Principles of Software Construction: Objects, Design, and Concurrency

Design for Robustness: Distributed Systems

Bogdan Vasilescu **Claire Le Goues**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

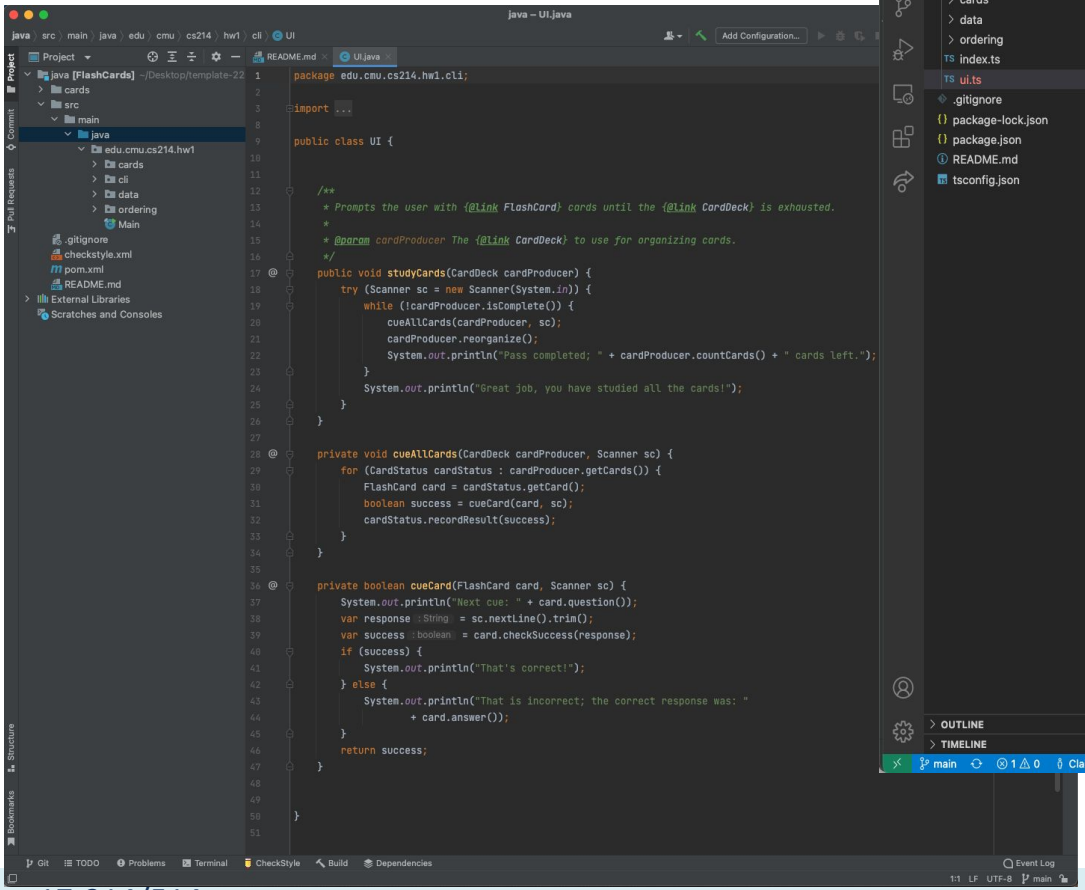
Reading/quiz for Tuesday.

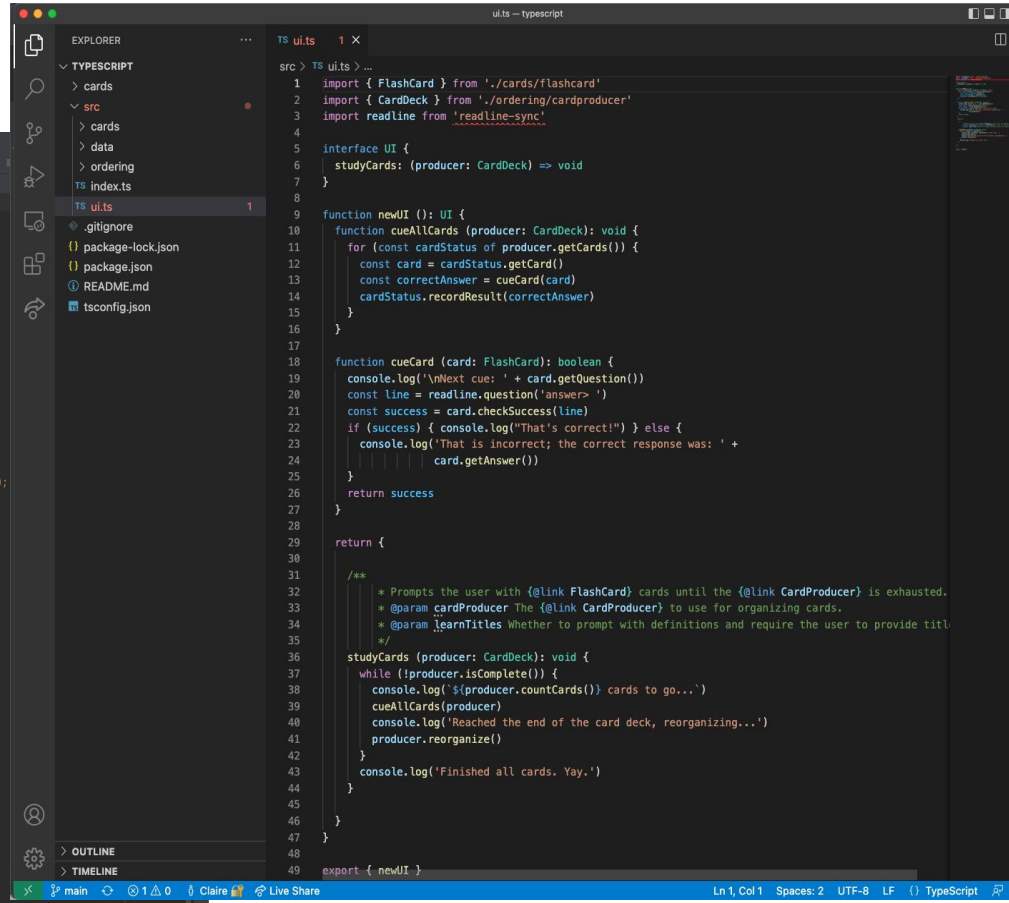
We are working with selected teams to clean up framework code, will announce/release later today.

Where we are

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

Where did we start?





