API Gateway

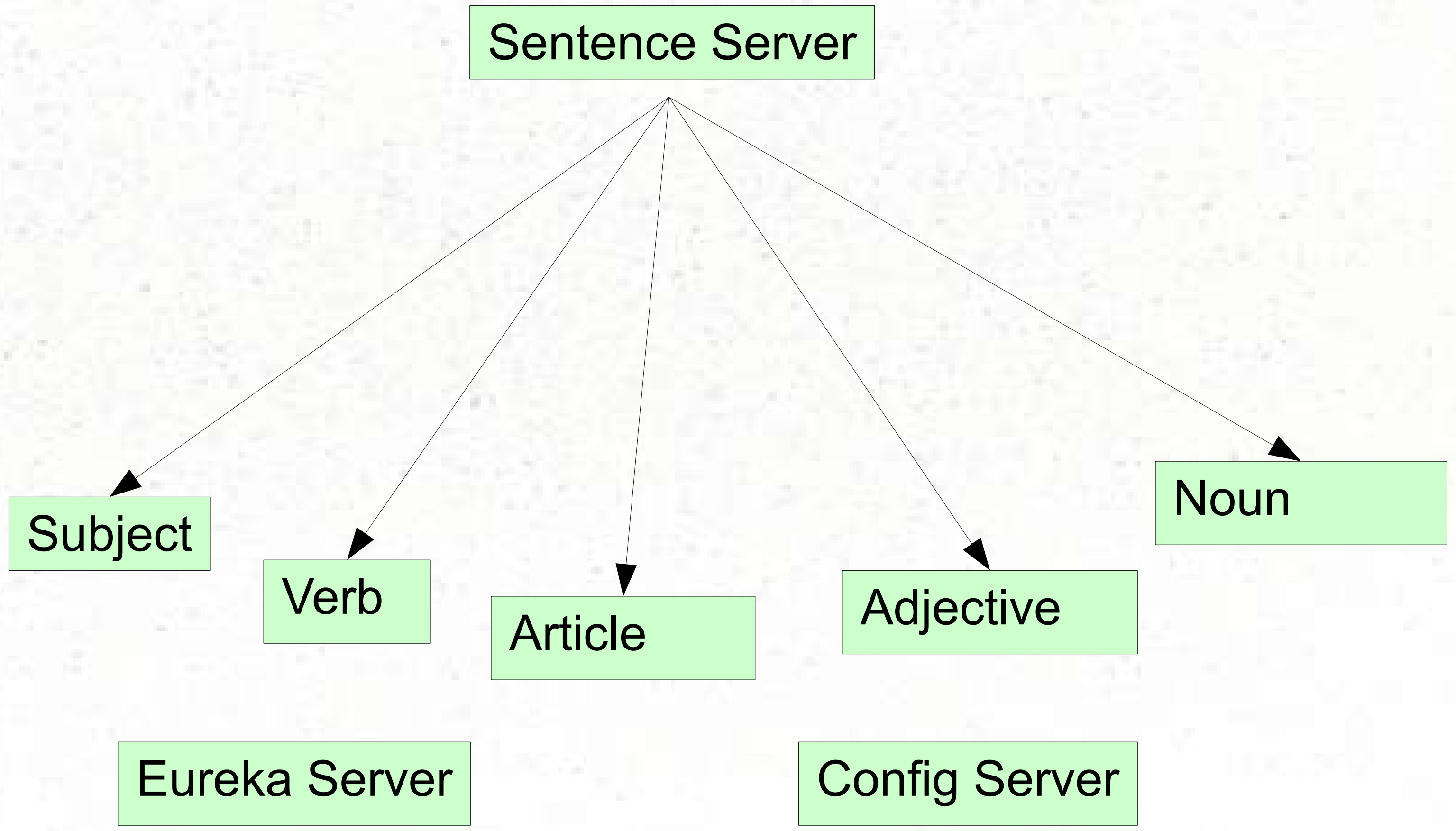
Understanding and Implementing
an API Gateway
for efficient client access

Module Outline

- **The Need for API Gateway**
- Spring Cloud Netflix Zuul
- Caching
- Resource Expansion
- Protocol Translation

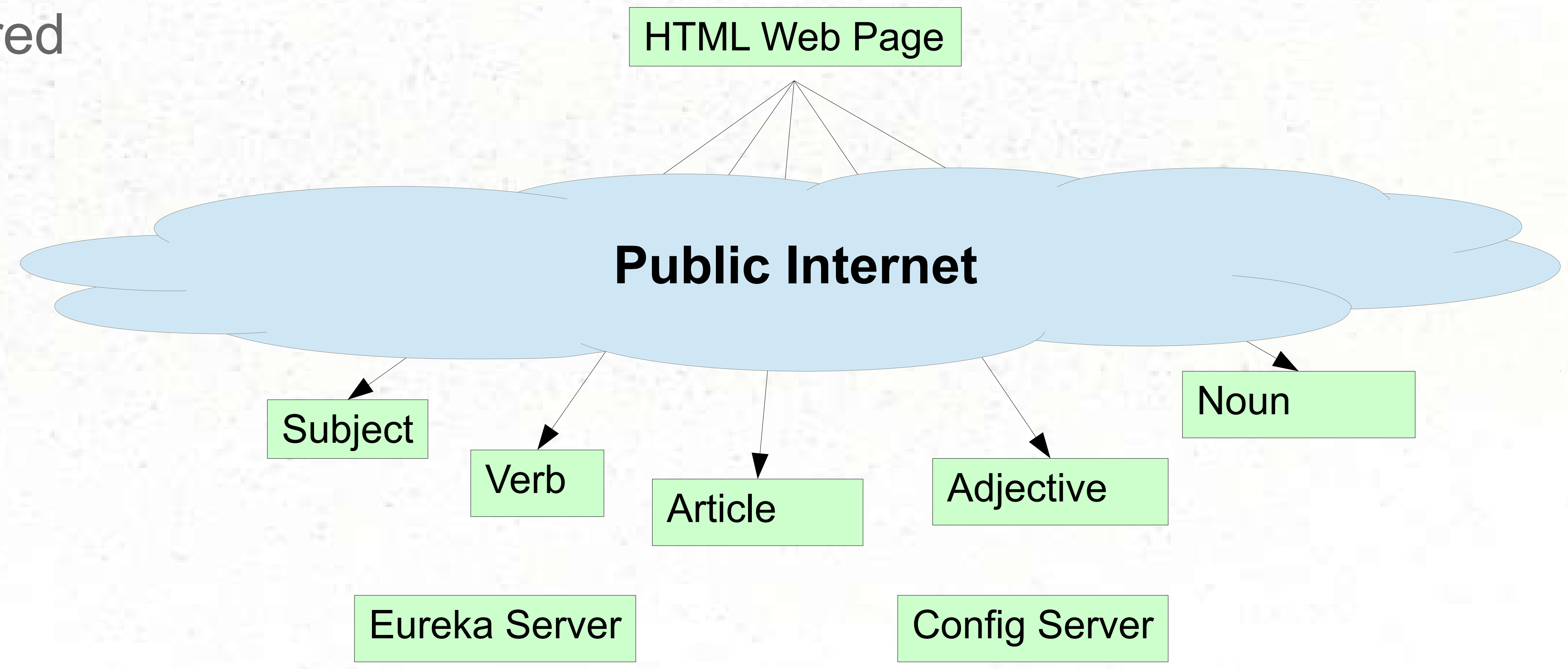
The Current System

- Existing Application
  - Runs fine, within a reliable, high-speed, secure network



Accessing Microservices Via Web

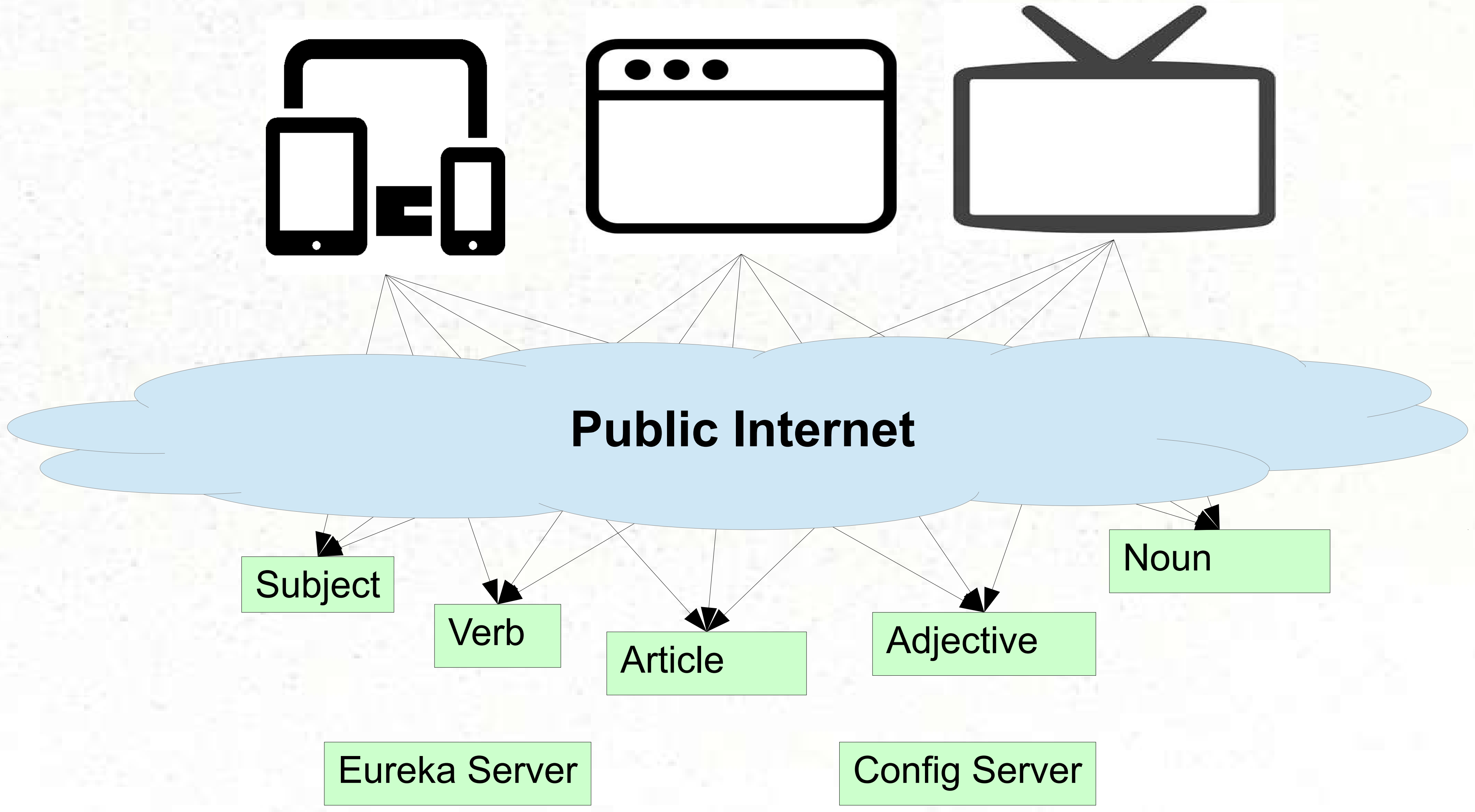
- Accessing over public internet problematic
  - Internal API exposed
  - Security
  - CORS Required
  - Multiple Trips
  - Etc.



CORS – Cross Origin Resource Sharing

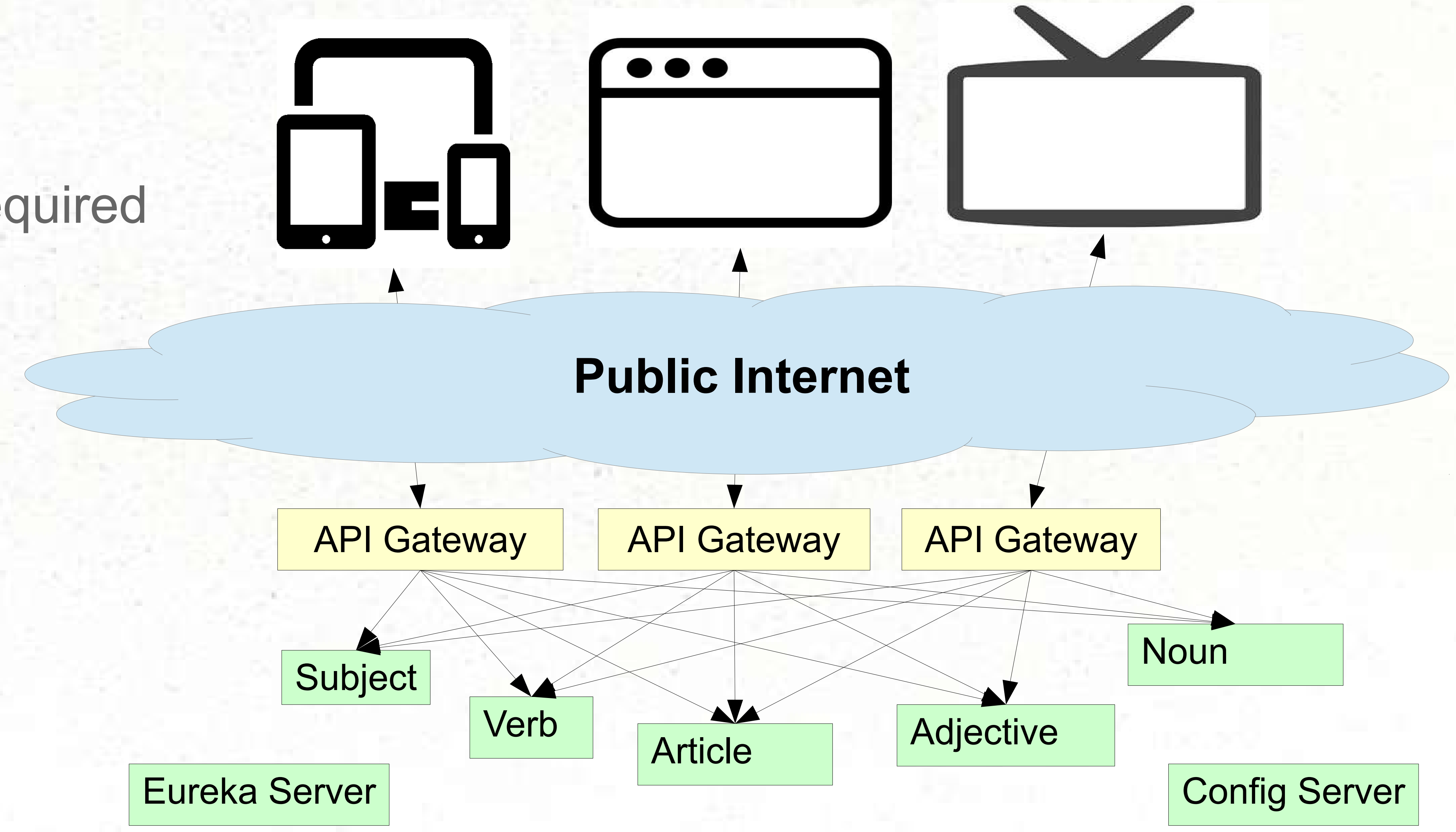
Accessing Microservices Via Web

- Different Clients Have Different Needs



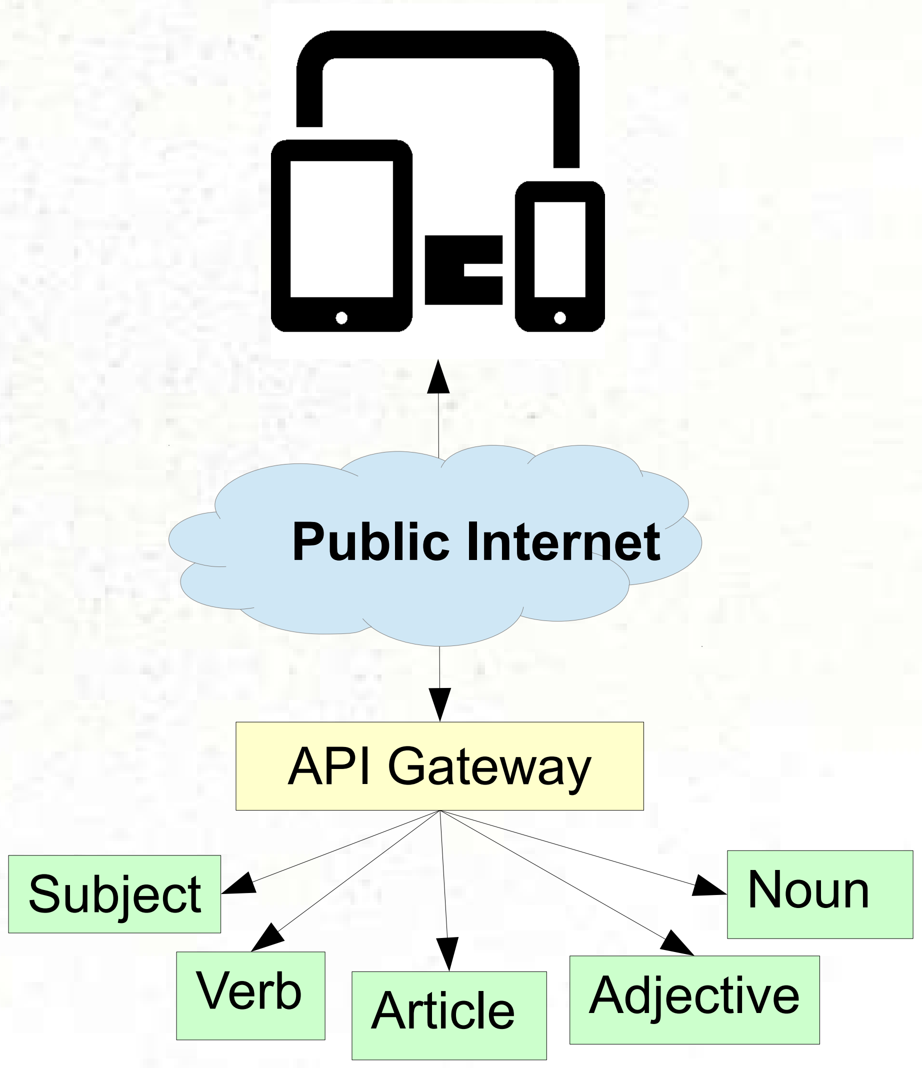
API Gateway

- API Gateway provides simplified access for client
  - Custom API
  - Security
  - No CORS Required
  - Fewer Trips
  - etc.



API Gateway

- Built for specific client needs ("facade")
- Reduces # remote calls
- Routes calls to specific servers
- Handles Security / SSO
- Handles caching
- Protocol Translation
- Optimizes Calls / Link Expansion



Module Outline

- The Need for API Gateway
- **Spring Cloud Netflix Zuul**
- Caching
- Resource Expansion
- Protocol Translation

Zuul – Routing and Filtering

- Zuul – JVM-based router and Load Balancer
  - Can be used for many API Gateway needs
  - Routing – Send request to real server
    - Reverse Proxy

Zuul Basic Usage

- Dependencies: (spring-cloud-starter-zuul)
  - Includes Ribbon and Hystrix
- Annotation: @EnableZuulProxy
- Default Behavior:
  - Eureka client ids become URIs
    - /subject routes to the "subject" service
    - /verb routes to the "verb" service
    - Etc.

Zuul Basic Usage

```
Mapped URL path [/article/**] onto handler [class org.springframework.cloud.netflix.zuul.web.ZuulController]
Mapped URL path [/verb/**] onto handler [class org.springframework.cloud.netflix.zuul.web.ZuulController]
Mapped URL path [/subject/**] onto handler [class org.springframework.cloud.netflix.zuul.web.ZuulController]
Mapped URL path [/adjective/**] onto handler [class org.springframework.cloud.netflix.zuul.web.ZuulController]
Mapped URL path [/noun/**] onto handler [class org.springframework.cloud.netflix.zuul.web.ZuulController]
```

Log Output / Console

Client can call:		...And Zuul will call:
localhost:8080/subject/	→	localhost:59334/
/verb/	→	localhost:54232/
/article/	→	localhost:53452/
/adjective/	→	localhost:45732/
/noun/	→	localhost:44232/

Zuul Features

- Services can be exluded: zuul.ignored-services
- Add a prefix: zuul.prefix: /api
  - /api/subject, /api/verb
- URL can be adjusted:

```
---
zuul:
  prefix: /api
  ignored-services: verb
  routes:
    subject:
      path: /sentence-subject/**
    noun:
      path: /sentence-noun/**
```

Result:
localhost:8080/api/sentence-subject/
~~/api/verb/~~
/api/article/
/api/adjective/
api/sentence-noun/

Is Zuul an API Gateway?

What's Missing?

- Zuul is a tool for creating an API Gateway
  - Specifically routing

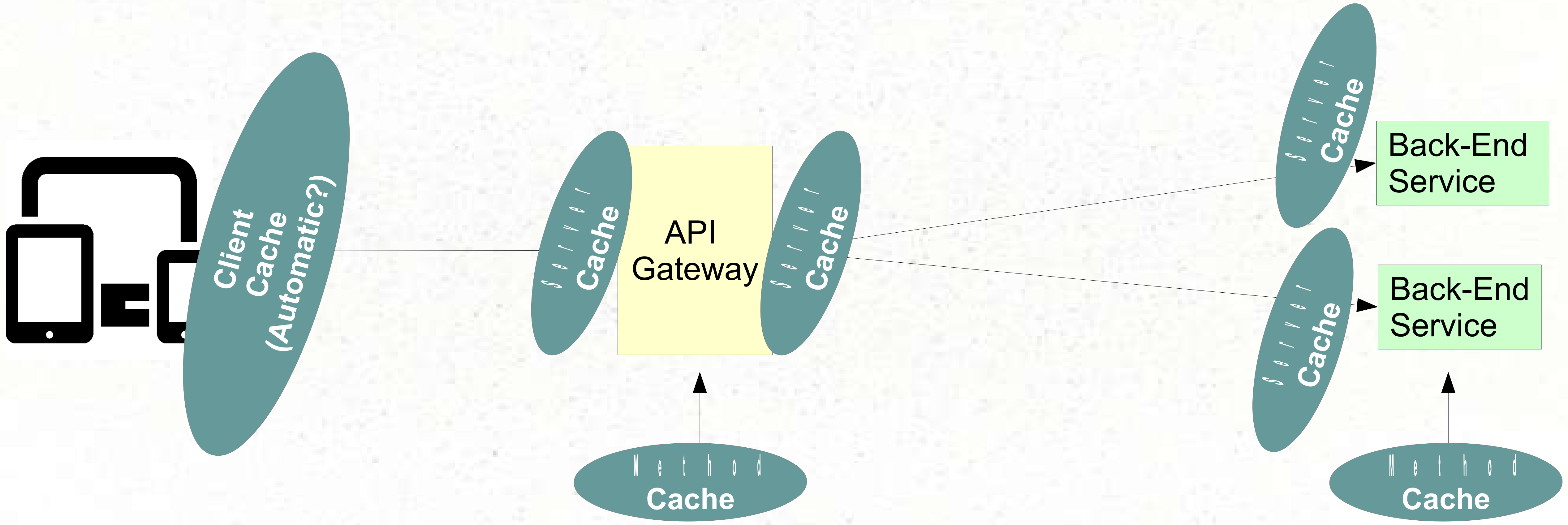
- What parts are missing?
  - Caching
  - Protocol translation
  - Resource Expansion / Link Resolution

Module Outline

- The Need for API Gateway
- Spring Cloud Netflix Zuul
- **Caching**
- Resource Expansion
- Protocol Translation

Caching Possibilities

- Where can we use caching in our application?



Spring's Caching Abstraction

- Spring Framework provides a Caching Abstraction
  - Annotate methods using @Cacheable
  - Describe cache and key
  - Define a CacheManager
    - Backed by SynchronizedMaps, EHCache, Gemfire, etc.

```
@FeignClient(url="localhost:8080/warehouse")
public interface InventoryClient {

   @Cacheable( value="inventory",  key="#sku" )
   @RequestMapping("/inventory/{sku}" )
   public @ResponseBody Item getnventoryItem(@PathVariable Long sku);
}
```

@EnableCacheable

@Cacheable Shortcoming

- @Cacheable works great! But...
  - Cache policy should ideally be directed by the "warehouse" service.
  - Warehouse server should use *expires* and *etag* headers

```
@FeignClient(url="localhost:8080/warehouse")
public interface InventoryClient {

   @Cacheable( value="inventory",  key="#sku" )
   @RequestMapping("/inventory/{sku}" )
   public @ResponseBody Item getnventoryItem(@PathVariable Long sku);
}
```

@EnableCacheable

ETags

- Modern, HTTP-based Caching, better than *expires*.
  - Client requests resource
  - Server returns resource with Etag
    - Hash value calculated from content
  - Client sends if-none-match header with Etag value whenever requesting the same resource
  - Server calculates new hash.
    - If it matches, return 304
    - If not, return 200, new content, new Etag.

ETags – Server Side

- Use the “shallow” ETag Servlet Filter

```
//  Within Spring Boot Application:
@Bean
public Filter shallowEtagHeaderFilter() {
    return new ShallowEtagHeaderFilter();
}
```

- How it works:
  - Calculates Hash value on Response Body.
  - Returns ETag Header with Hash value.
  - Stores Hash with original URL
  - Examines subsequent requests for same resource
    - If the if-none-match hash value matches, return 304.

ETag Client Side - RestTemplate

- RestTemplate does not have Caching built in
  - But HttpClient does!

```
CacheConfig cacheConfig = CacheConfig.custom()
        .setMaxCacheEntries(1000)
        .setMaxObjectSize(8192)
        .build();

CloseableHttpClient cachingClient = CachingHttpClients.custom()
        .setCacheConfig(cacheConfig)
        .build();

RestTemplate template =
      new RestTemplate(
          new HttpComponentsClientHttpRequestFactory(cachingClient));
```

Dependencies: org.apache.httpcomponents, httpclient and httpclient-cache 4.5, http-core 4.4.1

ETag Client Side - Feign

- Feign does not have Caching capability
  - :-(
- Consider creating your own Aspect for use with Feign

Module Outline

- The Need for API Gateway
- Spring Cloud Netflix Zuul
- Caching
- **Resource Expansion**
- Protocol Translation

Resource (Link) Expansion

```
{
  "_embedded" : {
    "team" : [ {
      "name" : "Harlem",
      "location" : "Globetrotters",
      "_links" : {
        "self" : { "href" : "http://localhost:8080/teams/1" },
        "players" : { "href" : "http://localhost:8080/teams/1/players" }
      }
    }, { …  } ]
  }
}
```

```
{
  "_embedded" : {
    "player" : [ {
      "name" : "Big Easy",
      "position" : "Showman",
      "_links" : { "self" : { "href" : "http://localhost:8080/players/3" }
      }
    }, {
      "name" : "Dizzy",
      "position" : "Guard",
      "_links" : { "self" : {  "href" : "http://localhost:8080/players/1" }
      }
    }, { …  }  ]
  }
}
```

Resource Expansion Options: Traverson

- Traverson – Part of Spring HATEOAS Project
  - Original library made for Node JS
  - “Traverses” links.
  - Dependencies: spring-hateoas and json-path

```
<dependency>
   <groupId>org.springframework.hateoas</groupId>
   <artifactId>spring-hateoas</artifactId>
</dependency>

<dependency>
   <groupId>com.jayway.jsonpath</groupId>
   <artifactId>json-path</artifactId>
</dependency>
```

Traverson Basic Usage

- Step 1: Create “Resources” for your domain objects
  - Resources comprehend HAL structure / links:

```
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.Resources;

public class PlayerResources extends Resources<Resource<Player>>{

}
```

Traverson Basic Usage, continued

- Step 2: Traverse

Create Traverson object
Define URL, Content Type

```
Traverson traverson = new Traverson(
    new URI("http://localhost:8080/"),
    MediaTypes.HAL_JSON);

PlayerResources playerResources = traverson
  .follow("$_links.team.href", "$_embedded.team[0]._links.players.href")
  .toObject(PlayerResources.class);

for ( Resource<Player> playerResource : playerResources.getContent() ) {
   Player player = playerResource.getContent();
   System.out.println(player);
}
```

Describe link structure to traverse
Describe return type

```
Player: Buckets, Guard
Player: Big Easy, Showman
Player: Dizzy, Guard
```

Traverson Drawbacks

- Traversal, not Expansion
- Limited capability with other formats
- No support for XML

Resource Expansion Option:
Spring Data REST Projections

- Part of Spring Data REST
- Causes links to be "inlined"
  - Not links

```
{
  "_embedded" : {
    "teams" : [ {
      "name" : "Harlem",
      "location" : "Globetrotters",
      "players" : [ { "name" : "Buckets",
                      "position" : "Guard" },
        {             "name" : "Dizzy",
                      "position" : "Guard" },
        {             "name" : "Big Easy",
                      "position" : "Showman"  } ],
      "_links" : { … },
    }, { … } ]
} }
```

Spring Data REST Projections

- Step 1: Define the Projection as an Interface

```
@Projection(name = "inlinePlayers", types = { Team.class })
public interface InlinePlayers {
      String getName();
      String getLocation();
      String getMascotte();
      Set<Player> getPlayers();
}
```

- Step 2: Supply projection name on GET:
  - http://localhost:8080/teams/1**?projection=inlinePlayers**

Spring Data REST Projections

/teams/1

```
{
  "name" : "Harlem",
  "location" : "Globetrotters",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/teams/1{?projection}",
      "templated" : true
    },
    "players" : {
      "href" : "http://localhost:8080/teams/1/players"
    }
  }
}
```

/teams/1**?projection=inlinePlayers**

```
{
  "name" : "Harlem",
  "location" : "Globetrotters",
  "players" : [ {  "name" : "Dizzy",
                   "position" : "Guard"         }, {
                   "name" : "Big Easy",
                   "position" : "Showman"       }, {
                   "name" : "Buckets",
                   "position" : "Guard"         } ],
  "_links" : {
    "self" : { "href" : "http://localhost:8080/teams/1{?projection}",
      "templated" : true
    },
    "players" : { "href" : "http://localhost:8080/teams/1/players" }
  }
}
```

Spring Data REST Projections: Drawbacks

- Drawbacks:
  - Only works when using Spring Data REST
  - Only works when projections are part of the same microservice.

- For more information:
  - http://docs.spring.io/spring-data/rest/docs/current/reference/html/#projections-excerpts

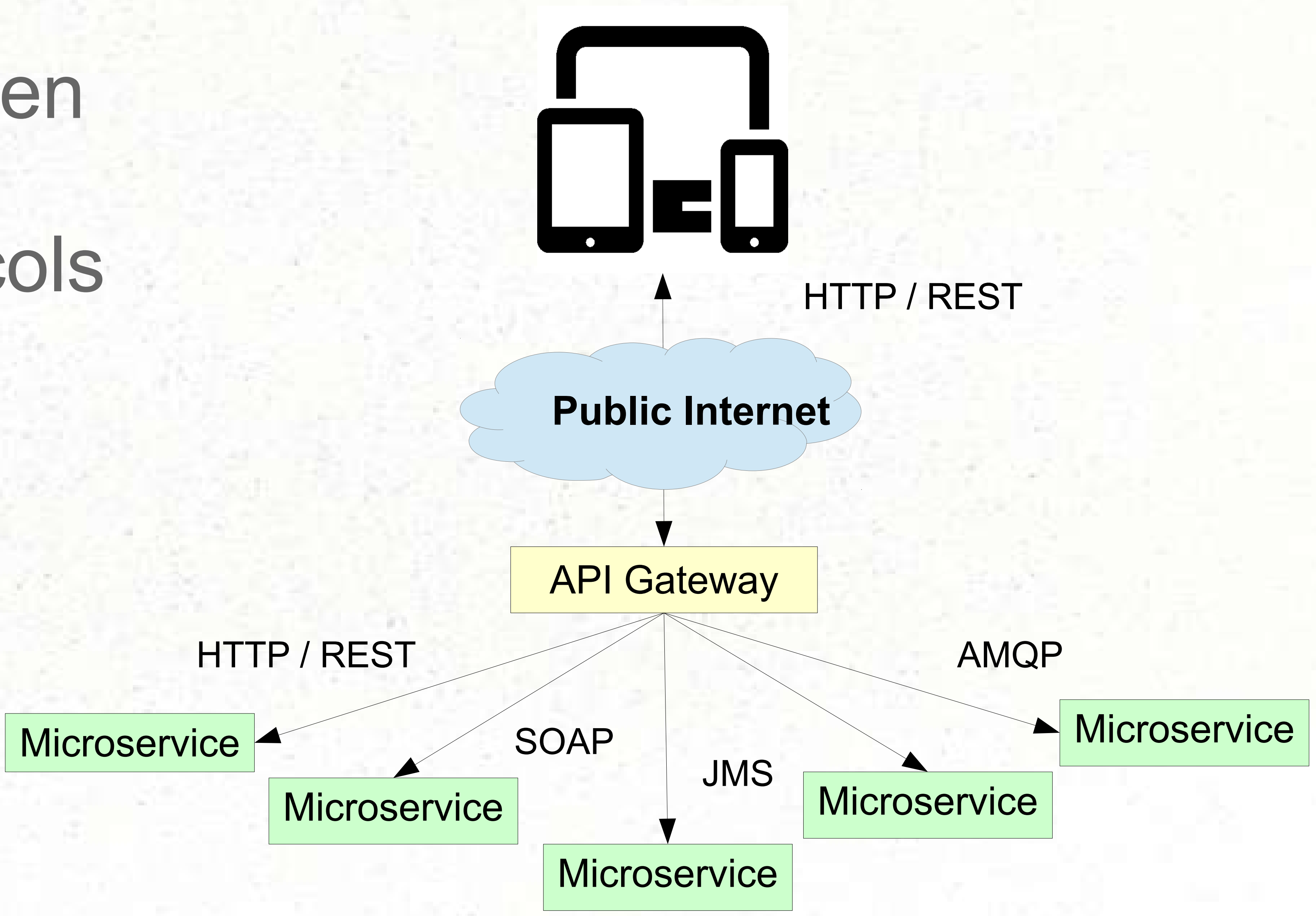
Module Outline

- The Need for API Gateway
- Spring Cloud Netflix Zuul
- Caching
- Resource Expansion
- **Protocol Translation**

Protocol Translation

The Issue...

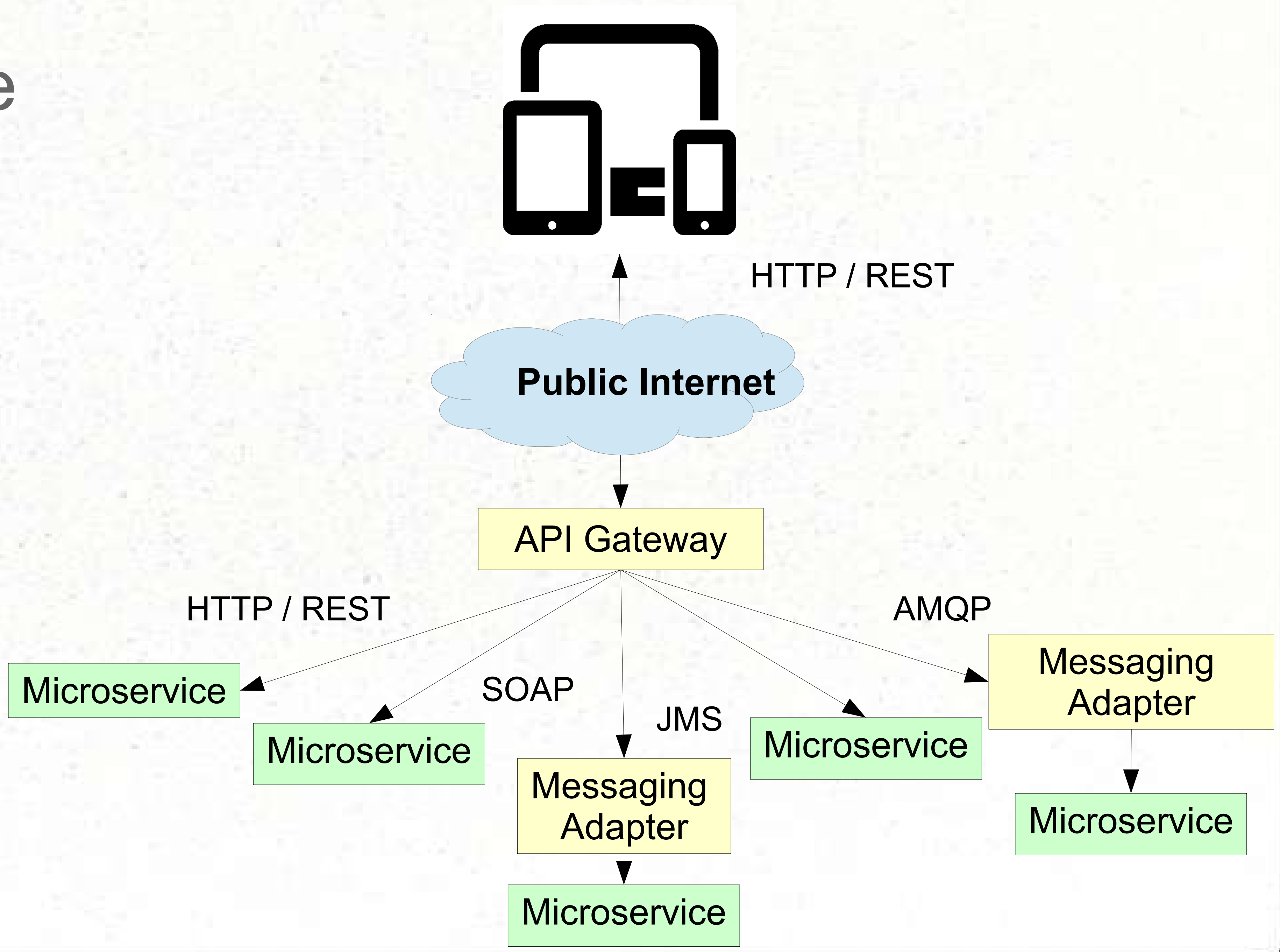
- Translate between front-end and back-end protocols



Protocol Translation

The Issue...

- Adapters can be used...



Protocol Translation
The Options

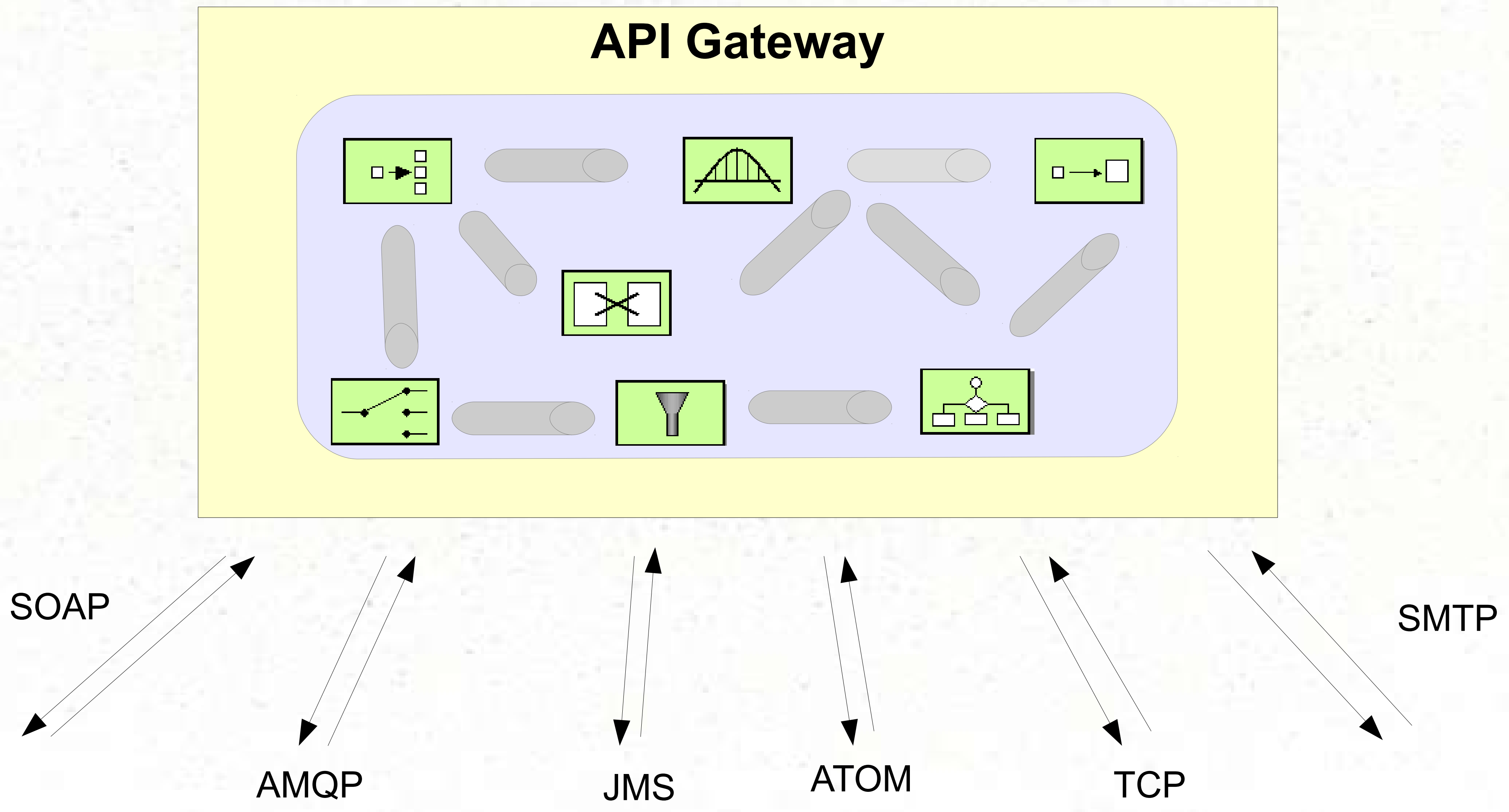
- No Quick Fixes

- JMS – Use JmsTemplate
  - http://docs.spring.io/spring/docs/current/spring-framework-reference/html/jms.html
- AMQP – Use AmqpTemplate
  - http://docs.spring.io/spring-amqp/reference/html/amqp.html
- SOAP – Use WebServiceTemplate
  - http://docs.spring.io/spring-ws/docs/current/reference/htmlsingle/#client-web-service-template

Protocol Translation: Spring Integration

- Spring Integration
  - Powerful framework for enterprise integration patterns and in-memory messaging.
  - http://projects.spring.io/spring-integration/
- Adapters and Gateways for:
  - AMQP, ATOM, Flat Files, FTP(S), Gemfire, HTTP, JDBC, JMS, JMX, JPA, eMail, Mongo, MQTT, Redis, RMI, SFTP, streams, syslog, TCP, Twitter, UDP, Web Services (SOAP), Web Sockets, XMPP

Spring Integration



Summary

- API Gateway - “Adapter” built for client needs
- Zuul – Easy Routing and Load Balancing
- Caching
- Resource Expansion
- Protocol Conversion

Exercise – API Gateway

Setup a Simple Zuul Proxy Server

Instructions: Student Files, Lab 9