

# Designing OO and Web APIs

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 39

# Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

- Some bits taken from my EID API lecture
- Some from Joshua Bloch's lecture on API design
  - <http://fwdinnovations.net/whitepaper/APIDesign.pdf>
  - Also see Bloch's Effective Java book
- Martin Fowler – he's back

# What is an API



- An API (Application Programming Interface) is a set of selected functionality defining interactions between a subsystem or a service and their clients
  - It provides an abstraction layer (much like the Façade pattern) to provide only the subsystem or service functionality needed by clients
  - And it provides encapsulation of the details of elements of the subsystem that a client doesn't need to know
- In a firmware system, APIs may be thought of as analogous to Hardware Abstraction Layers (HALs), which
  - Protect higher level elements of the system from having to know details of underlying hardware implementations and
  - Allow that underlying hardware to be changed without severely impacting the implementations above

# Code to an Interface (not an Implementation)

- Again, remember we are not talking specifically about a certain kind of interface here – like a Java Interface
  - We would like the clients of our interfaces to focus on the functionality we expose in the interface
  - We would like code that implements a REST API or a Java Interface (or a set of abstract class methods) to consider just the specific transactions that need to be modeled and provided
  - The interface defines the interaction
- And we'd like to maintain the Interface Segregation Principle
  - Small, cohesive interfaces
  - Reduce the impact and frequency of changes

# Types of APIs

- Certainly, the set of methods provided by a Java class, for instance, can be considered an API
  - What methods are made available
  - What attributes are needed to pass data, etc.
- APIs can also be independently defined using standard Web-based API approaches such as
  - RPC – Remote Procedure Calls
  - REST – REpresentational State Transfer (used in a “RESTful” API)
  - or other RESTless custom command syntax
- In Web-based APIs, both clients and servers would have to reference the defined API to pass data between each other

# Importance of API Design

- APIs can be thought of as a company's asset
  - Customers invest when they use them
  - Once an API is made public, it likely will stay public, important to get it right
  - Bad APIs cause support issues, customer loss
- All programmers are API designers
  - Modular code communicates through interfaces
  - Once a module is in use, it becomes harder to change the API
  - Thinking in terms of interfaces improves code quality
- From How to Design a Good API and Why it Matters - Bloch
- Good APIs
  - Clear naming
  - Easy to learn and use, even without documentation
  - Hard to misuse
  - Using the API makes code easier to read and maintain
  - Can be extended
  - Developed with the audience in mind
- Most API designs are over-constrained
  - You can't allow for all users/usage



**Leon Bambrick**  
@secretGeek



There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors.

7:20 AM · Jan 1, 2010



1.4K



1.4K people are Tweeting about this

# General OO API Principles

- The API should do ONE THING well
  - Easy to recognize by name
    - Standard and self-explanatory names
    - Should make code readable
  - Easy to explain the functionality
- May need to split/merge to get the right fit of functional cohesion
- APIs should be as small as possible
  - Focus on key requirements
  - You can add later, but it's hard to remove once published
- Keep implementation details out of the API
- Minimize accessibility
  - Maximize information hiding
- APIs must be documented
  - Classes: what is an instance representing?
  - Methods: the contract
    - Pre- or post-conditions
    - Side effects
  - Parameters: units, formats
- Use API-friendly elements
  - Generics, varargs, enums, defaults
- From How to Design a Good API and Why it Matters - Bloch

# Fluent Interfaces

Consider...

```
private void makeNormal(Customer customer) {  
    Order o1 = new Order();  
    customer.addOrder(o1);  
    OrderLine line1 = new OrderLine(6, Product.find("TAL"));  
    o1.addLine(line1);  
    OrderLine line2 = new OrderLine(5, Product.find("HPK"));  
    o1.addLine(line2);  
    OrderLine line3 = new OrderLine(3, Product.find("LGV"));  
    o1.addLine(line3);  
    line2.setSkippable(true);  
    o1.setRush(true);  
}
```

Simple to write the constructors, setters, and addition methods, but...