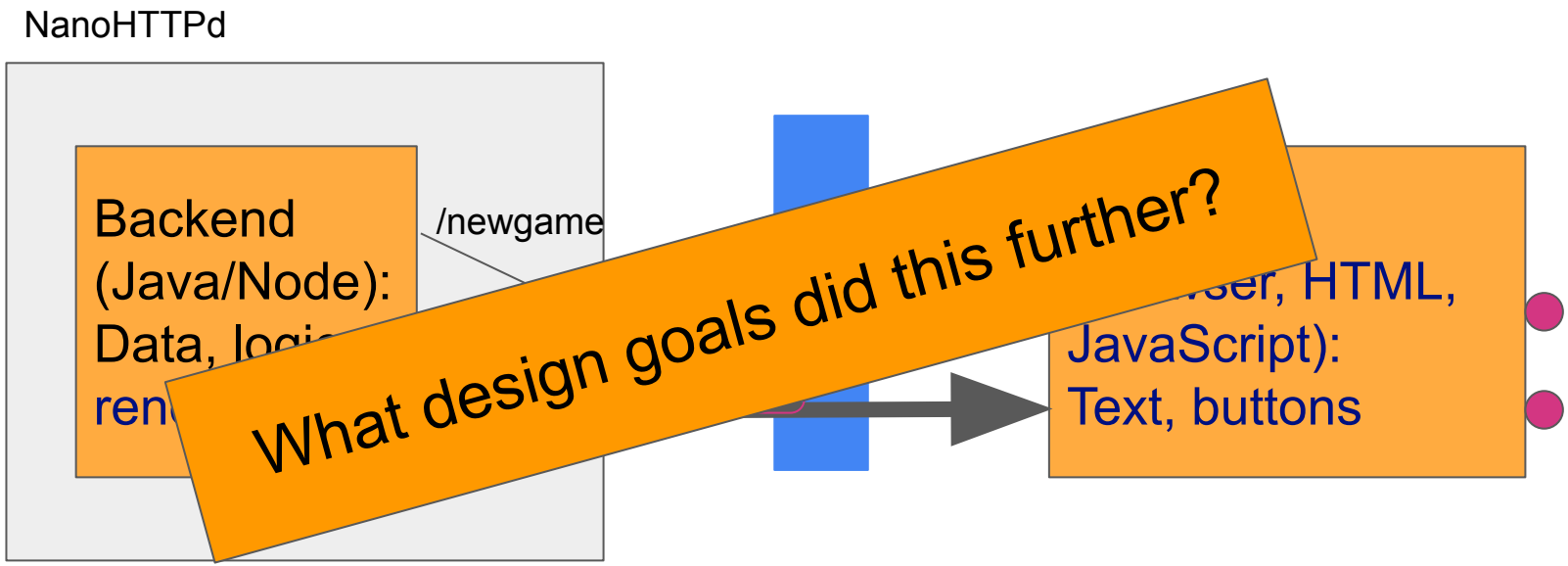
We design for robustness even in “small” systems.

- Single-threaded, local systems:
  - Problems are (typically) 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

We design for robustness even in “small” systems.

