

BITSIGHT

Microservices FTW

Nelson Gomes / BitSight Technologies

Presentation Topics

Server Setup

- Microservices Introduction
- Setting up Jetty/Jersey
- Database Access
- Caching

REST APIs

- Swagger Introduction
- Swagger File Generation
- Client Generation
- Jackson Behaviour Control

Distribute It

- Circuit Breakers
- Hazelcast Introduction
- Distributed Processing
- Dockerize it

“Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.” - Linus Torvalds

The background of the slide is a grayscale photograph. In the upper half, a server rack is visible with several circular indicator lights. In the lower half, a laptop is open on a desk, with its keyboard and screen visible. The screen shows some graphical data. A blue horizontal band is overlaid across the middle of the image, containing the main title.

Server Setup

BIT SIGHT

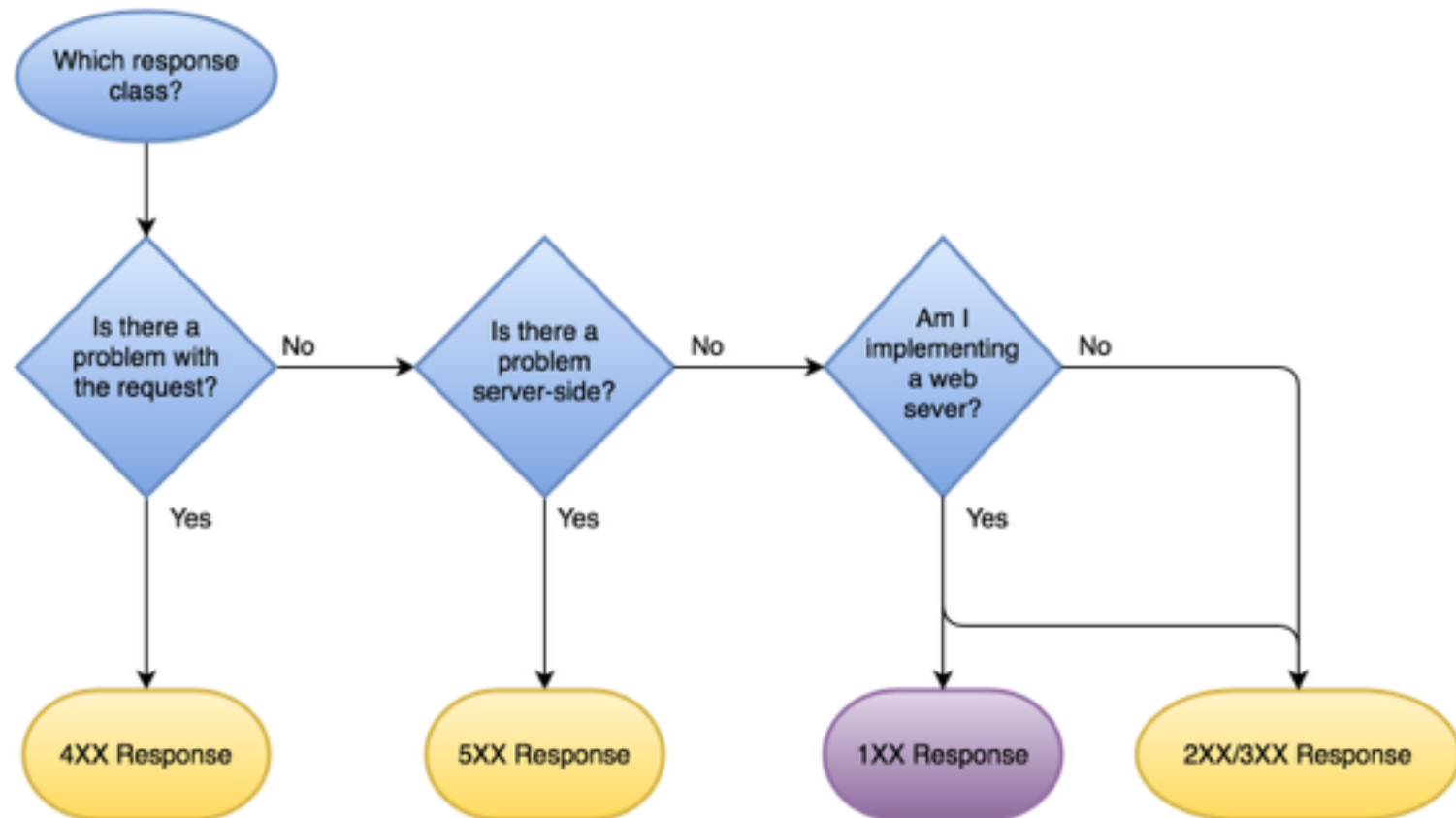
Microservices Introduction

Microservices refers to a software design architecture where applications are made of independently deployable services.

- ▶ Small pieces of software exposed via HTTP/REST
- ▶ Each resource should be a noun (ex: user, company, product)
- ▶ HTTP provides the verbs for CRUD operations:
 - ▶ Create = POST
 - ▶ Read = GET
 - ▶ Update = PUT
 - ▶ Delete = DELETE
- ▶ Stateless, Cacheable, Layered system

Roy Thomas Fielding dissertation for further reading.