# Fluent Interfaces continued

A fluent version of the same code:

```
private void makeFluent(Customer customer) {  
    customer.newOrder()  
        .with(6, "TAL")  
        .with(5, "HPK").skippable()  
        .with(3, "LGV")  
        .priorityRush();  
}
```

Fowler describes this as almost an internal Domain Specific Language, but certainly it makes for an API that's readable and flows, but it takes more work to design and implement

<https://martinfowler.com/bliki/FluentInterface.html>

# Minimal vs. Humane Interfaces

- Minimal Interface: an API that allows the client to do everything they need to do, but keeps the capabilities at the smallest reasonable set of methods that will do the job
- Humane Interface: Find out what people want to do and design the interface to make it easy to do the common case
- Getting the last item from a list:
- In Java

```
aList.get(aList.size-1)
```
- In Ruby

```
anArray.last
```

(Ruby also has a `.first`)

<https://martinfowler.com/bliki/HumaneInterface.html>

# Java – Service Provider Interfaces

- SPI (Service Provider Interfaces) - A method of formatting classes as services for clients or consumers
  - Creates an easy plug-in interface for multiple implementations
  - Structures a jar file with the SPI elements
- To access an SPI service, you drop the SPI jar file on the classpath for your application
- <https://docs.oracle.com/javase/tutorial/sound/SPI-intro.html>
- Four elements to a Java SPI
  - Service
    - Interfaces and classes for accessing functionality
  - SPI
    - Interface or abstract class to reach the service
  - Service Provider
    - A concrete implementation of the SPI
  - ServiceLoader
    - Provides lazy loading of implementations as needed, using an internal cache
- Standard Java SPIs
  - TimeZoneNameProvider
  - JsonProvider
- <https://www.baeldung.com/java-spi>

# Web-based APIs

- RPC
- RESTful
- RESTless/Custom

# RPC-based APIs

- Web-based RPC is generally characterized as a single URI (Uniform Resource Identifier) on which many operations may be called, usually solely via an HTTP POST (a REST command)
- Examples include
  - XML-RPC (<http://xmlrpc.com/>) →
  - SOAP (Simple Object Access Protocol) (<https://www.w3.org/TR/soap/>)
- You pass a structured request that includes the operation name to invoke and any arguments you wish to pass to the operation; the response will be in a structured format

POST /xml-rpc HTTP/1.1

Content-Type: text/xml

```
<?xml version="1.0" encoding="utf-8"?>
<methodCall>
  <methodName>status.create</methodName>
  <params>
    <param>
      <value><string>First post!</string></value>
    </param>
    <param>
      <value><string>mwop</string></value>
    </param>
    <param>
      <value><dateTime.iso8601>20140328T15:22:21</dateTime.iso8601>
    </value>
    </param>
  </params>
</methodCall>
```

# Considerations of RPC-based APIs

- Advantages

- One service endpoint, many operations
- One service endpoint, one HTTP method (usually POST)
- Structured, predictable request format, structured, predictable response format
- Structured, predictable error reporting format
- Structured documentation of available operations

- Disadvantages

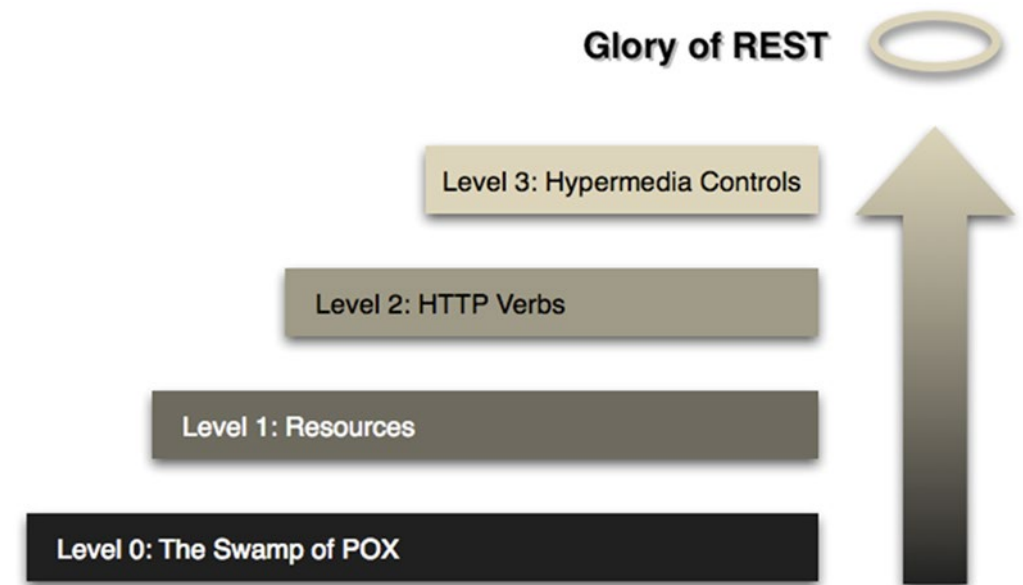
- You cannot determine via the URI how many resources are available
- Lack of HTTP caching, inability to use native HTTP verbs for common operations; lack of HTTP response codes for error reporting requires introspection of results to determine if an error occurred
- "One size fits all" format can be limiting; clients that consume alternate serialization formats cannot be used, and message formats often impose unnecessary restrictions on the types of data that can be submitted or returned