- 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

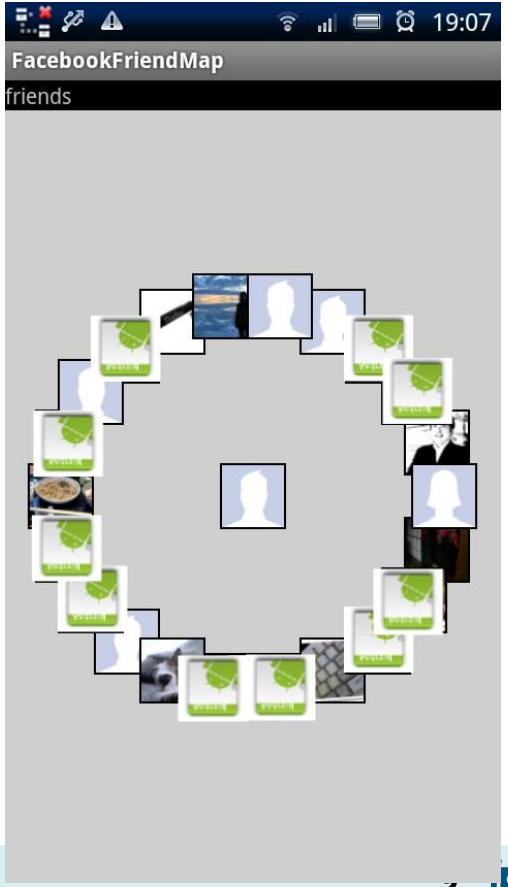
Then we forced you to do this…

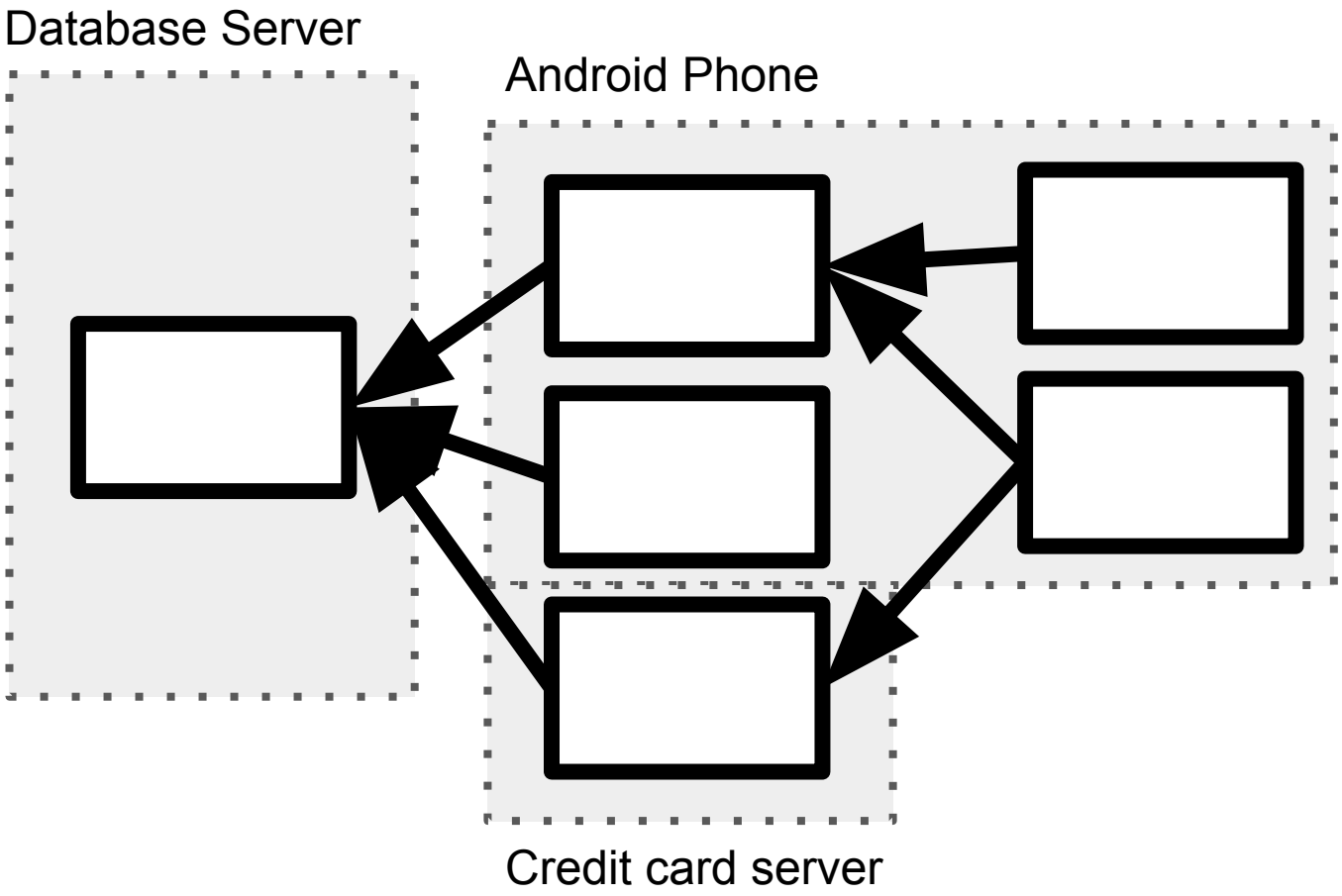


Modern software is dominated by systems composed of [components, APIs, modules], developed by completely different people, communicating over a network!

For example

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




Database Server
Android Phone
Credit card server

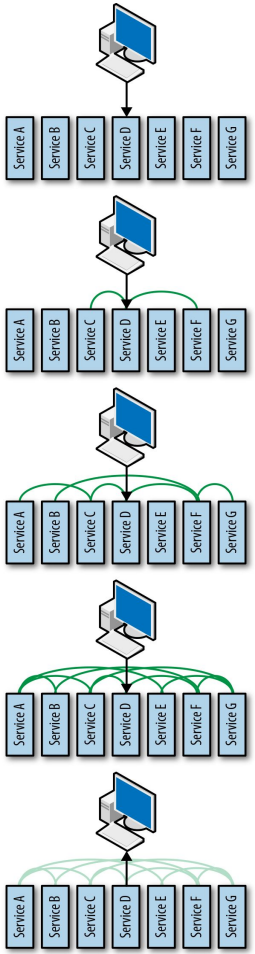
What is a distributed system?

- 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 …
- Remote procedure calls instead of function calls
  - Typically REST API to URL

- Benefits? Drawbacks?



Distributed System Benefits

Scalability

Very strong encapsulation (only APIs public)

Computation beyond local resources

Independent deployment, operations, and evolution

Also multiple containers on single system

Pay per transaction / storage / use

What is a distributed system?

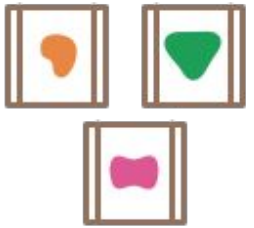
“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”

-- Leslie Lamport

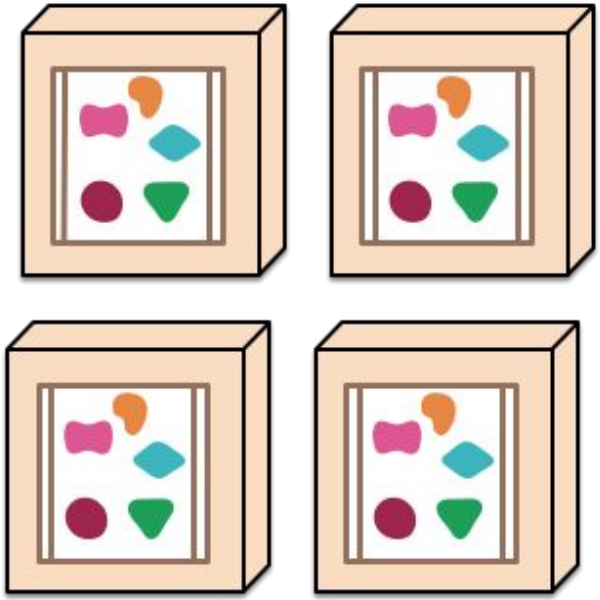

Microservices
Microservices Everywhere

A monolithic application puts all its functionality into a single process...

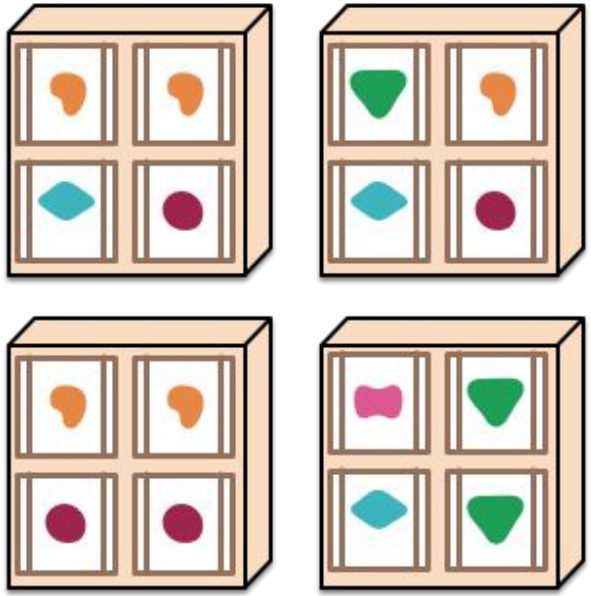
A microservices architecture puts each element of functionality into a separate service...



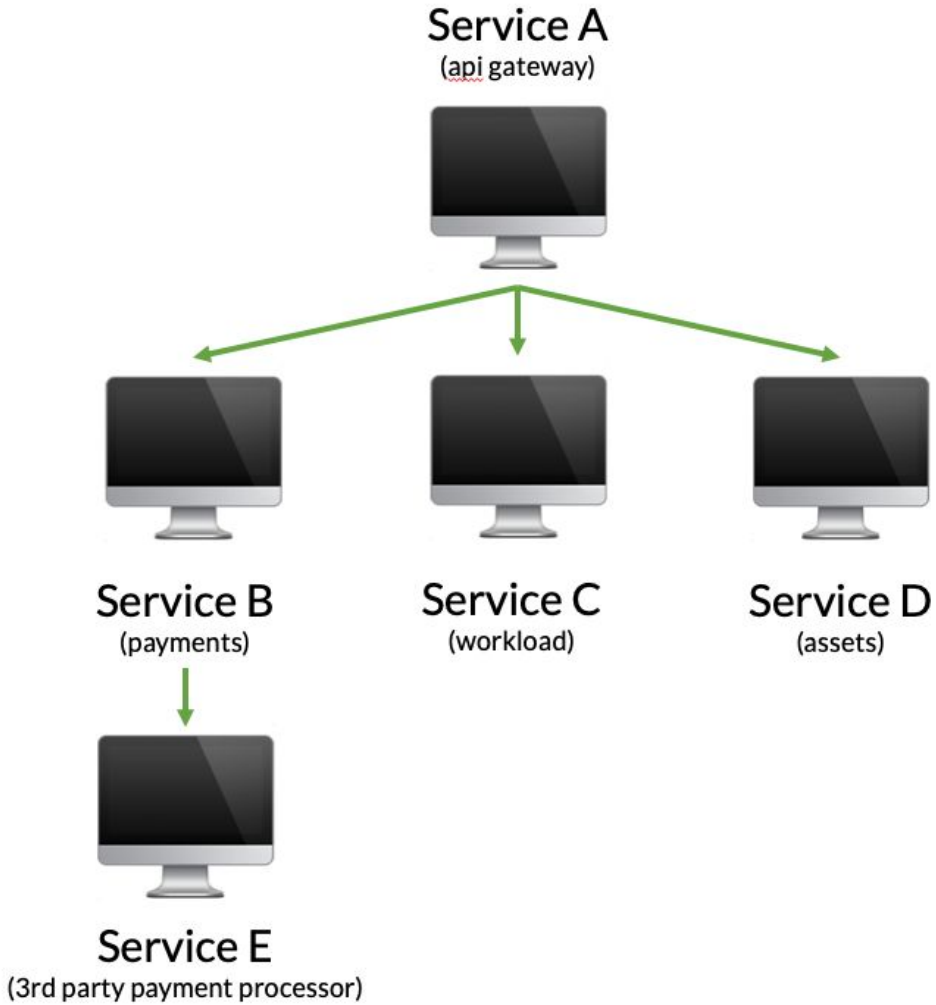
... and scales by replicating the monolith on multiple servers

