APIs and Microservices

Embedded Interface Design

with **Bruce Montgomery**

Learning Objectives

- Students will be able to…
  - Understand the use of APIs in Embedded Development
  - Recognize popular API styles and tools
  - Understand microservice architecture and relation to APIs

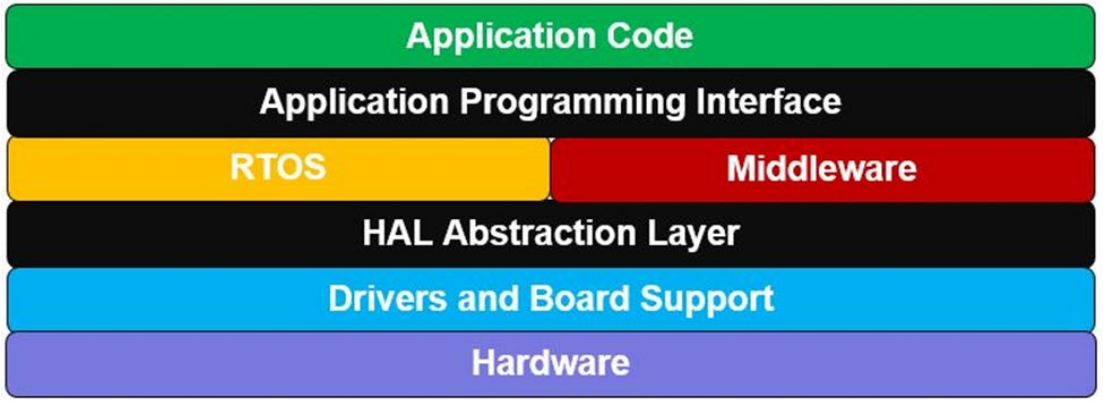
Why APIs for Embedded Interfaces?

- In practice, an Embedded Device may be created without any user interface, OR
- Not all functions provided by the device may be available in its user interface – so to…
  - Configure the device
  - Diagnose issues
  - Provide inputs
  - Gather logs or output data
  - Use extended functions
  - Connect to other systems
- …it may be necessary to connect to the device via a program or Web-based API – Application Programming Interface – defined for that purpose.

Reference [1]

APIs for Embedded Devices

- APIs for embedded systems generally provide a way for an external application to communicate to the firmware running on the device hardware
- Much like a Hardware Abstraction Layer (HAL) prevents the firmware for an RTOS or other middleware/ firmware from having to know details of the hardware implementation, the API provides an abstract and simplified view of commands to access exposed device functionality
- Generally, APIs for devices are based on RPC-based (Remote Procedure Calls) or more commonly, REST-based (REpresentational State Transfer) architectures



Reference [2]