- <https://apigility.org/documentation/api-primer/what-is-an-api>

# RESTful/REST-based APIs

- Most common Web-based API approach in modern systems
- REST leverages HTTP's strengths, and builds on URIs as unique identifiers for resources
- Rich set of HTTP verbs for operations on resources (PUT, GET, POST, DELETE, PATCH, OPTIONS, HEAD, others)
- Clients can specify representation formats they can render, and request those from the server (if the server can provide)
- Linking between resources to indicate relationships (e.g., hypermedia links, such as those found in plain old HTML documents)
- REST API implementations can vary in complexity and completeness
- <https://apigility.org/documentation/api-primer/what-is-an-api>

- 4 Levels of the Richardson Maturity Model (yes, another Fowler article)
  - <https://martinfowler.com/articles/richardsonMaturityModel.html>



- POX = Plain Old XML transfers
- Hypermedia controls – more dynamic command structures for enhanced resource provision

# A good RESTful API

- A good REST API
  - Uses unique URIs for services and the items exposed by those services
  - Uses the full spectrum of HTTP verbs to perform operations on those resources, and the full spectrum of HTTP to allow varying representations of content, enabling HTTP-level caching, etc.
  - Provides relational links for resources, to tell the consumer what can be done next
- Design questions for REST APIs
  - REST does not dictate any specific formats
  - What representation formats will you provide, how will you report failed requests?
  - REST does not dictate any specific error reporting format (suggests using standard HTTP Codes)
  - Which HTTP methods will be available for a given resource; how do you respond to a failed request?
  - Authentication – web APIs are generally stateless, and should not rely on features like session cookies
  - Credentials – HTTP authentication, or OAuth2, or create API tokens?