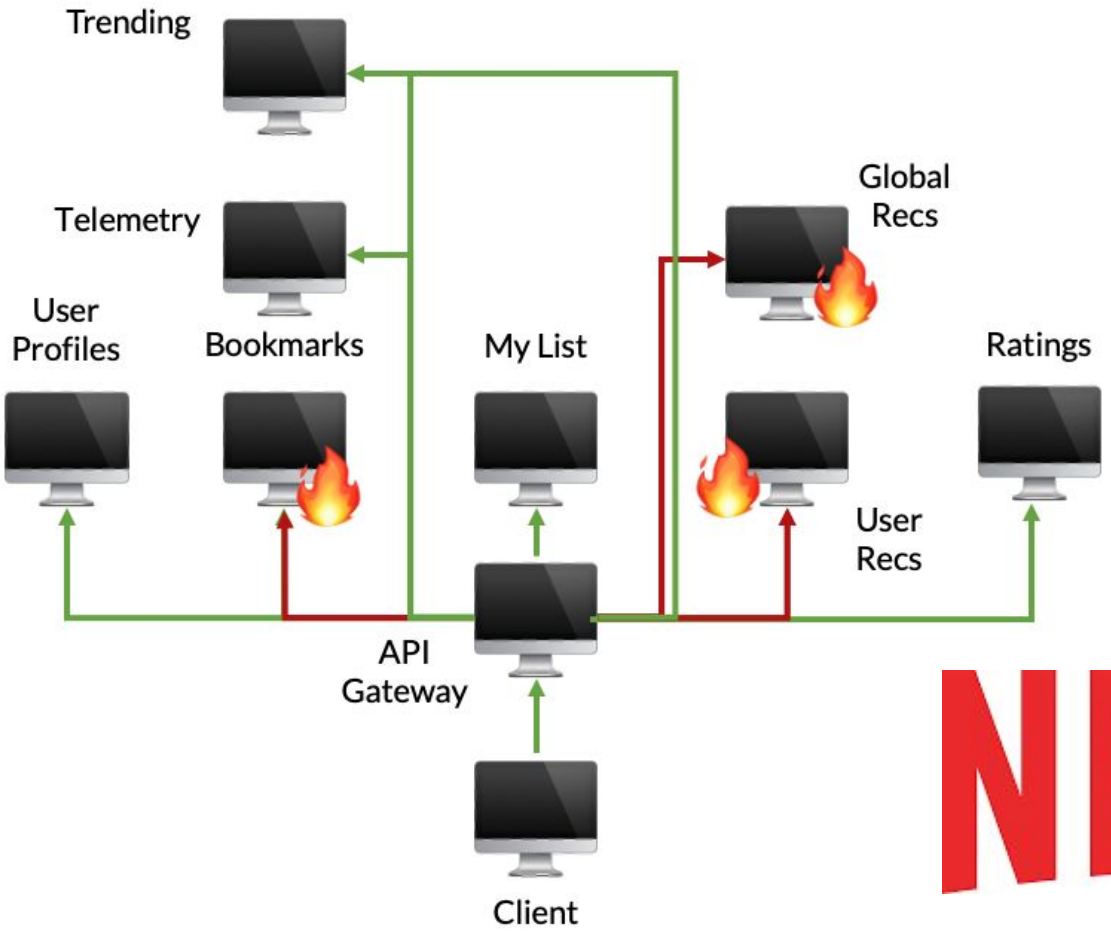


... and scales by distributing these services across servers, replicating as needed.



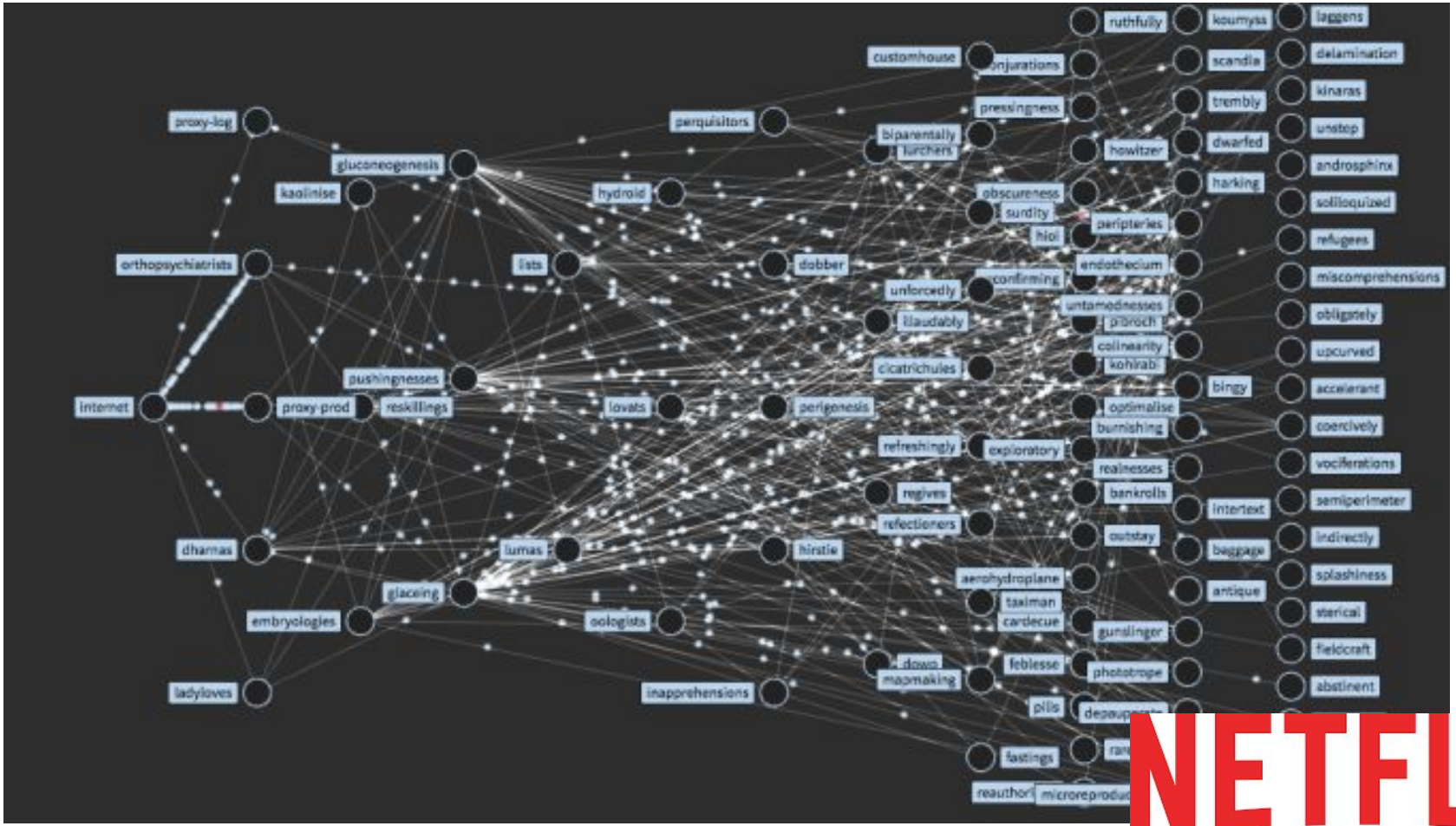
source: http://martinfowler.com/articles/microservices.html


Service A
(api gateway)
Service B
(payments)
Service C
(workload)
Service D
(assets)
Service E
(3rd party payment processor)



NETFLIX

http://christophermeiklejohn.com/filibuster/2021/10/14/filibuster-4.html



NETFLIX

Microservices

Building applications as suite of small and easy to replace services

- fine grained, one functionality per service
- (sometimes 3-5 classes)
- composable
- easy to develop, test, and understand
- fast (re)start, fault isolation

Modelled around business domain

Interplay of different systems and languages, no commitment to technology stack

Easily deployable and replicable

Embrace automation, embrace faults

Highly observable

Technical Considerations

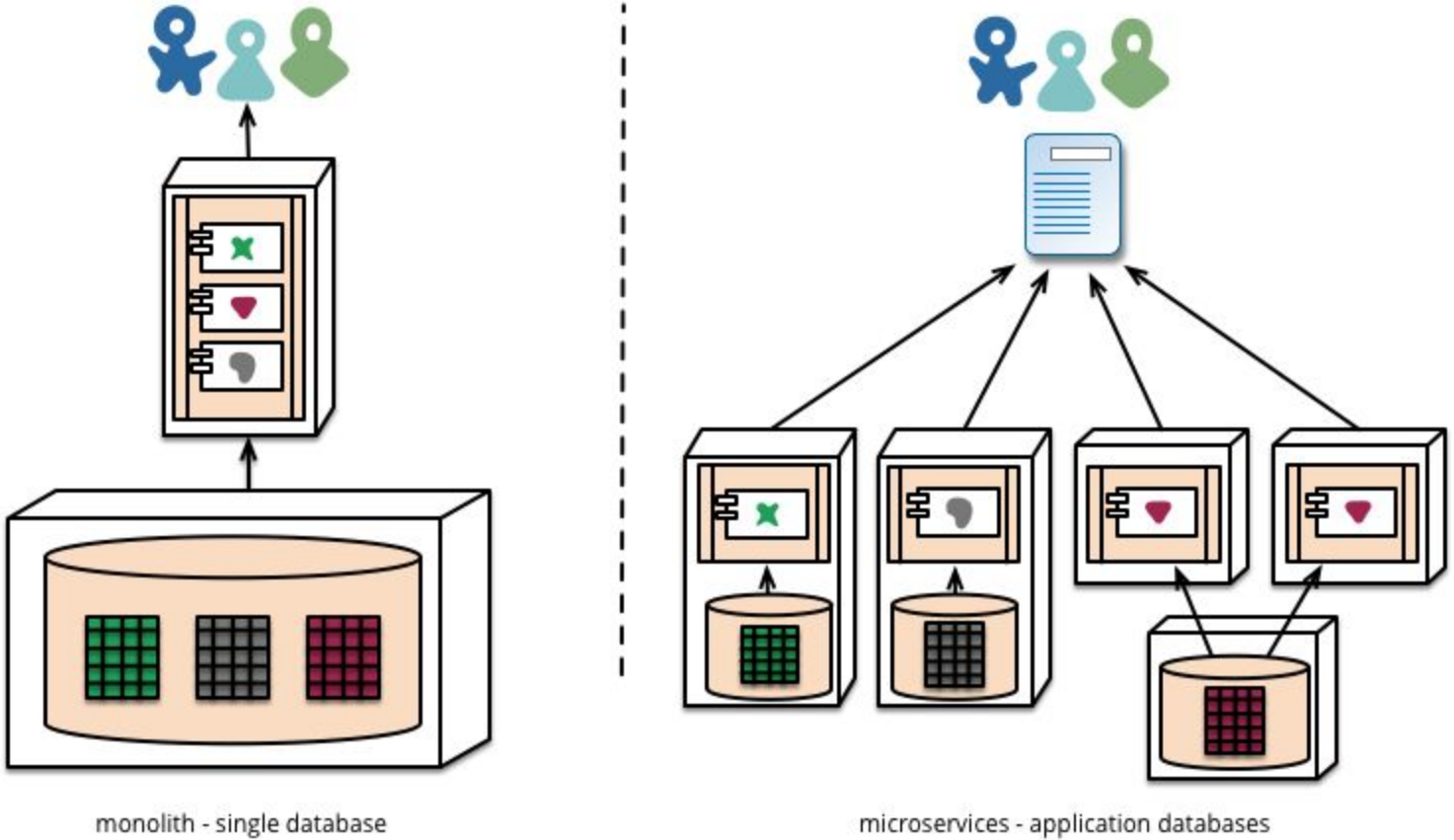
REST APIs

Independent development and deployment

Self-contained services (e.g., each with own database)