RPC-based APIs

- RPC is generally characterized as a single URI on which many operations may be called, usually solely via POST.
- Examples include XML-RPC [3] and SOAP (Simple Object Access Protocol) [4]
- 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>
```

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.

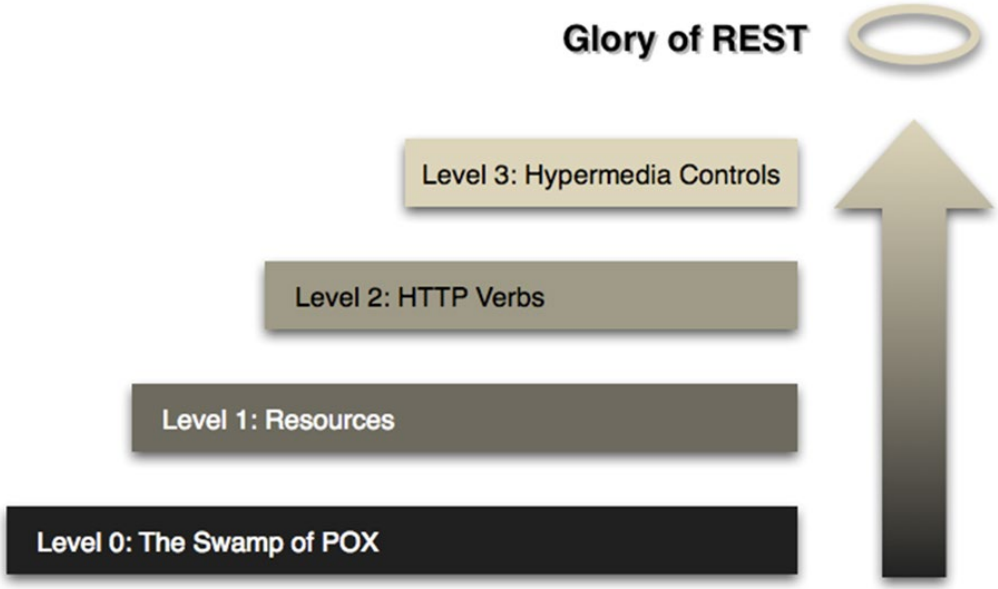
Disadvantages

- 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

Reference [5]

REST-based APIs

- Most common 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
- 4 Levels of the Richardson Maturity Model



Reference [6], [5]

REST-based APIs

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

Reference [5]

REST-based APIs

Design Questions

- REST does not dictate any specific formats: Which representation formats will you provide? How will you report that you cannot fulfill requests?
- REST does not dictate any specific error reporting format (suggests using standard HTTP Codes): How will you report errors?
- Which HTTP methods will be available for a given resource? What will you do if the consumer uses a request method you do not support?
- How will you handle Authentication? Web APIs are generally stateless, and should not rely on features like session cookies; how will consumers provide credentials? Will you use HTTP authentication, or OAuth2, or create API tokens?
