outage/

Testing Distributed Systems

- Challenges:
  - Volatility
    - Real-world effects -- things crashing, delays.
    - Users are hard to simulate
  - Performance
    - Massive databases? Systems with minutes-long start-up times?
    - Very common in ML

For example

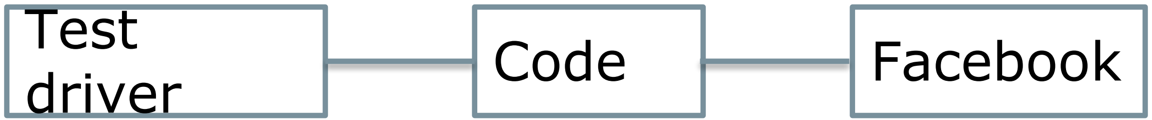
- 3rd party Facebook apps
- Android user interface
- Backend uses Facebook data



Testing in real environments


```
void buttonClicked() {
   render(getFriends());
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   List<Node> persons = api.getFriends("john");
   for (Node person1 : persons) {
      ...
   }
   return result;
}
```

Eliminating Android dependency

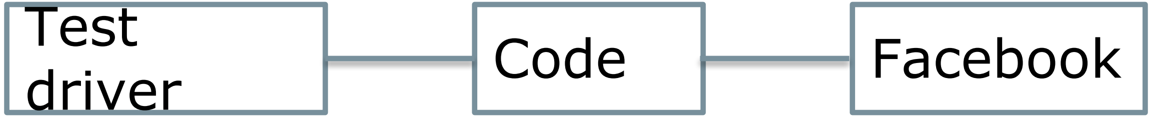



```
@Test void testGetFriends() {
   assert getFriends() == ...;
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookAPI api = new FacebookAPI(c);
   List<Node> persons = api.getFriends("john");
   for (Node person1 : persons) {
      ...
   }
   return result;
}
```

That won’t quite work

- **GUI applications process *thousands* of events**
- Solution: automated GUI testing frameworks
  - Allow streams of GUI events to be captured, replayed
- These tools are sometimes called *robots*

Eliminating Android dependency




```
@Test void testGetFriends() {
    assert getFriends() == ...;
}
List<Friend> getFriends() {
    Connection c = http.getConnection();
    FacebookAPI api = new FacebookAPI(c);
    List<Node> persons = api.getFriends("john");
    for (Node person1 : persons) {
        ...
    }
    return result;
}
```

How about this one?

Test Doubles

- Stand in for a real object under test
- Elements on which the unit testing depends (i.e. collaborators), but need to be approximated because they are
  - Unavailable
  - Expensive
  - Opaque
  - Non-deterministic
- Not just for distributed systems!

http://www.kickvick.com/celebrities-stunt-doubles

How Test Doubles Help

1. Speed: simulate response without going through the API

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
            List<Node> result=new List();
      result.add(…);
      return result;
         }
     }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
            List<Node> result=new List();
          result.add(…);
          return result;
            }
         }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
            List<Node> result=new List();
      result.add(…);
      return result;
         }
     }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)
4. Insight: expose internal state

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
            List<Node> result=new List();
      result.add(…);
      return result;
         }
     }
}
```

How Test Doubles Help

1. Speed: simulate response without going through the API
2. Stability: guaranteed deterministic return, reduces flakiness
3. Coverage: reliably simulate problems (e.g., return 404)
4. Insight: expose internal state
5. Development: presume functionality not yet implemented

```
class FakeFacebook implements FacebookInterface {
    void connect() {}
    List<Node> getFriends(String name) {
        if ("john".equals(name)) {
            List<Node> result=new List();
      result.add(…);
      return result;
        }
     }
}
```

Types of Test Doubles

- Most often talk about Mocks and Stubs
  - Technically, a few other categories, see next slide
- Mocks give you a lot of power
  - Dictate what should be returned when (very broadly construed)
  - Requires framework using reflection
    - E.g., Mockito in Java; Mock functions in Jest*
- Stubs are way simpler; use when possible

*https://jestjs.io/docs/mock-functions

Design Implications

- Think about testability when writing code
- When a mock may be appropriate, design for it
- Hide subsystems behind an interface
- Use factories, not constructors to instantiate
- Use appropriate tools
    - Dependency injection or mocking frameworks

What will you do if

- Facebook withdraws its DNS routing information?
  - Fact-of-life: be prepared (test for this)
  - Reduce coupling; don’t let someone else’s outage cripple your program
    - Like separating your GUI from the backend

https://blog.cloudflare.com/october-2021-facebook-outage/

Designing for Robustness

- As a *client* of distributed systems (mainly the Internet):
  - No harm trying again (redundancy)
  - Have a backup plan (resiliency)
  - Maintain awareness of what can go wrong (transparency)
    - HTTP status codes, API documentation, keeping tabs on vulnerabilities

Designing for Robustness

- As a *client* of distributed systems (mainly the Internet):
    - No harm trying again (redundancy)
    - Have a backup plan (resiliency)
    - Maintain awareness of what can go wrong (transparency)
        - HTTP status codes, API documentation, keeping tabs on vulnerabilities
    - **Isolation, isolation, isolation**
        - Use test doubles liberally
        - Rely on protocols to contain and manage failures
        - Never let one module crash another
            - More pointers coming up

For Application Designers

Some considerations when contributing to *the* distributed system

Why build a distributed system?

Why build a distributed system?

- Unlimited scaling
  - Can be used for capacity or speed
- Geographical dispersion – people and data around the world
- **Robustness** to failures including physical catastrophes

???

Why build a distributed system?

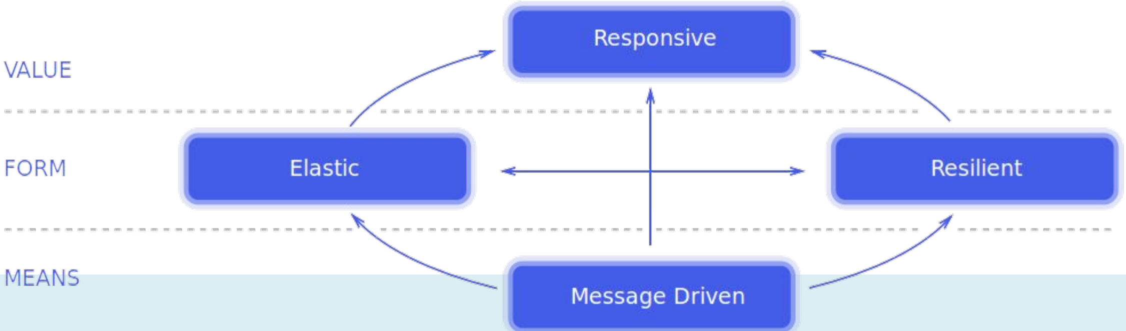
- Test Santorini all you want, it will die when I turn off my laptop
  - A local server is a Single Point of Failure
- Distributed systems offer robustness through redundancy, duplication
  - Netflix famously unplugs random servers in production

Measuring Robustness

- **Reliability:** works well
  - Often in terms of availability: fraction of time system is working
    - 99.999% available is "5 nines of availability"
- **Performance:** works fast
  - Low latency
  - High throughput
- **Scalability:** adapts well to increased demand
  - Ability to handle workload growth

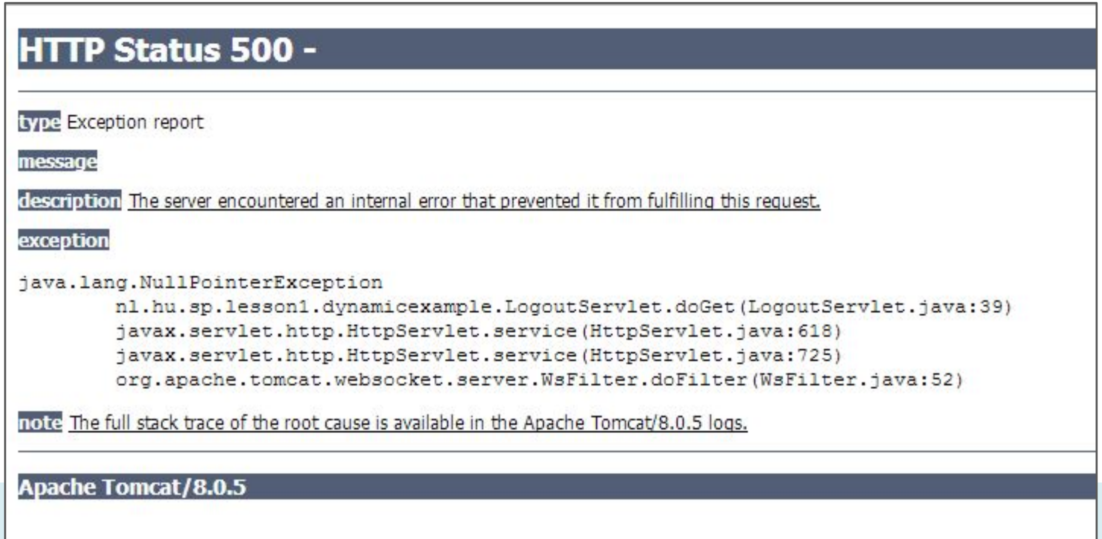
Robust Distributed System Design

- Consider reading: https://www.reactivemanifesto.org
  - Yet another meaning for “Reactive”!
  - Short guide identifying key principles
    - Goals: robustness, resilience, flexibility
    - Principles: responsiveness, elasticity, message-driven
    - Patterns/Heuristics: isolation, delegation, verbosity, replication, asynchrony



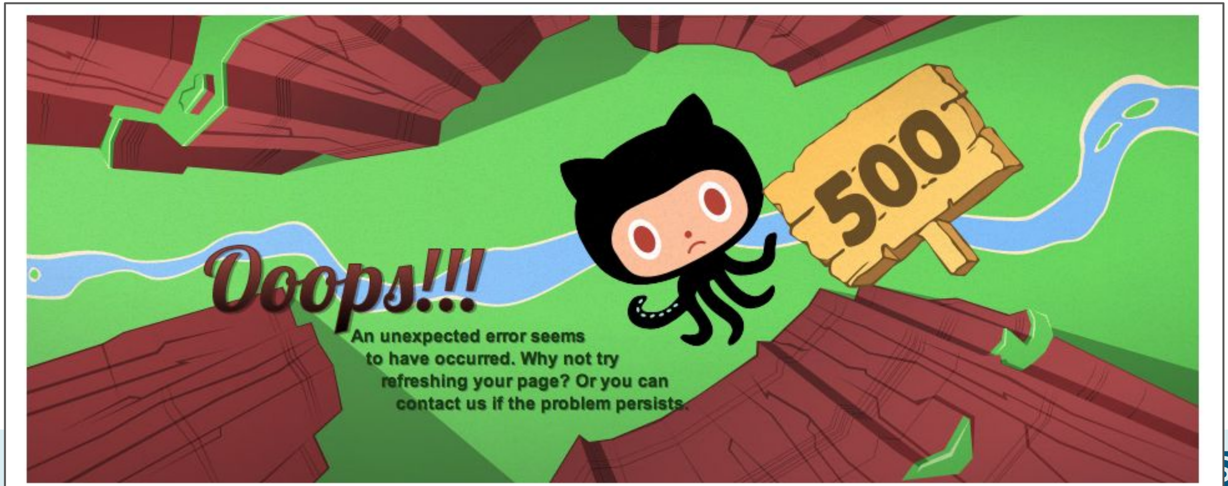
Principle: Modular Protection

- Errors should be contained and isolated
  - A failing printer should not corrupt a document
  - Handle exceptions locally as much as possible, return useful feedback
  - **Don’t do this:**



Principle: Modular Protection

- Online: use HTTP response status codes effectively
  - Don’t just hand out 404, 500
    - Unless they really apply
  - Provide and <u>document</u> fall-back options, information
    - Good RESTful design helps



Principle: Delegating Recovery

(Again?)

- Don’t make a failing node/module serve a client
  - It needs to clean itself up
  - Forward clients to designated recovery service
    - A bit like the proxy pattern
  - Consider asynchrony
    - Failure is often expensive

Principle: Consider Idempotence

- Idempotency: the same call from the same context should have the same result
  - Hitting “Pay” twice should not cost you double!
  - A resource should not suddenly switch from JSON to XML
  - Makes APIs predictable, resilient

Ensuring Idempotence

- Fairly easy for read-only requests
  - Ensure consistency of read-only data
  - Never attach side-effects to GET requests*
- Also for updates, deletes
  - Not “safe”, because data is mutated
  - Natural idempotency because the target is identified
- How about writing/sending new data?

*https://twitter.com/rombulow/status/990684463007907840

Ensuring Idempotence

- How about writing/sending new data?
  - Could fail anywhere
    - Including in displaying success message after payment!
  - POST is not idempotent
  - Use Unique Identifiers
  - Server keeps track of requests already handled

```
curl https://api.stripe.com/v1/charges \
   -u sk_test_BQokikJOvBiI2HlWgH4olfQ2: \
   -H "Idempotency-Key: AGJ6FJMkGQIpHUTX" \
   -d amount=2000 \
   -d currency=usd \
   -d description="Charge for Brandur" \
   -d customer=cus_A8Z5MHwQS7jUmZ
```

https://stripe.com/blog/idempotency

Distributed Systems

There are entire courses on getting these right; not a goal here

But do:

- Understand challenges and solutions to achieving robustness
  - Primarily as a *client* of a distributed system (we all are these days)
  - Test for all scenarios, leveraging test doubles
  - Provide error handling through isolation
- Learn to communicate with, and provide your own, nodes
  - API design, last week
  - Microservices, next week