# RESTful API Design Best Practices

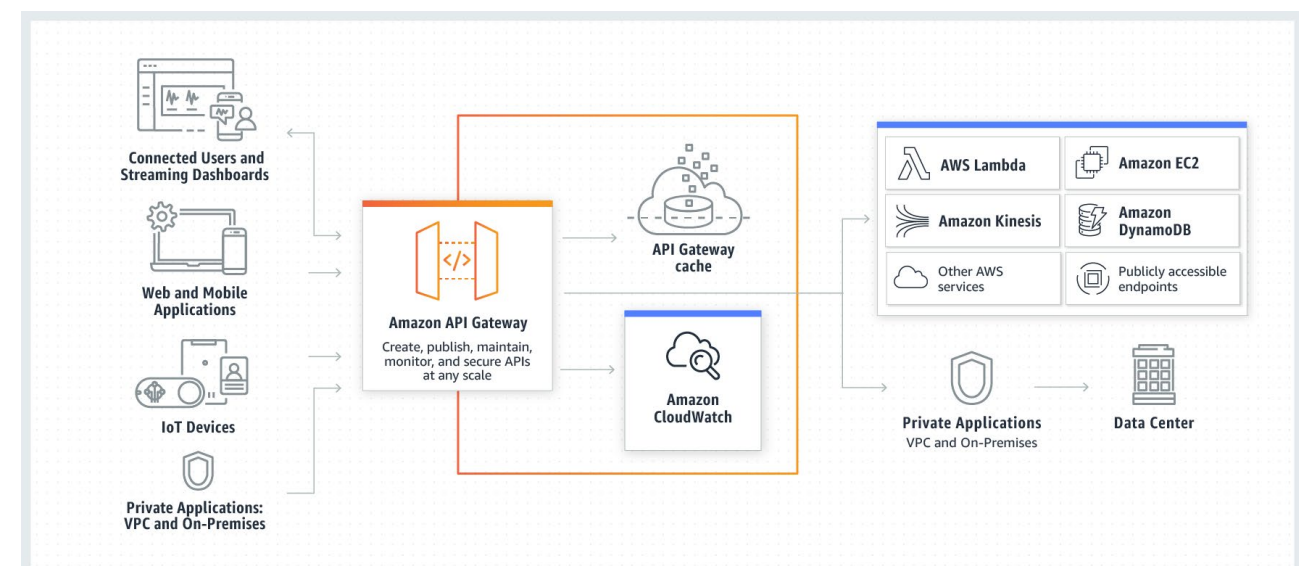
- Documentation (->SDK)
  - API, Endpoints, Functions (often automatically generated) – technical reference
  - Usage examples and tutorials – first steps for new users, how-tos, examples of typical operations
  - Validate the documentation with outside users
- Stability and Consistency
  - Include a version number in the API from the beginning
  - Ok to maintain version for additions, but not for changes
  - Internal consistency – all functions should work the same way
  - Publicize changes, changelogs, and document difference in versions
- Flexibility
  - Allowing for easily identifying supported output formats: JSON, YAML, XML
  - Ex: /api/v1/widgets.json
  - Allowing for multiple standard input formats
- Security
  - Use standard approaches
  - Token-based authentication (SHA-1 token)
  - SSL & OAuth2
  - Identify whitelisted (allowed) and blacklisted (security required) functions
  - Verify/validate access to key resources
  - Validate all input
- Ease of Adoption
  - Have others try to program/connect to the API to verify ease of use and operations
  - Use standard approaches, JSON, SOAP, REST, etc.
  - Provide language specific examples and/or libraries
  - Tools for generating libraries from APIs
    - Alpaca: Node.js, Python, Ruby, PHP
    - Apache Thrift: Node.js, Python, C++, Java, C#, many others
  - Provide for simple sign up, support and bug reporting
- <https://www.toptal.com/api-developers/5-golden-rules-for-designing-a-great-web-api>

# Tools for RESTful API Design

Nice basic RESTful API example for Python:

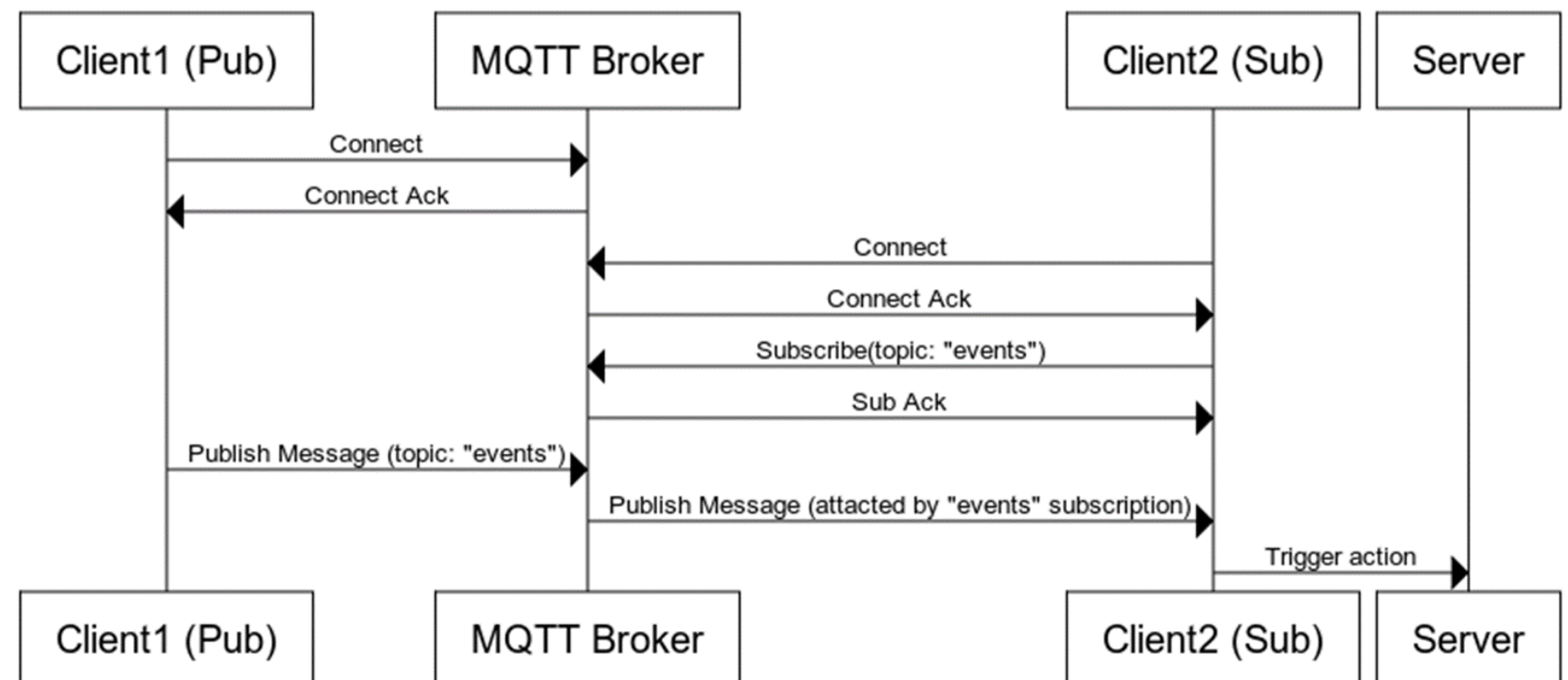
<https://realpython.com/api-integration-in-python/>

- Swagger (<https://swagger.io/>)
  - Open source and commercial
  - OpenAPI 3.0 specification
- Open source elements include Swagger Editor, Codegen, and UI for documentation
- Easy to create the definition of the API and then generate
  - Server stubs
  - Client SDKs
  - Documentation
- Try here:
  - <https://editor.swagger.io/>
- RAML (<https://raml.org/>)
  - Open source
  - Design, build, test, document, and share APIs
- AWS API Gateway
  - <https://aws.amazon.com/api-gateway/>
  - Typical of API tools from Cloud services



# Example of a Custom API - MQTT

- Commands: Connect, Publish, Subscribe, Unsubscribe, Disconnect
- Topics – Specially formatted labels for grouping of application messages, matched against subscriptions to forward the messages
- Publish messages under specific topics
- Subscribe/Unsubscribe to/from Topic filters
- Built on top of TCP/IP but not RESTful



# Summary

- APIs and Interfaces, broadly considered
- Best practices for interface designs
- APIs for OO language classes/methods/attributes
  - Fluent, minimal, humane interfaces
  - Java SPIs
- Web-based APIs
  - RPC
  - REST
  - Custom

# Next Steps

- Project 6 due 12/4 Noon
  - Final part of three for the semester project
  - Get started sooner than later!
  - Be aware of how your travel plans may impact your team and turning in your assignments!
- Quiz 9 up, due today, Quiz 10 this weekend
  - Quiz 10 is last topic quiz, Quiz 11 will be a class survey for your feedback
- Graduate Final Presentations are due Monday Dec 7
- Graduate Pecha Kuchas are due Tuesday Dec 1
  - **Sign up for a presentation slot here:**
    - <https://docs.google.com/document/d/1BYxUTTIh66DLLhVU8GbDZm94H2JtdMYXylrN7vPaWh8/edit?usp=sharing>
- Article Reviews are available for extra bonus points...
- New discussion topic coming up... Visit Piazza often – it is for your participation grade, so participate!
- Coming up: Antipatterns and other patterns, What is not OOAD, graduate Pecha Kuchas, more...
- If you need help – Office hours, Piazza, e-mail – we are here for you!