- How will you handle hypermedia linking? XML has linking built-in; others, such as JSON, have no native format for links.

Reference [5]

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 (UX!)
- 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

Reference [7]

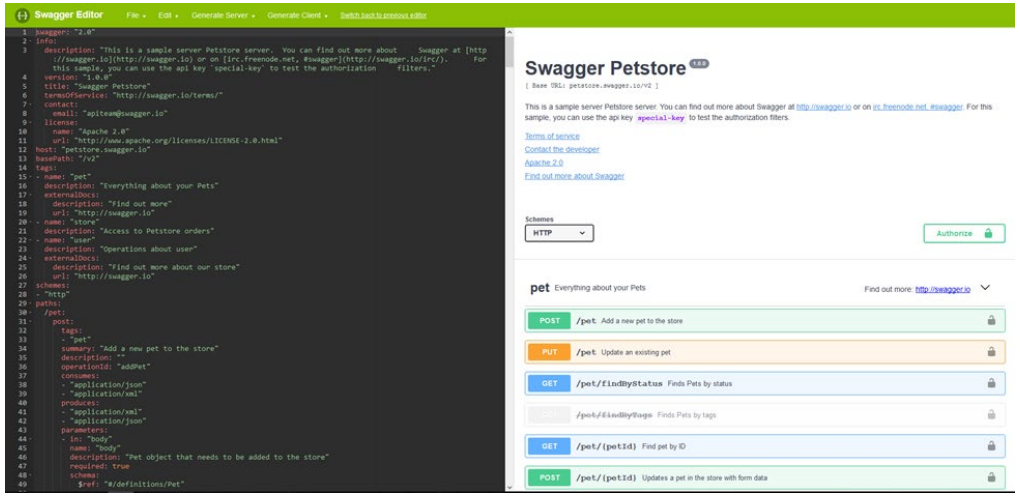
API Design Best Practices

- 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 (UX!)
  - 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

Reference [7]

Tools for API Design - Swagger

- Swagger [8]
- OpenAPI Specification (2.0 and 3.0.0) – standard for defining/describing a language-agnostic RESTful JSON-based API [9]
- Swagger Editor – Design or Edit REST APIs (GET, PUT, POST, DELETE)
- Swagger Codegen – Generates server stubs and client SDKs from a Swagger specification – support for many languages: Python, Node.js, C++, Java, C#, many others.
- Swagger UI – automatic generation of documentation
- Swagger JS – Javascript library for connecting to Swagger APIs from browser or Node.js apps
- Swagger Node – Design-driven server implementation for Node.js
- SwaggerHub – lets you collaborate on API designs
- Others…



Alternative: RAML (RESTful API Modeling Language) (1.0) [10]

Other Tools for API Design

- Note: Many of these tools have limited free functionality
- Cloud API Tools:
  - AWS: Amazon API Gateway [11]
  - Google API Design Guide [12]
  - Azure API Management