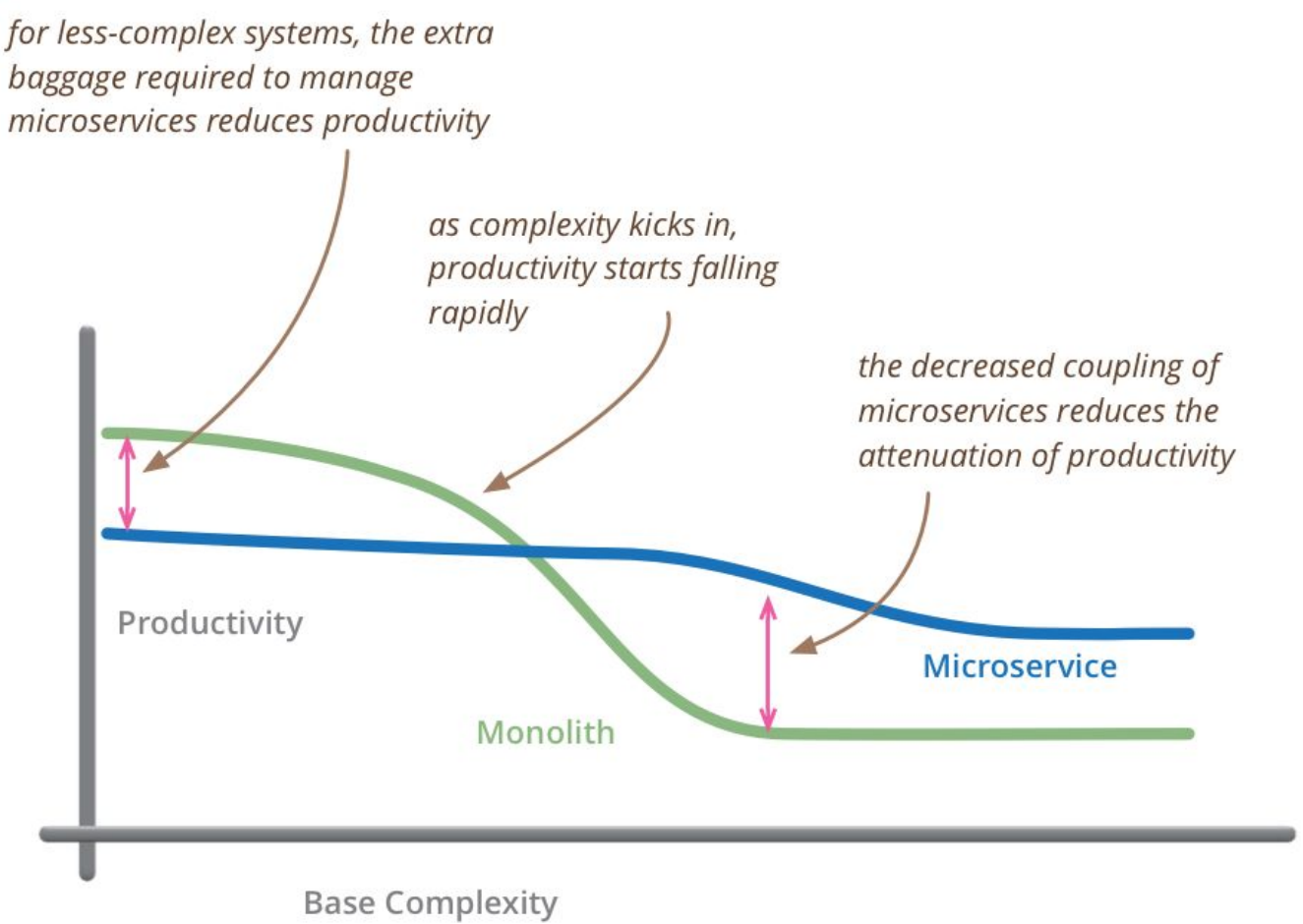
- multiple instances behind load-balancer

Streamline deployment



source: http://martinfowler.com/articles/microservices.html

Overhead



but remember the skill of the team will outweigh any monolith/microservice choice

Software Architecture vs Design Patterns

Design patterns: Composition and interaction of objects

Architectural pattern: System-level structures, subsystems

Architecture often has focus on system qualities as performance, scalability, robustness, security

Typical architectural patterns/styles: client server, microservice, event-based, pipe and filter

This introduces new challenges when 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? How can we make a robust system in light of this?
  - How can we test considering all the different components and dependencies?
  - What if the system is too big to test?

This introduces new challenges when 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? How can we make a robust system in light of this?**
  - How can we test considering all the different components and dependencies?
  - What if the system is too big to test?

What will you do if

- An API your data plugin uses is temporarily down?
  - Or returns a surprising error code?
- Consider: retry
  - How Long? How often?
- But, you still
  - F recovery via a redirect in case of failure
  - If hand work over to proxy
    - data(set), fallback service, caching (e.g., store last Twitter feed)
  - If not, recruit clean-up service
    - Proces, display errors

What if Facebook withdraws its DNS routing information?

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

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

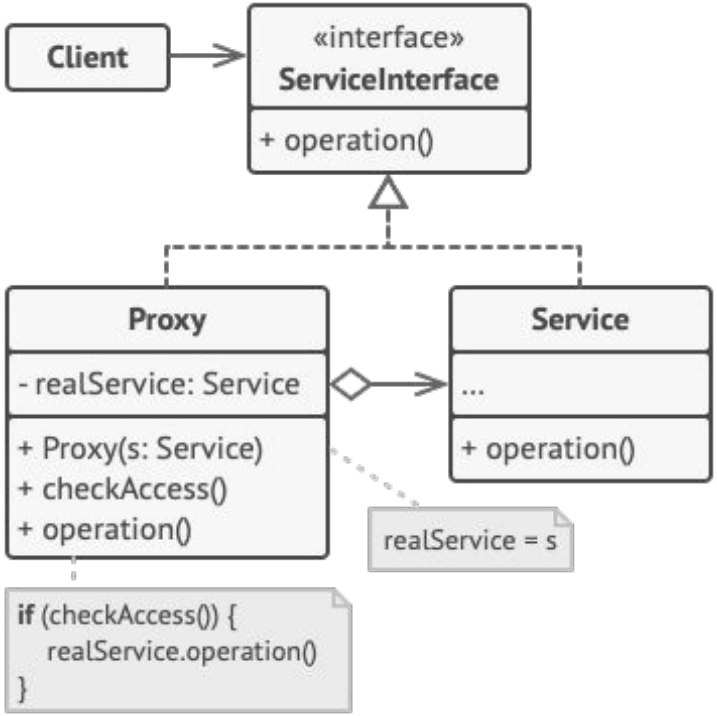
```
const delay = retryCount => new Promise(resolve =>
                    setTimeout(resolve, 10 ** retryCount));

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/

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);
        }
    }
```

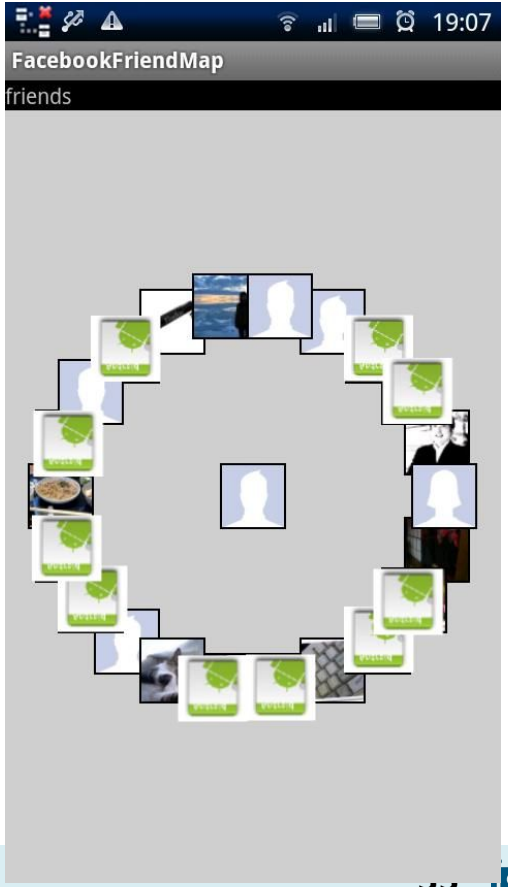
Testing Distributed Systems

- Challenges:
  - Volatility
    - Users are hard to simulate
    - Real-world effects -- things crashing, delays, indicative use/data.
  - 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**

How do we test this?



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
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 other categories exist)
- 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

Eliminating the Remote Service 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;
}
```

Replace by Double

Introducing a Double (Stub)


```
@Test void testGetFriends() {
   assert getFriends() == …;
}
List<Friend> getFriends() {
   Connection c = http.getConnection();
   FacebookInterface api = new FacebookStub(c);
   List<Node> perso
   for (Node person1
       for (Node per
      …
       }
   }
```

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

REST API Calls and Testing

Test happy path

Test also error behavior!

- Correct timeout handling? Correct retry when connection down?
- Invalid response detected?
- Graceful degradation?

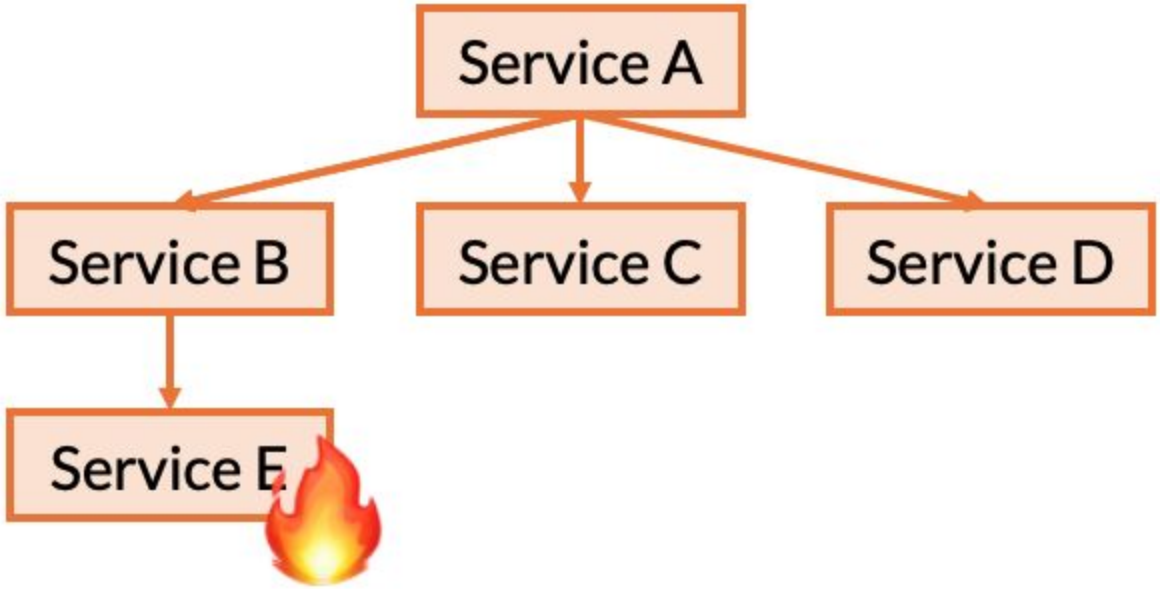
Need to understand possible error behavior first

Fallacies of distributed computing by Peter Deutsch

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

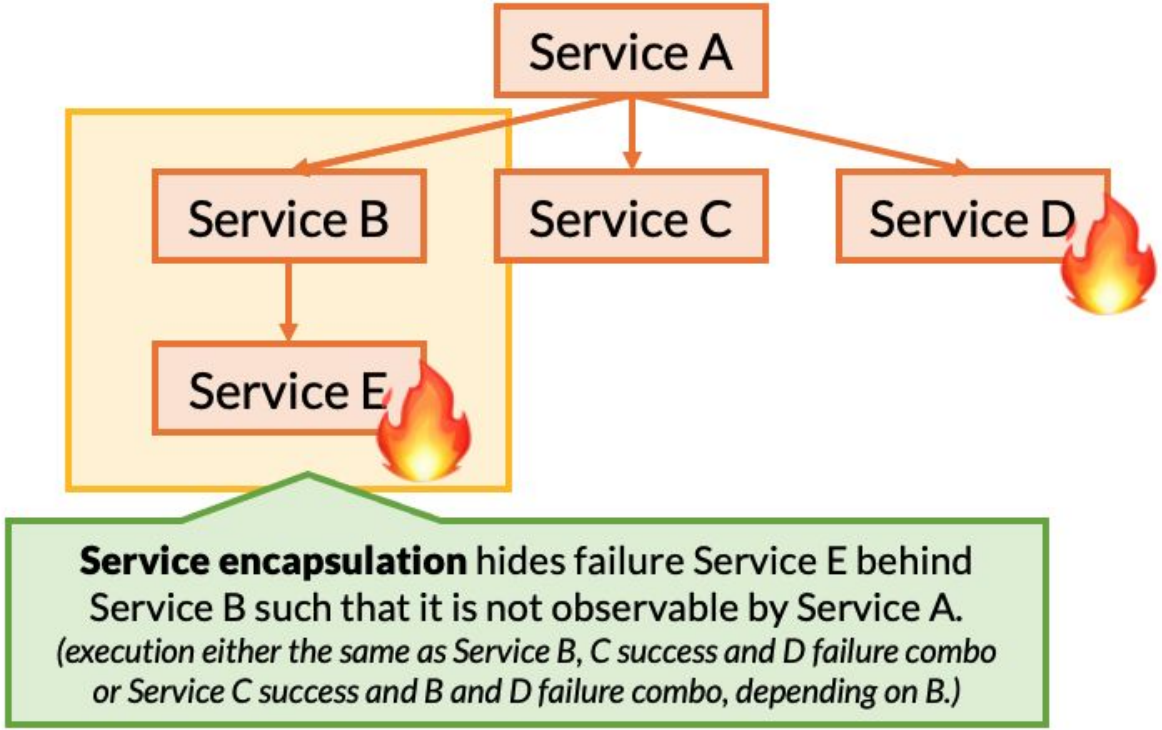
How to test?

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.



http://christophermeiklejohn.com/filibuster/2021/10/14/filibuster-4.html

Handle Errors Locally



http://christophermeiklejohn.com/filibuster/2021/10/14/filibuster-4.html

Fault injection

- Mocks can emulate failures such as timeouts
- Allows you to verify the robustness of system

```
class FacebookSlowStub implements FacebookInterface {
    void connect() {}
    int counter = 0;
    List<Node> getFriends(String name) {
        Thread.sleep(4000);
        if ("john".equals(name)) {
            List<Node> result=new List();
            result.add(…);
```

Fault injection


```
class FacebookErrorStub implements FacebookInterface {
    void connect() {}
    int counter = 0;
    List<Node> getFriends(String name) {
        counter++;
        if (counter % 3 == 0)
            throw new SocketException("Network is unreachable");
        if ("john".equals(name)) {
            List<Node> result=new List();
            result.add(…);
            return result;
```

Chaos Engineering

Experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production

Design: Testability

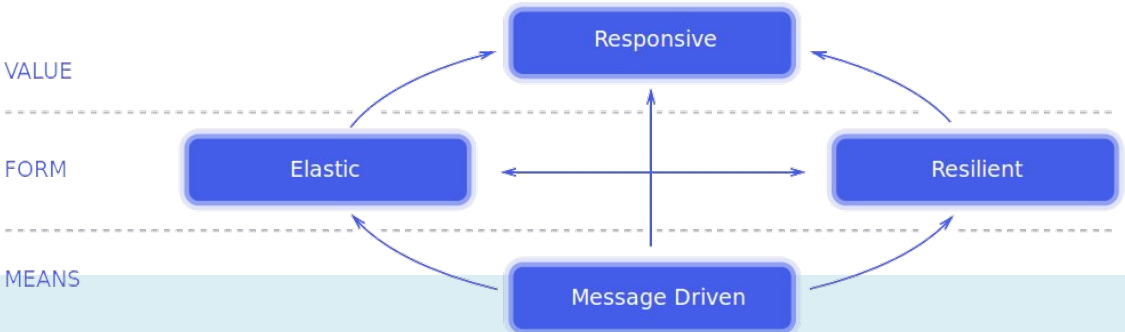
- Single responsibility principle
- Dependency Inversion Principle (DIP)
  - High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.
- Law of Demeter: Don’t acquire dependencies through dependencies.
  - avoid: `this.getA().getB().doSomething()`
- Use factory pattern to instantiate new objects, rather than new.
- Use appropriate tools, e.g., dependency injection or mocking frameworks

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

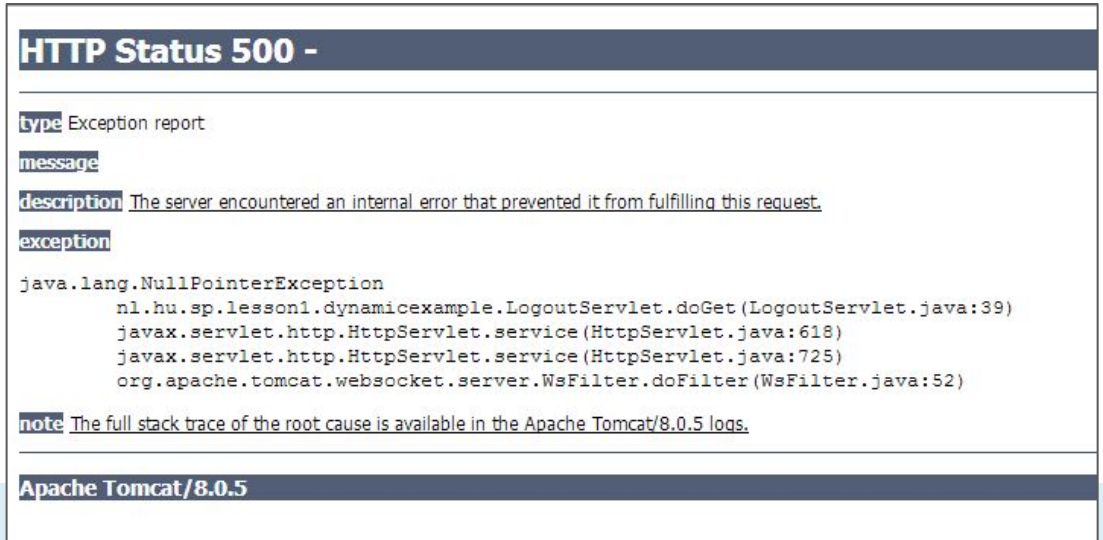
Robust Distributed System Design

- 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:**



HTTP Status 500 -

type Exception report

message

description The server encountered an internal error that prevented it from fulfilling this request.

exception

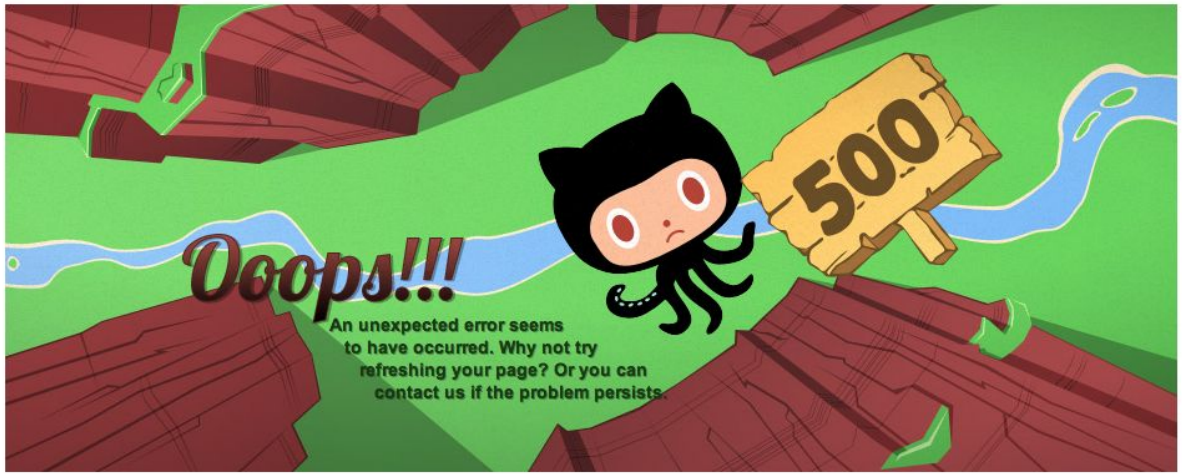
```
java.lang.NullPointerException
        nl.hu.sp.lesson1.dynamicexample.LogoutServlet.doGet(LogoutServlet.java:39)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:618)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:725)
        org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
```

note The full stack trace of the root cause is available in the Apache Tomcat/8.0.5 logs.

Apache Tomcat/8.0.5

Principle: Modular Protection

- Online: use HTTP response status codes effectively
  - Don’t just hand out 404, 500
    - Unless they really apply
  - Provide and document 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
  - Microservices