HTTP Response Codes applied to REST:



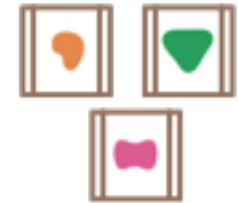
Read about at <http://racksburg.com/choosing-an-http-status-code/>

**Microservices should not be confused with REST APIs.
A Microservice should be:**

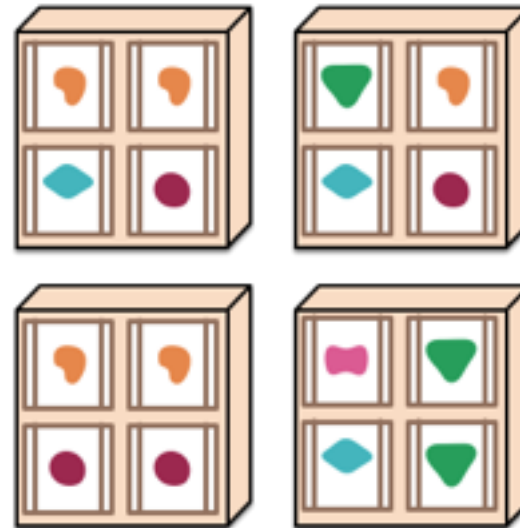
- ▶ **Scalable** - adding more servers to handle specific REST calls
- ▶ **Layered** - abstracting what server handles the request
- ▶ **Redundant** - a server down will have theoretically no impact
- ▶ **Stateless** - no server side state, each request must provide all information
- ▶ **Unique functionality** - only one service implements it

Microservices architecture:

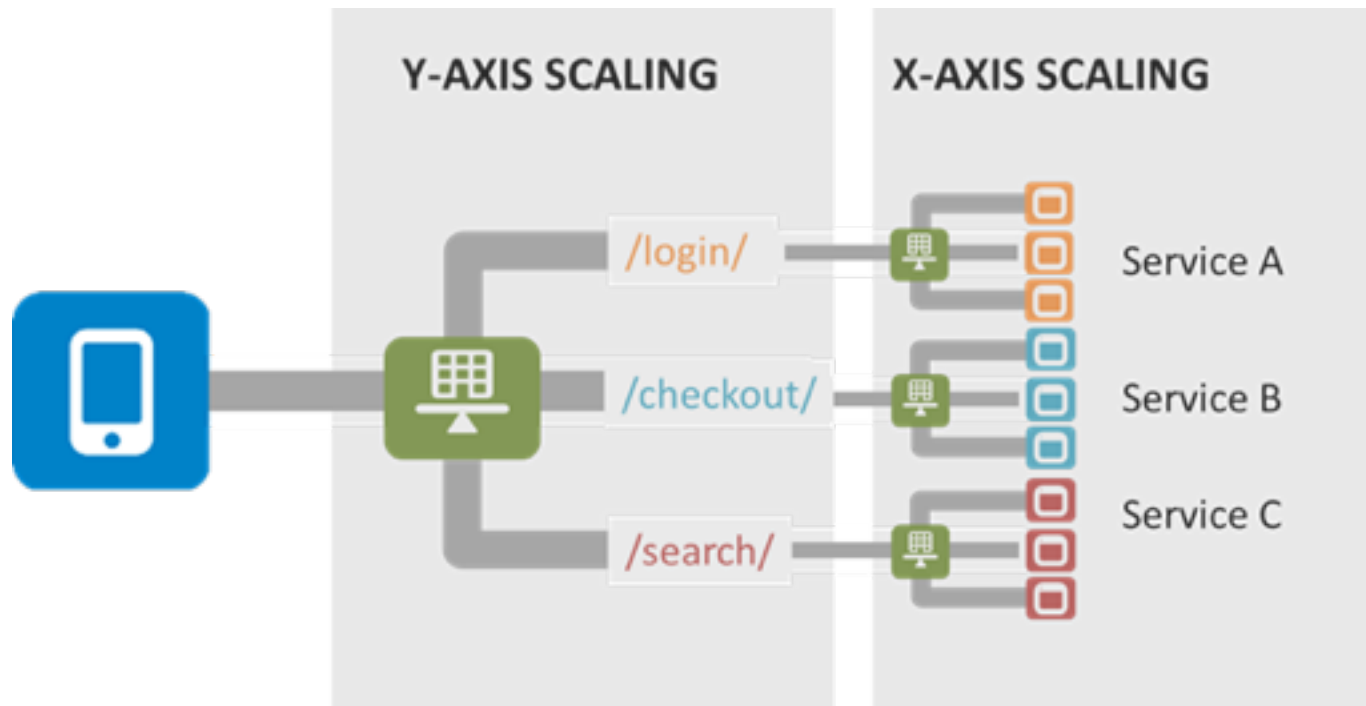
A microservices architecture puts each element of functionality into a separate service...



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



Microservices architecture:



Usually Y-AXIS scaling is done using hardware load balancers or software load balancers like Nginx or HAProxy. X-AXIS can be hosts, VMs or containers.

Microservices disadvantages:

- ▶ Distributed architectures are hard to implement
- ▶ Operational complexity
 - ▶ service monitoring
 - ▶ control cascading effect
 - ▶ problem detection
- ▶ Complexity of cross-service refactoring
- ▶ Service dependency and versioning

The very first thing you should have operational is a metrics system to allow global monitoring of the system, otherwise you won't have any awareness of it's health.

The background of the slide is a grayscale photograph. In the upper half, a server rack is visible with several glowing indicator lights. In the lower half, a laptop is open on a desk, with its keyboard and screen clearly visible. The overall scene suggests a technical or server environment.

Server Setup

BIT SIGHT

Setting up Jetty/Jersey

Setup steps:

- ▶ Create a java project with war packaging
- ▶ Add maven dependencies (see the github link for this)
- ▶ There are at least two ways to start a jetty server:
 - ▶ Start a main class (useful for development), or
 - ▶ Use jetty-runner to deploy the project after packaging

```
▶ $ mvn package
▶ $ java -jar jetty-runner.jar application.war
```

Jetty is a Java standalone web server and servlet container.