  - IBM Bluemix (API Connect, API Management, Connect & Compose)
- API Design
  - Restlet Studio (Web IDE for API Design, supports Swagger and RAML) [13]
  - Mulesoft [14]
- Load Testing – Loader.io; BlazeMeter
- Testing APIs – Httpbin.org; APImetrics; Runscope
- Mocking APIs – JsonStub; Mockable.io

Reference [15]

Microservices - Defined

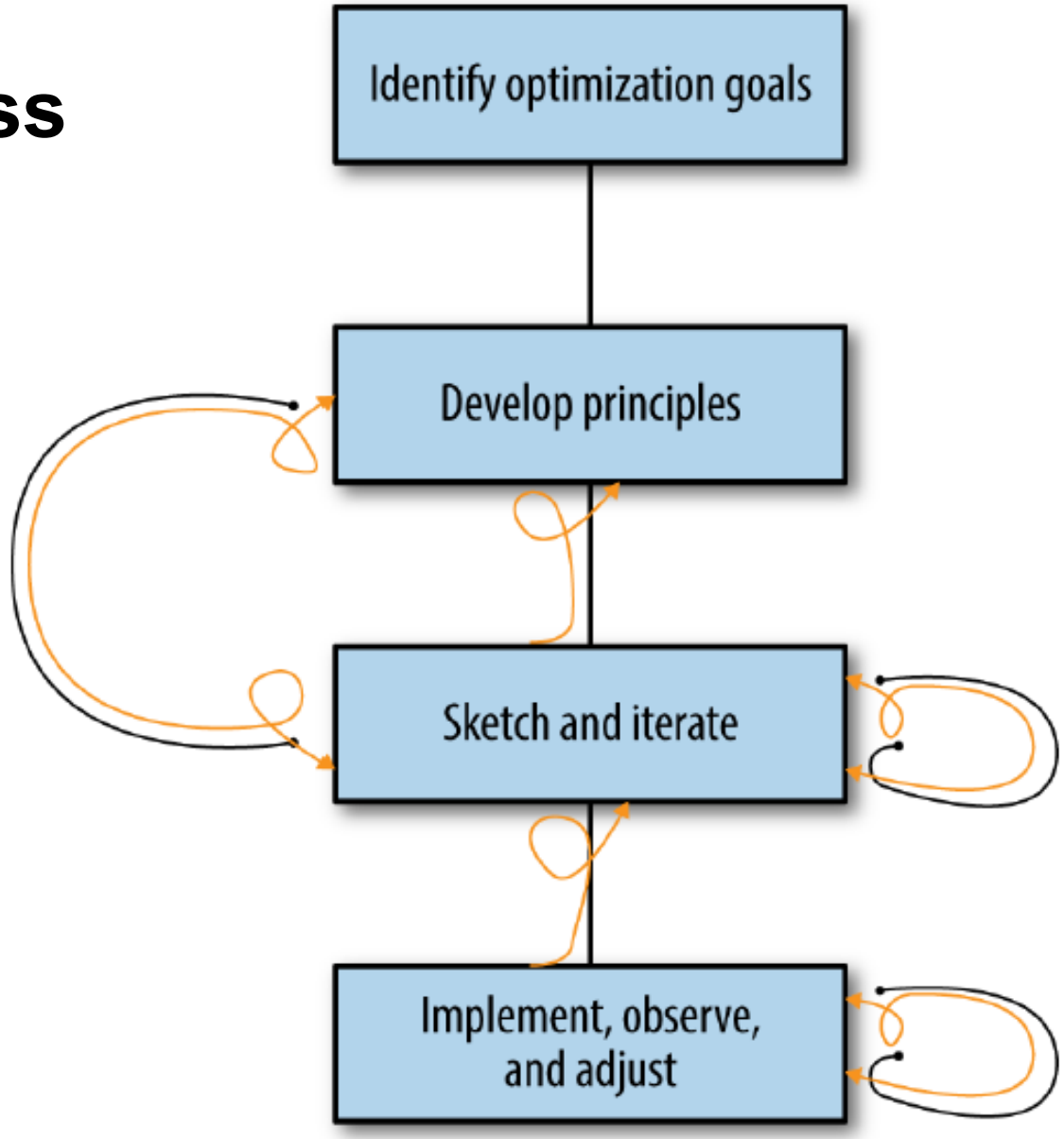
- Microservice – an independently deployable component of bounded scope that supports interoperability through message-based communication
- Microservice architecture - a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices
- Focus on being Replaceable vs. Maintainable
- Usually targeted at big systems and their performance
- Goal-oriented architecture
- Drawn from service-oriented architectures (SOA) for modularity and messaging, DevOps for automated deployment, and Agile for responding to identified needs as they arise
- From [16]

Microservices Application Characteristics

- Individual applications are:
- Small in size
  - Messaging enabled
  - Bounded by contexts
  - Autonomously developed
  - Independently deployable
  - Decentralized
  - Built and released with automated processes
- Goal – balance speed, safety, and scale
- Reference [16]

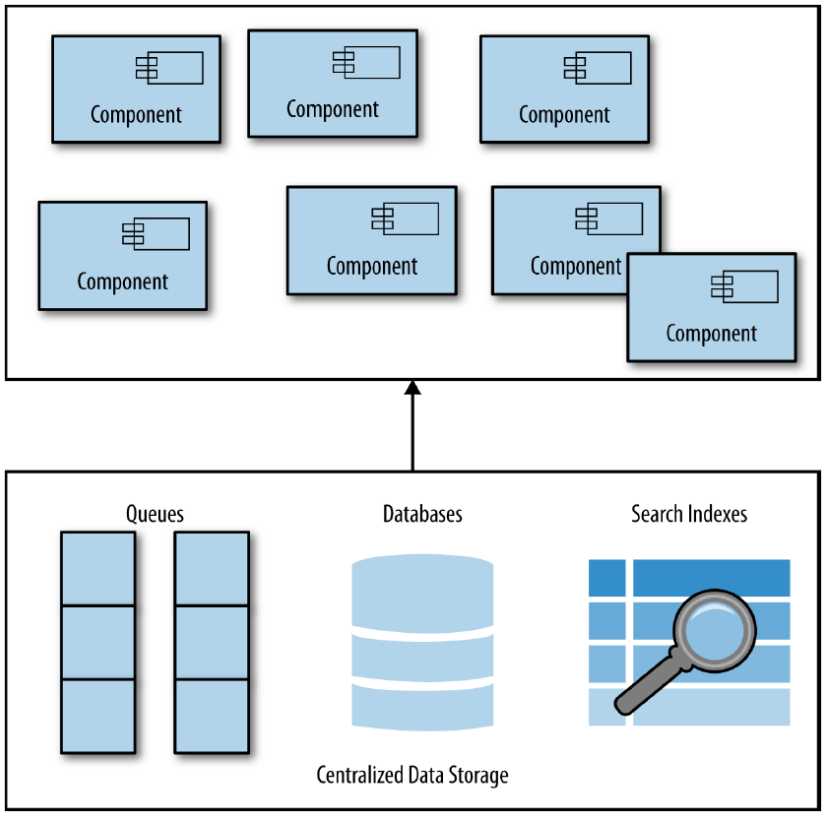
Microservices Design Process

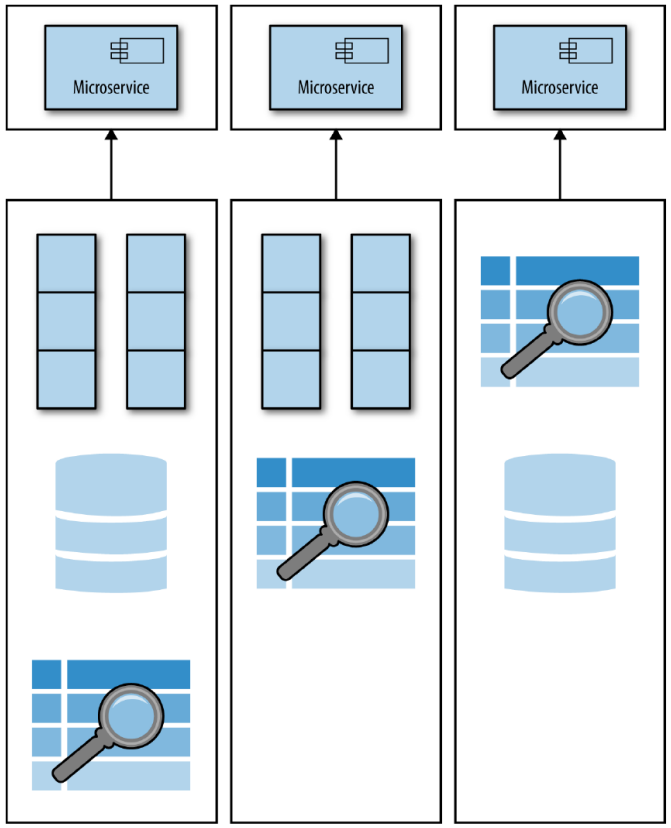
- Iterative process for
  - Determining goals and developing principles
  - Through observation and design cycles
- Microservices should embed their dependencies to make them independent and deployable
- Keys are API designs and messaging approaches
- Reference [16]

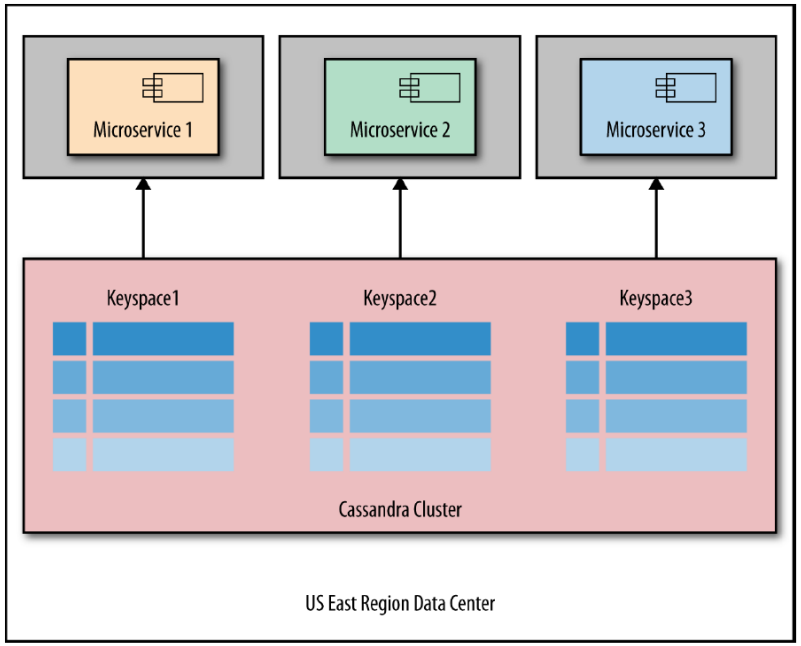


Microservices Example

From one large monolithic application -> to a decomposed set of microservices -> to a pragmatic implementation



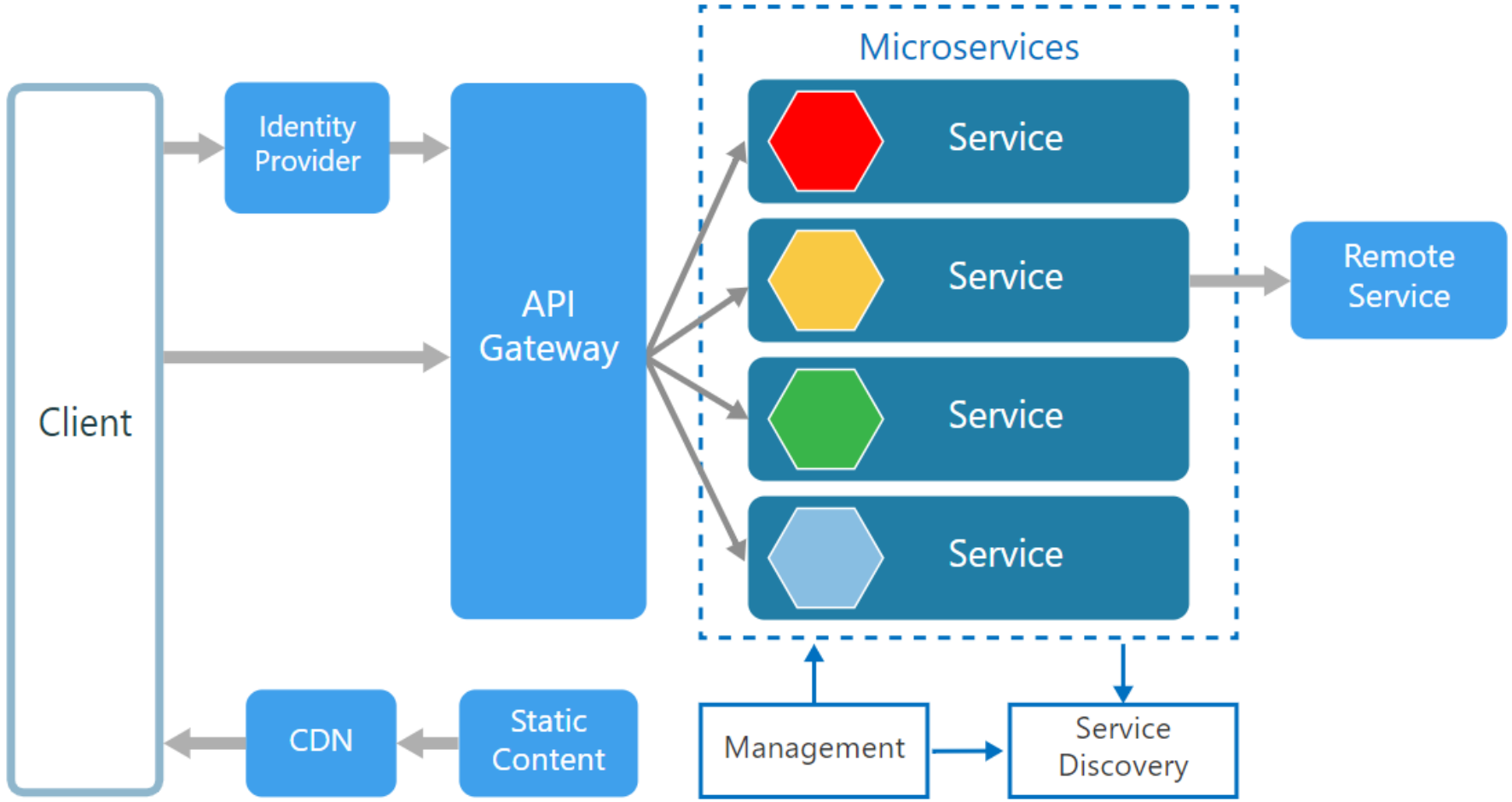




Reference [16]

Microservices – Key API Requirements

- Standard API Gateway
- Containers (e.g. Docker)
- Service Discovery
- Standard Security
- Routing
- Monitoring and Alerting
- Observation: Easily applicable to IoT-to-Cloud Architectures



Reference [17]

Next Steps

- Project 2 due Monday 10/7 before class
  - Please bring your systems into class for demonstrations
  - Make sure you get your AWS accounts!
- Quiz due Friday at class time, new Quiz up Fri/Sat – due in a week
- Next week: Start looking at UX and UI design
- Next week: We'll look at the Superproject and your options for proposing one
- EID Midterm is Wednesday 10/16 in class
- Class staff available to help
  - Shubham - Tues 12-2 PM, Fri 3-5 PM in ECEE 1B24
  - Sharanjeet - Tues 2-3 PM, Thur 2-3 PM in ECEE 1B24
  - Bruce - Tue 9:30-10:30 AM, Thur 1-2 PM in ECOT 242

SwaggerHub demo

- Hello World
- IoT Device

AWS Serverless Microservice w/REST API

- Serverless Microservice with REST API - Demonstration