Jersey is a framework for RESTful web services.

```
public static void main(String[] args) throws Exception {
    ServerMain.log.info("Starting jetty version {}", Server.getVersion());

    // configure jetty server
    Server jettyServer = new Server(8080);

    // setup up access log
    final NCSARequestLog requestLog = new NCSARequestLog("logs/access-yyyy_mm_dd.log");
    requestLog.setAppend(true);
    requestLog.setExtended(true);
    requestLog.setLogTimeZone("GMT");
    jettyServer.setRequestLog(requestLog);

    // setup web app
    final WebApplicationContext webapp = new WebApplicationContext();
    webapp.setResourceBase("src/main/webapp/");

    // pass webapp to jetty
    jettyServer.setHandler(webapp);

    // start web server
    jettyServer.start();
    jettyServer.join();
}
```

At this point we should have a
HTTP server ready to run on
port 8080:



Now let's configure our web app:

- ▶ Let's create a WEB-INF/web.xml file in our webapp folder

```
<web-app>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

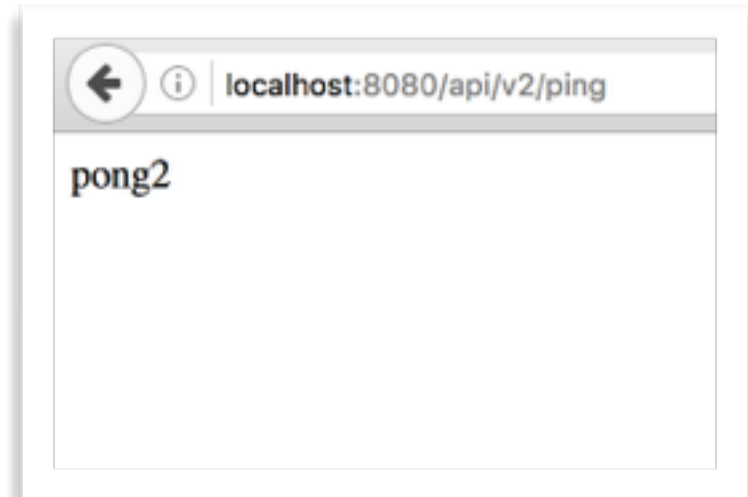
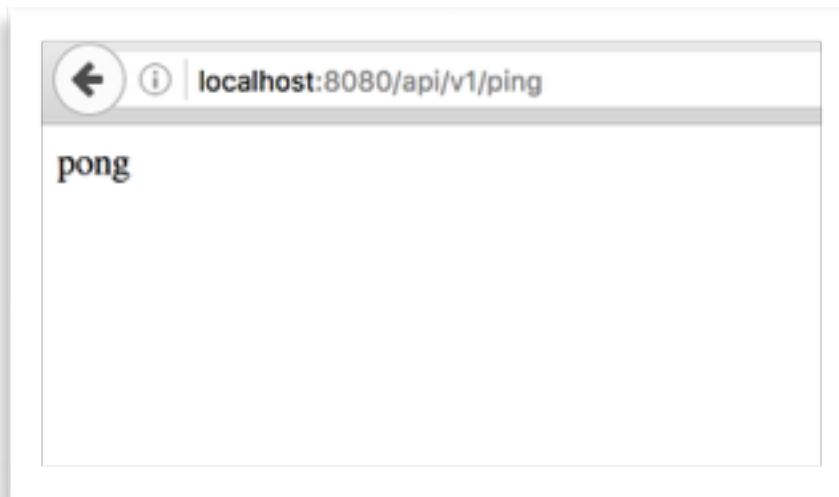
  <servlet>
    <servlet-name>RestApi</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>
        com.msftw.api.v1,
        com.msftw.api.v2
      </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>RestApi</servlet-name>
    <url-pattern>/api/*</url-pattern>
  </servlet-mapping>
</web-app>
```

```
@Path("/v1")
public class ApiV1 {

    @Path("/ping")
    @GET
    public String ping() {
        return "pong";
    }
}
```

At this point we should have a HTTP server responding on/ api/v1/ping and api/v2/ping





Server Setup

BIT SIGHT

Database Access

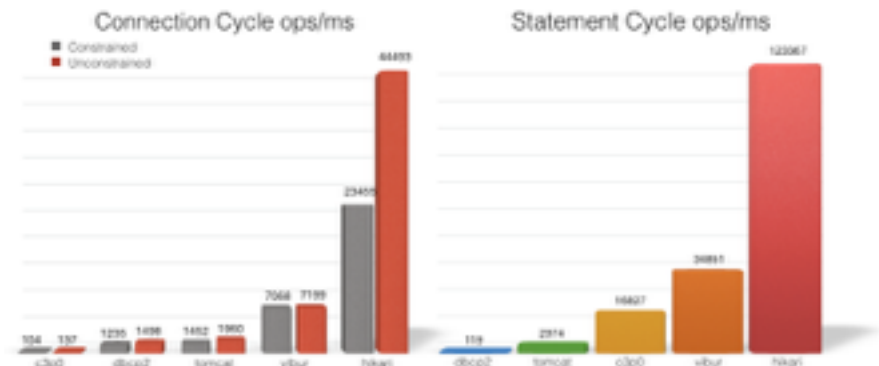
There are a lot of database access layers. You should choose the one that brings you the most benefits:

These frameworks usually depend on some other framework for connection pooling like Apache DBCP or HikariCP

My favourite combination is DbUtils with HikariCP

- ▶ Raw JDBC - avoid it at all cost, because resource management and connection pool are hard to manage
- ▶ Apache DbUtils - very reliable and very fast, my favourite
- ▶ JDBI - reliable, simple to use, but not so fast
- ▶ jOOQ - an ORM engine, gains on relations, loses some performance
- ▶ Hibernate - another ORM engine
- ▶ Sql2o - faster than DbUtils, to monitor in the future

DBUtils is a very fast, lightweight and reliable framework, HikariCP is a high performance JDBC connection pool



```
public class DatabaseHelper implements AutoCloseable {
    private static Map<String, HikariDataSource>
        pools = new ConcurrentHashMap<String, HikariDataSource>();

    public static DataSource getDataSource(String config) {
        if (!pools.containsKey(config)) createDataSource(config);
        return pools.get(config);
    }

    private static synchronized void createDataSource(String config) {
        if (!pools.containsKey(config))
            pools.put(config, new HikariDataSource(new HikariConfig(config)));
    }

    public void close() throws Exception {
        pools.clear();
    }
}

QueryRunner runner = new QueryRunner(DatabaseHelper.getDataSource("config"));
```



Server Setup

BIT SIGHT

Caching



Caching Paradigms:

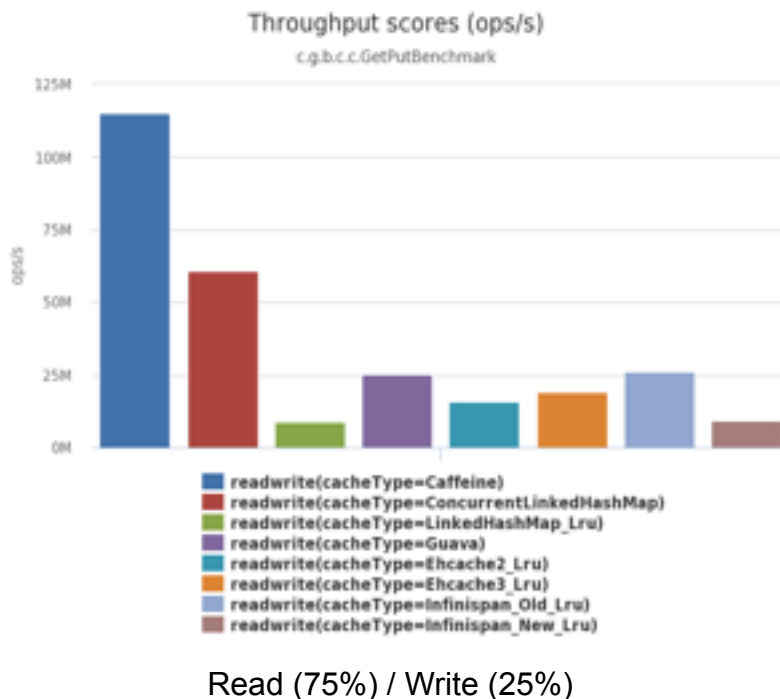
- ▶ Reactive - cache is generated on first miss, used when the cache domain is too big. Generates race conditions on heavy load. Usually has a TTL and is dropped by LRU algorithm.
- ▶ Proactive - cache domain is limited, can fit in memory and is generated beforehand. Usually this cache won't have a single miss, and is refreshed periodically or when an event occurs.

You should cache as much as you can if you want performance. If it saves you a query, a system call or some communication, cache it even for a short TTL.

Caching Levels:

- ▶ Static - cache content for URIs that don't change often
- ▶ Response - cache responses using call and parameters as key
- ▶ Data - cache data processed by some key
- ▶ Remote - cache remote call responses using call and parameter as key
- ▶ Query - cache query responses using the query and parameters as key

Caching Frameworks:



- ▶ Caffeine - very fast, Java 8, Guava drop-in replacement
- ▶ ConcurrentLinkedHashMap
- ▶ Google Guava
- ▶ Ehcache - distributed cache
- ▶ Infinispan - distributed cache
- ▶ HazelCast - distributed cache

A good option is to combine local cache with a distributed one, improving in speed and reliability.

Caching engine should be chosen by application requirements: Speed vs Reliability

Sample Caching:

```
▶ LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()  
    .maximumSize(10_000)  
    .expireAfterWrite(5, TimeUnit.MINUTES)  
    .refreshAfterWrite(1, TimeUnit.MINUTES)  
    .build(key -> createExpensiveGraph(key));
```

```
▶ Cache<Key, Graph> graphs = Caffeine.newBuilder()  
    .maximumSize(1000)  
    .expireAfterWrite(3600, TimeUnit.SECONDS)  
    .build();
```

```
Graph graph = graphs.getIfPresent(key);  
if(graph == null){  
    // generate graph and cache it  
    graph = createExpensiveGraph(key);  
    graphs.put(key, graph);  
}
```

A blurred background image showing a computer monitor and a laptop on a desk. The monitor is in the upper left, and the laptop is in the lower right. The image is in grayscale with a blue overlay on the right side.

REST APIs

BIT SIGHT

Swagger Introduction

Swagger is a simple yet powerful representation of your RESTful API.

In today's presentation we'll demonstrate three tools:

- **Swagger file generation in Java**
- **Swagger UI**
- **Swagger Editor**

- ▶ Swagger Specification has been donated to the Open API Initiative
- ▶ It's adoption has several benefits:
 - ▶ Interactive documentation
 - ▶ Client SDK generation
 - ▶ Server generation
 - ▶ API discoverability
 - ▶ Lots of development goodies

For our project to generate Swagger information we'll add annotations to our methods.

- ▶ Annotating Java code is a simple way of maintaining your API
- ▶ The maven plugin will do the rest by reflection and produce documentation and the Swagger file, advantages:
 - ▶ Easy update and addition of methods
 - ▶ No need to diff generated code with your codebase
 - ▶ Technology independent
 - ▶ Easier to maintain versions

Some common annotations:

- ▶ `@Api` - to declare an API resource (class level)
- ▶ `@ApiModel` - provides additional information

```
@Api
@ApiModel(value = "APIv1", description = "Microservices FTW Methods")
@Path("/v1")
public class ApiV1 {
```

Some common annotations:

- ▶ `@ApiOperation`
- ▶ `@ApiResponses`

```
@GET
@Path("/ping")
@Produces(MediaType.TEXT_PLAIN)
@ApiOperation(value = "Ping method", notes = "Method to check API connectivity.")
@ApiResponses(value = {
    @ApiResponse(code = 200, message = "successful operation",
        response = String.class) })
public String pingMethod(@Context final HttpServletRequest request) {
    log.info("Received request for {} from IP {}", request.getRequestURI(),
        request.getRemoteAddr());
    return "pong";
}
```

Response structures:

- ▶ responseContainerTypes: List, Set and Map

```
@Path("/books")
@Produces(MediaType.APPLICATION_JSON)
@ApiOperation(value = "List Books method",
              notes = "Method to get a books list.")
@ApiResponses(value = {
    @ApiResponse(code = 200, message = "successful operation",
                 response = Book.class, responseContainer = "List"),
    @ApiResponse(code = 500, message = "unsuccessful operation",
                 response = String.class) })
public Response getBooks() {
```

Some common annotations:

- ▶ @ApiParam
- ▶ @ResponseHeaders

```
@POST
@Path("/books")
@Produces(MediaType.APPLICATION_JSON)
@ApiOperation(value = "Create Books method", notes = "Method to create books.")
@ApiResponses(value = {
    @ApiResponse(code = 200, message = "successful operation",
        response = Boolean.class, responseContainer = "List"),
    @ApiResponse(code = 500, message = "unsuccessful operation", response = String.class,
        responseHeaders = {@ResponseHeader(name="X-Host",
            description="hostname that failed")}) })
public Response createBooks(
    @ApiParam(value="Json array of Books", required = true) final LinkedList<Book> books,
    @ApiParam(value="sample comment") @QueryParam("comment") String comment) {
```

Some common annotations:

► @ApiModelProperty

```
public class Book {
    @ApiModelProperty(value = "Book id", required = false, example = "1")
    private Integer id;
    @ApiModelProperty(value = "Book title", required = true, example = "Book Title")
    private String title;
    @ApiModelProperty(value = "Book ISBN", required = true, example = "Book ISNTitle")
    private String isbn;
    @ApiModelProperty(value = "Book Price", required = true, example = "12.12")
    private Float price;
    @ApiModelProperty(value = "Publisher Name", required = true, example = "Publisher")
    private String publisher;

    public Integer getId() {
        return id;
    }

    (...)
}
```

A blurred background image showing a computer monitor and a laptop on a desk. The monitor is in the upper left, and the laptop is in the lower right. The text is overlaid on a dark horizontal band across the middle.

REST APIs

BIT SIGHT

Swagger File Generation

We're going to use Swagger Maven plugin to generate our Swagger file

- ▶ Download latest version and copy folder test/resources/templates to src/main/resources
- ▶ Now we just need to hook it to our Maven project
- ▶ Now your Swagger API and file will be updated instantly every time you package your project

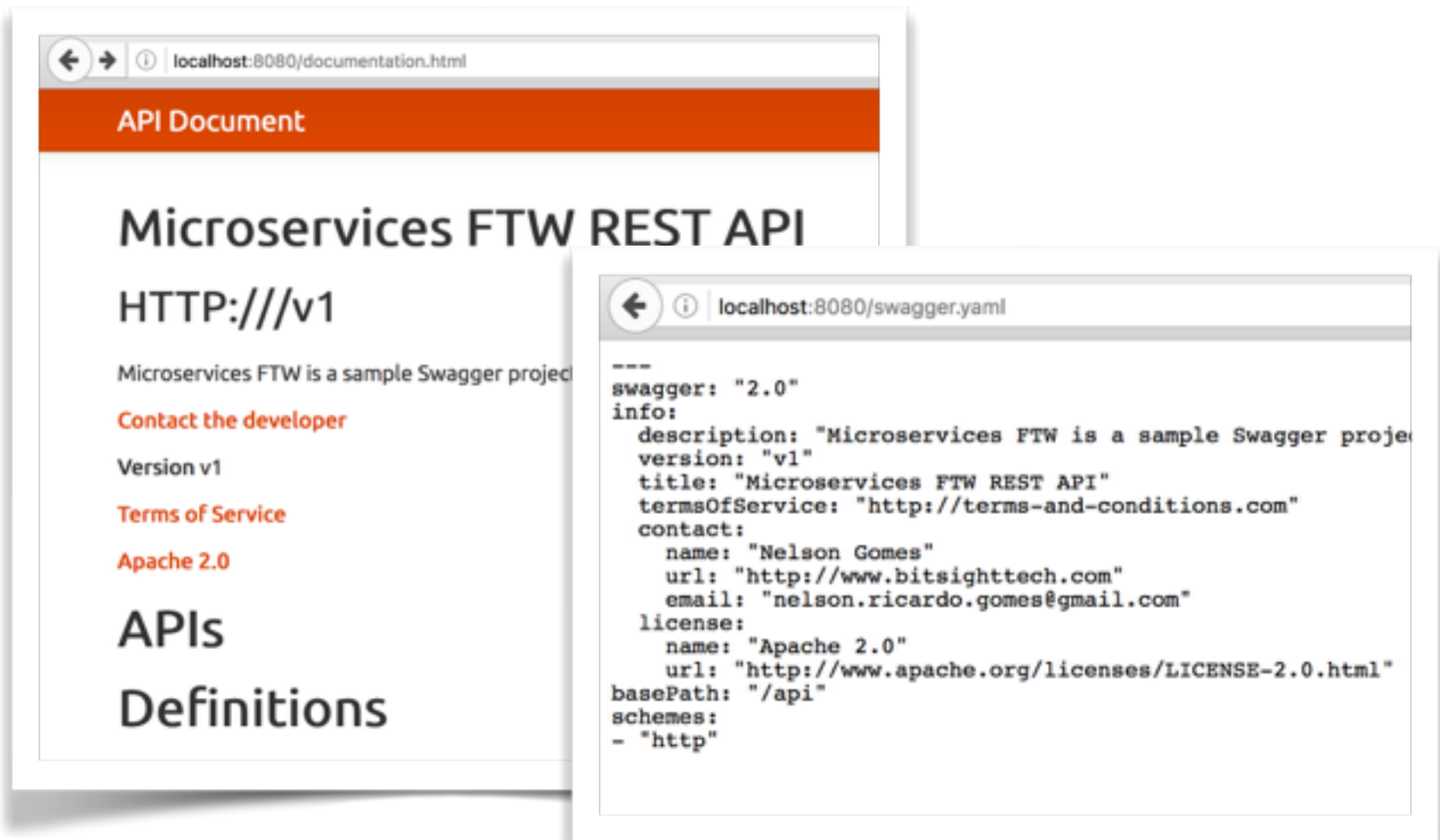
```
▶ $ mvn package
```

Download source from <http://kongchen.github.io/swagger-maven-plugin/>

changes to pom.xml

```
<plugin>
  <groupId>com.github.kongchen</groupId>
  <artifactId>swagger-maven-plugin</artifactId>
  <version>3.1.3</version>
  <configuration>
    <apiSources>
      <apiSource>
        <springmvc>>false</springmvc>
        <locations><!-- declare packages -->
          com.msftw.api.v1
        </locations>
        <schemes>http</schemes>
        <basePath>/api</basePath>
        <info> (...) </info>
        <templatePath>(…)/strapdown.html.hbs</templatePath>
        <outputPath>(…)/webapp/documentation.html</outputPath>
        <swaggerDirectory>(…)/webapp</swaggerDirectory>
        <outputFormats>yaml,json</outputFormats>
        <attachSwaggerArtifact>true</attachSwaggerArtifact>
      </apiSource>
    </apiSources>
  </configuration>
  (...)
```

At this point we should have documentation generated and an empty Swagger file generated:



The image displays two browser windows side-by-side. The left window shows the Swagger API documentation for 'Microservices FTW REST API' at 'localhost:8080/documentation.html'. It features an orange header, a title, a version 'v1', and links for 'Contact the developer', 'Terms of Service', and 'Apache 2.0'. The right window shows the 'swagger.yaml' file at 'localhost:8080/swagger.yaml', which contains the Swagger 2.0 configuration in YAML format.

API Document

Microservices FTW REST API

HTTP:///v1

Microservices FTW is a sample Swagger project

[Contact the developer](#)

Version v1

[Terms of Service](#)

[Apache 2.0](#)

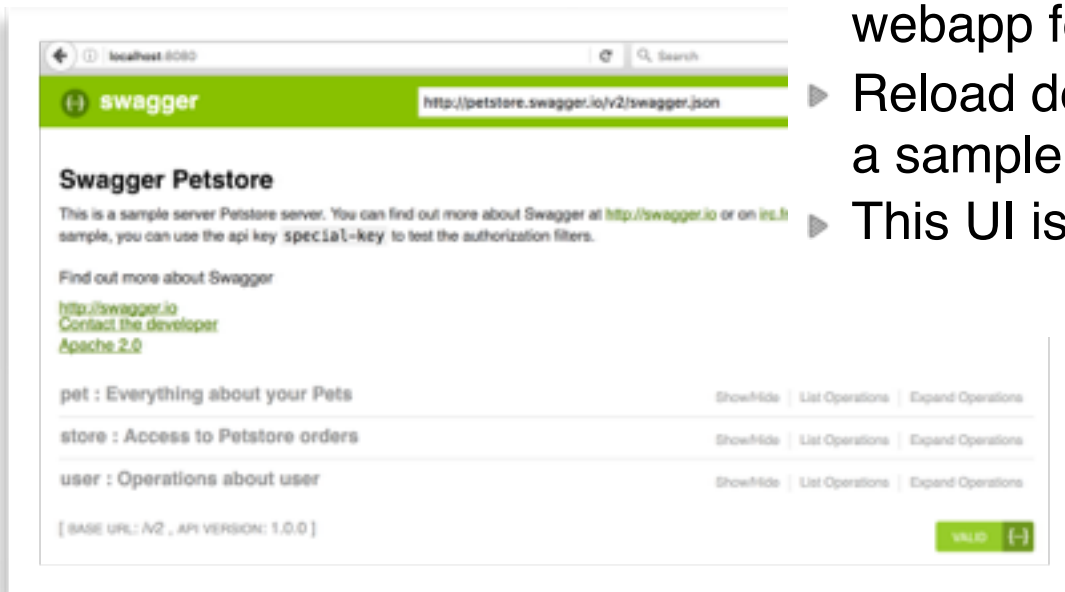
APIs

Definitions

```
---
swagger: "2.0"
info:
  description: "Microservices FTW is a sample Swagger project"
  version: "v1"
  title: "Microservices FTW REST API"
  termsOfService: "http://terms-and-conditions.com"
  contact:
    name: "Nelson Gomes"
    url: "http://www.bitsighttech.com"
    email: "nelson.ricardo.gomes@gmail.com"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
basePath: "/api"
schemes:
- "http"
```

Now let's setup Swagger-UI:

- ▶ Go to <http://swagger.io/swagger-ui/> for documentation
- ▶ Download latest Swagger UI from <https://github.com/swagger-api/swagger-ui/releases>
- ▶ Decompress and copy dist folder contents to our project src/main/webapp folder
- ▶ Reload default webpage to see a sample swagger interface
- ▶ This UI is a very useful tool



A blurred background image showing a computer monitor and a laptop on a desk. The monitor is in the upper left, and the laptop is in the lower right. The image is in grayscale with a blue overlay on the right side.

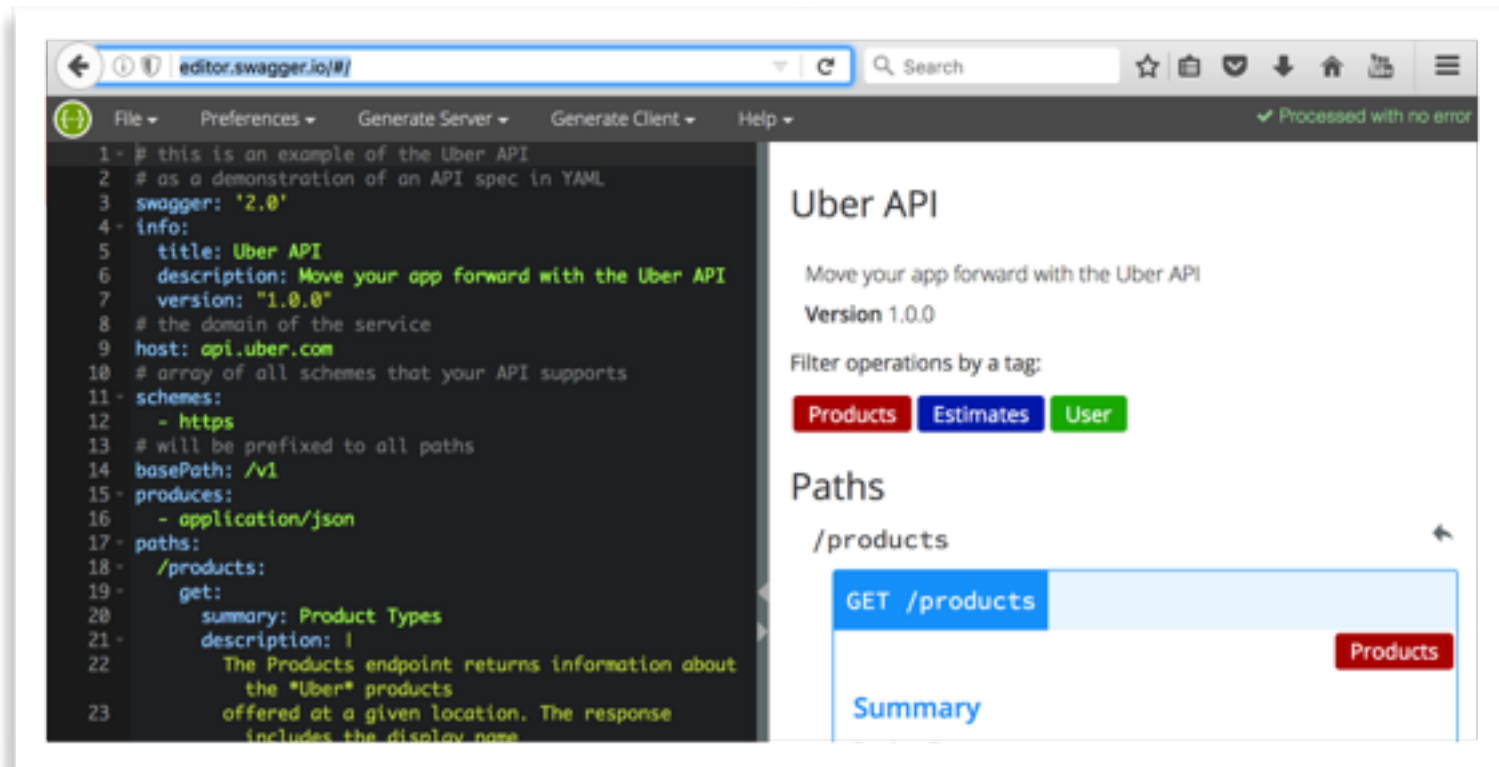
REST APIs

BIT SIGHT

Client Generation

The biggest benefit in using Swagger is the easiness to generate client stubs for almost every language:

- ▶ Go to <http://editor.swagger.io/> to generate your client/server
- ▶ Automate code generation into your development cycle







REST APIs

BIT SIGHT

Jackson Behaviour Control

Jackson handles json parsing but does not handle some special cases:

- ▶ An empty POST gives a 500 error before calling our method
- ▶ This change allows receiving null objects in our methods and handle those errors

```
@Provider
public class CustomJsonProvider extends JacksonJaxbJsonProvider {
    private static ObjectMapper mapper = new ObjectMapper();
    static {
        CustomJsonProvider.mapper.configure(
            DeserializationFeature.ACCEPT_EMPTY_STRING_AS_NULL_OBJECT, true);
    }

    public CustomJsonProvider() {
        super();
        this.setMapper(CustomJsonProvider.mapper);
    }
}
```



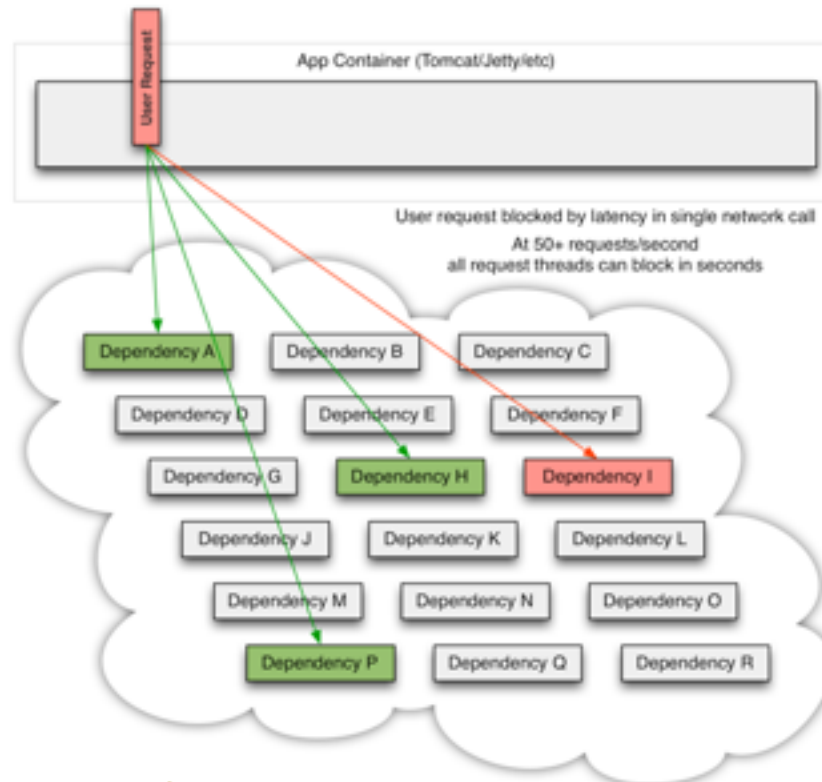
Distribute It

BIT SIGHT

Circuit Breakers

Circuit Breakers are a very important feature to avoid domino effect when failures occur in chained Microservices

- Hystrix is a good framework to help implement circuit breakers



<https://github.com/Netflix/Hystrix/wiki/How-To-Use#Common-Patterns>

Hystrix helps alleviate the domino effect that can cause crashes in distributed systems

- ▶ Preventing any single dependency from using up all user threads.
- ▶ Shedding load and failing fast instead of queueing.
- ▶ Providing fallbacks wherever feasible to protect users from failure.
- ▶ Using isolation techniques to limit the impact of any one dependency.
- ▶ Optimizing for time-to-discovery through near real-time metrics, monitoring, and alerting
- ▶ Optimizing for time-to-recovery
- ▶ Protecting against failures in the entire dependency client execution, not just in the network traffic.

A blurred background image showing a computer monitor and a laptop on a desk. The monitor is in the upper left, and the laptop is in the lower right. The image is in grayscale with a blue overlay on the right side.

Distribute It

BIT SIGHT

Hazelcast Introduction

Hazelcast is a zero configuration cluster framework

- ▶ With Hazelcast you can create a cluster with 1 line of code

```
private static HazelcastInstance
    clusterNode = Hazelcast.newHazelcastInstance(new Config());
private static ConcurrentMap<String, String> map;

static {
    // initialize our distributed map
    map = clusterNode.getMap("my-distributed-map");
}

// Concurrent Map methods
map.putIfAbsent("somekey", "somevalue");
map.replace("key", "value", "newvalue");
```

<https://hazelcast.org/>
<https://hazelcast.org/features/>

Microservices are not the answer to everything:

- ▶ If you can process a request later, put it in a queue
- ▶ Parallelise your task if possible
- ▶ Avoid long waits on responses
- ▶ Cache everything you can
- ▶ Your system should be resource aware
- ▶ Avoid multi-request transactions
- ▶ Page your results to avoid large requests

Consul can be useful to help managing distributed systems.



Distribute It

BIT SIGHT

Dockerize It

Containers are a very useful tool when you want to scale:

- ▶ A single docker image can be replicated to hundreds of machines
- ▶ Tools like Kubernetes, Docker Swarm, Amazon ECS automate this
- ▶ Containers usually have a small footprint and are very fast to deploy
- ▶ Need to test a specific Linux distribution?

```
# Download an ubuntu image
docker pull ubuntu
# run shell
docker run -it ubuntu /bin/bash
```

A blurred background image showing a computer monitor and a laptop on a desk. The monitor is in the upper left, and the laptop is in the lower right. The text is overlaid on a dark horizontal band across the middle of the image.

Microservices FTW

BIT SIGHT

Demo Time!!!

<https://github.com/nelsongomes/microservices-ftw>

QUESTIONS?

For more information:

nelson.gomes@bitsighttech.com

BitSight Technologies
125 CambridgePark Drive, Suite 204
Cambridge, MA 02140
www.bitsighttech.com