  - AWS API Gateway <-> AWS Lambda <-> AWS Dynamo DB
- Create a Lambda function using a blueprint for a microservice http endpoint
- Create a security Role with simple microservice permissions
- Create a new API in AWS API Gateway
- Create a table in AWS DynamoDB
- Issue test PUT and GET REST commands using Lambda tests
- See https://docs.aws.amazon.com/lambda/latest/dg/with-on-demand-https-example-configure-event-source_1.html

References

[1] https://www.ecobee.com/home/developer/api/introduction/index.shtml
[2] https://www.beningo.com/embedded-basics-apis-vs-hals/
[3] http://www.xmlrpc.com/
[4] http://www.w3.org/TR/soap/
[5] https://apigility.org/documentation/api-primer/what-is-an-api
[6] https://martinfowler.com/articles/richardsonMaturityModel.html
[7] https://www.toptal.com/api-developers/5-golden-rules-for-designing-a-great-web-api
[8] https://swagger.io/
[9] https://swagger.io/specification/
[10] https://raml.org/
[11] https://aws.amazon.com/api-gateway/
[12] https://cloud.google.com/apis/design/
[13] https://studio.restlet.com/apis/local/info
[14] https://www.mulesoft.com/platform/api/anypoint-designer
[15] https://www.infoworld.com/article/3060731/apis/10-free-tools-for-api-design-development-and-testing.html#slide1
[16] Microservice Architecture, Nadareishvili et al., 2016, O'Reilly
[17] https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices