DEVNEXUS™

MICRONAUT

SINGLE PAGE APPS FOR A MICROSERVICES ARCHITECTURE

ABOUT ME

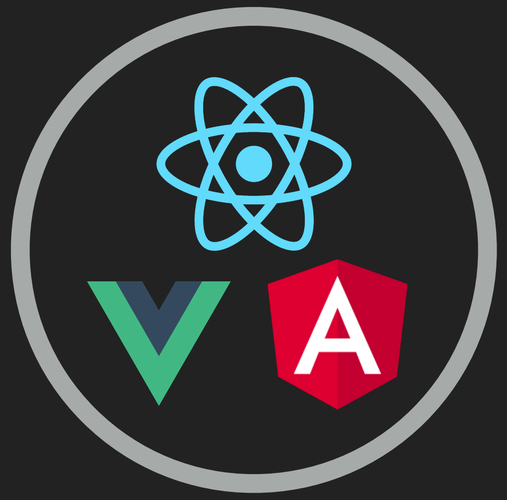
- Zachary Klein
- Principal Software Engineer at Object Computing, Inc
- OSS contributor to Grails and Micronaut
- Twitter: @ZacharyAKlein
- LinkedIn: http://linkedin.com/in/zacharyaklein/
- https://objectcomputing.com

SINGLE PAGE APPS VS MICROSERVICES

OCI

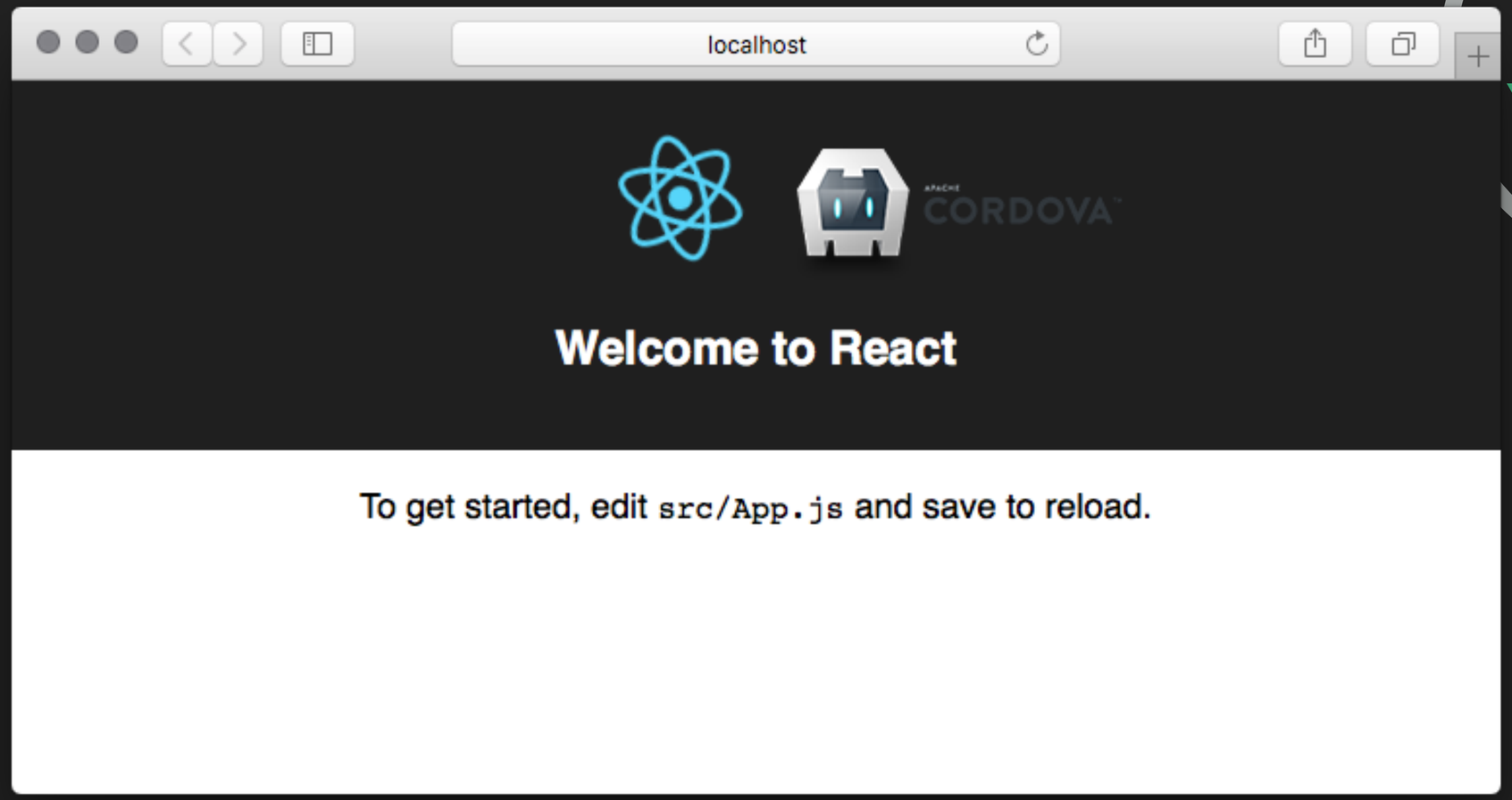
DEVNEXUS™

SINGLE PAGE APPS



- Typically JavaScript Applications
- Run in the browser or via native frameworks (Electron)
- “Single Page” - a single bundle of JavaScript typically provides all the various pages/features of the app
- Typically delegate to a RESTful backend for data access/persistence
- Popular frameworks for writing SPAs include React, Vue, and Angular

SINGLE PAGE APPS



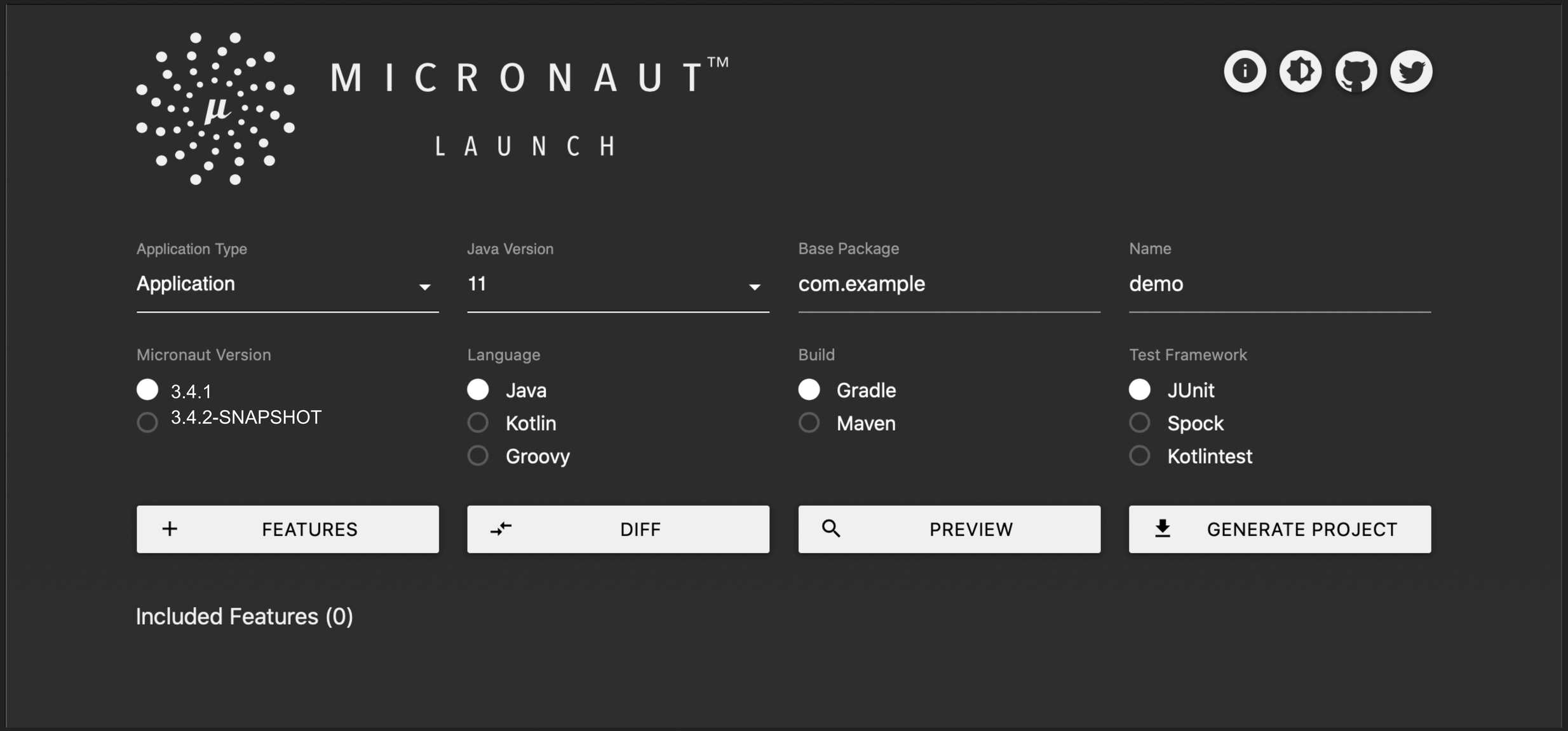
MICROSERVICES

- https://martinfowler.com/articles/microservices.html
- Distributed architecture for service programming
- Services are independently buildable, deployable
- Bounded contexts & separation of concerns
- Popular frameworks (in the Java ecosystem) include Spring Boot, Quarkus, & Micronaut

MICRONAUT – CLOUD NATIVE MICROSERVICES

https://micronaut.io

- A Java Framework designed with Microservices in mind
- Reactive HTTP
- AOT (Ahead of Time) Compilation
- Declarative HTTP Client
- “Natively” Cloud-Native: service-discovery, load-balancing, circuit-breakers, tracing, and more!
- First-Class Support for Java, Kotlin & Groovy



https://launch.micronaut.io

MICRONAUT: CONTROLLERS & CLIENTS

```
@Controller("/")
class HelloController {

  @Get("/hello/{name}")
  String hello(String name) {
    return "Hello " + name;
  }

}
```

```
@Client("/")
interface HelloClient {

  @Get("/hello/{name}")
  String hello(String name);

  // Implementation generated
  // at compile time
}
```

MICRONAUT: CONTROLLERS & CLIENTS

```
@Controller("/")
class HelloController {

  @Get("/hello/{name}")
  String hello(String name) {
    return "Hello " + name;
  }

}
```

```
@Client("/")
interface HelloClient {

  @Get("/hello/{name}")
  String hello(String name);

  // Implementation generated
  // at compile time
}
```

MICRONAUT: DEPENDENCY INJECTION

```
@Singleton //Bean definition generated at compile time
class WeatherService {
  Integer currentTemp() { //... }
}

@Controller('/weather')
class WeatherController {

    @Inject WeatherService weatherService
    //DI computed at compile time

    @Get("/")
    Integer currentTemp() {
      return weatherService.currentTemp()
    }
}
```

MICRONAUT: CLOUD NATIVE

SERVICE DISCOVERY

```
//Lookup client from service-discovery registry
@Client(id="billing", path="/billing")
interface BillingClient { ... }
```

RETRYABLE

```
//Automatically retry failing calls
@Client("https://api.external.service")
@Retryable(attempts = '3', delay = '5ms')
interface ExternalApiClient { ... }
```

CIRCUIT BREAKERS

```
//Immediately fail after set number of failures
//Begin accepting calls after `reset` interval
@Singleton
@CircuitBreaker(attempts = '5', reset = '300ms')
class MyService { ... }
```

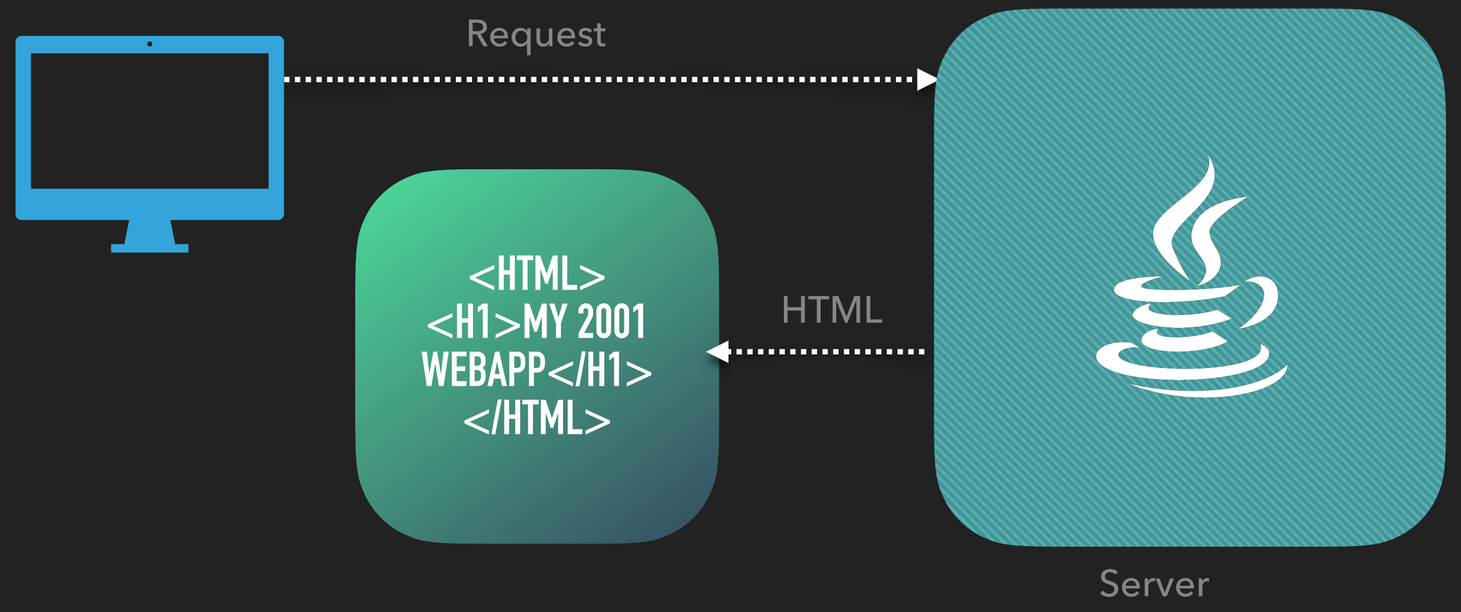
DEVNEXUS™

SPA VS MICROSERVICES?

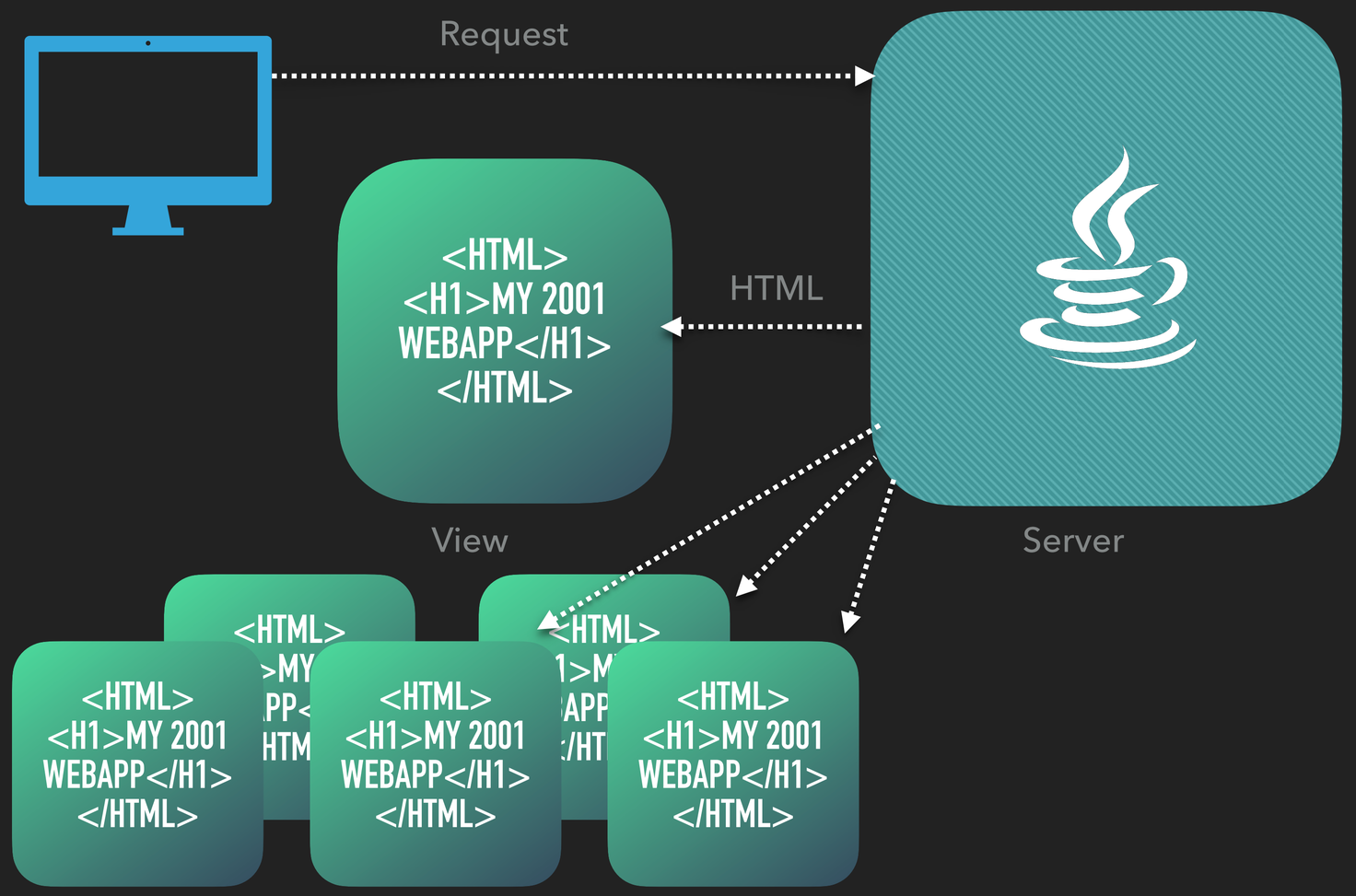
OCI

https://www.123rf.com/photo_9516382_red-apple-with-orange-fillings-genetically-modified-organism.html

TRADITIONAL WEB APPS

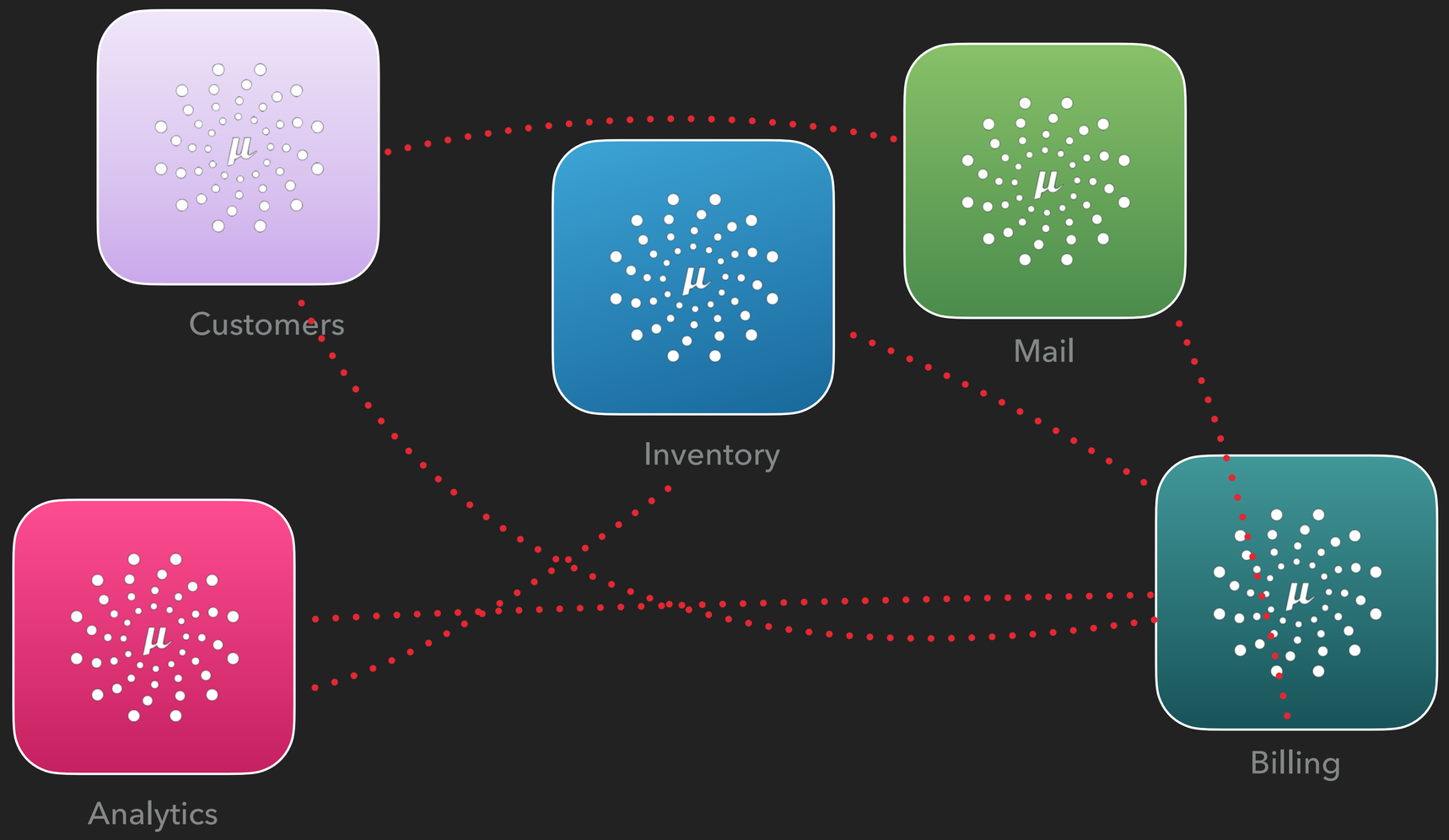


TRADITIONAL WEB APPS



TRADITIONAL REST BACKEND FOR SPA

MICROSERVICE ARCHITECTURE



PROBLEMS FOR SPAS + MICROSERVICES

- Microservice Architectures provide **granularity**
- Granularity offers many benefits, but complicates life for SPAs
- Services may be registered through service discovery (not known URLs)
- Not all clients (SPAs, mobile apps, traditional web apps) require the same data
- The frontend (SPA) shouldn't need to be "aware" of the topology of the backend system

PROBLEMS FOR SPAS + MICROSERVICES

- SPAs typically work best with a single backend API
  - Orchestrating multiple backends for different requests quickly becomes unmanageable
- Microservice architectures prioritize bounded contexts
- The design imperatives for microservices and SPAs are not aligned
- User interfaces are inevitably cross-cutting concerns

USER INTERFACES ARE INEVITABLY CROSS-CUTTING CONCERNS

Somebody, probably

TWO APPROACHES

- All approaches have trade-offs
- None are mutually exclusive
- Pick the best tool for the job

TWO APPROACHES

- 1. Traditional REST* with API Gateway
- 2. Microfrontends

* also asynchronous REST, WebSockets, Server-Sent Events, etc

TWO APPROACHES

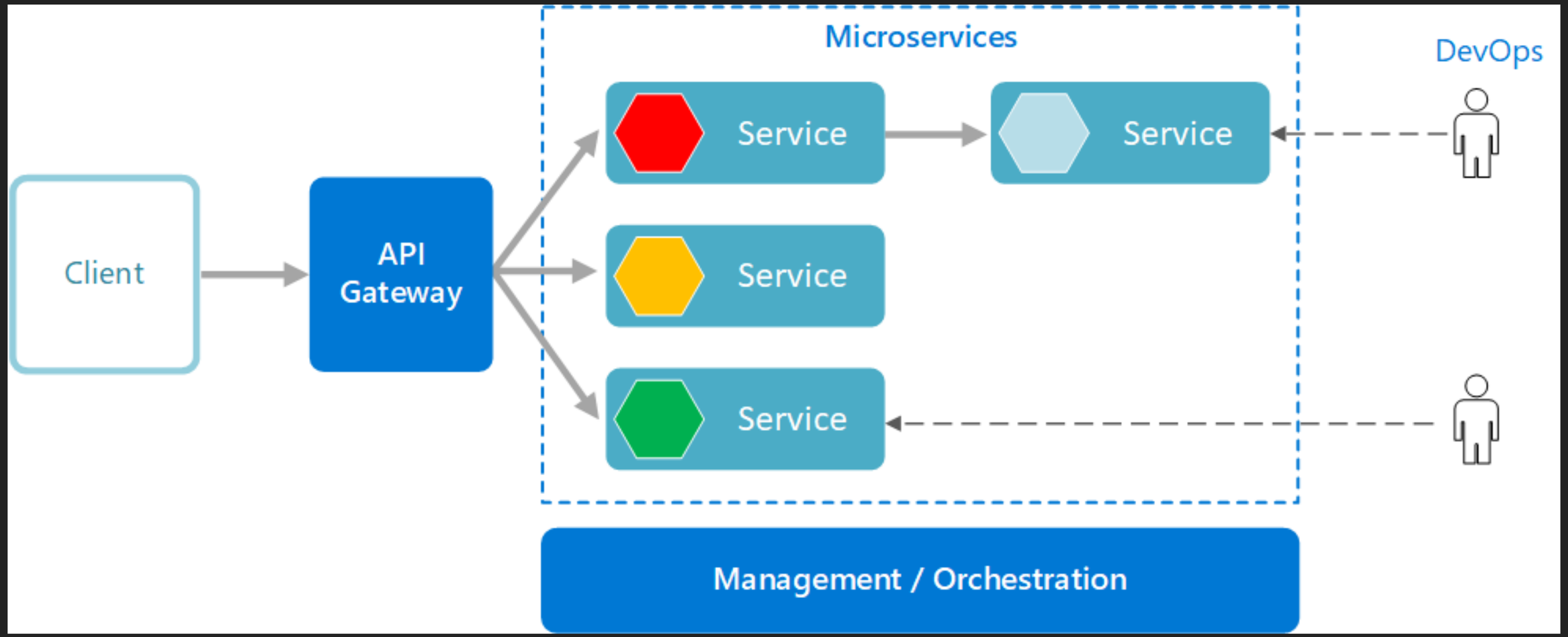
- **1. Traditional REST with API Gateway**
- 2. Microfrontends

* also asynchronous REST, WebSockets, Server-Sent Events, etc

APPROACH #1 TRADITIONAL REST WITH API GATEWAY

- Architectural pattern for microservice-based systems
- Expose client-facing API/s (SPA, mobile, etc) backed by N number of services
- Minimizes integration points for clients - decoupling
- https://microservices.io/patterns/apigateway.html
- https://docs.microsoft.com/en-us/azure/architecture/microservices/design/gateway

APPROACH #1 TRADITIONAL REST WITH API GATEWAY



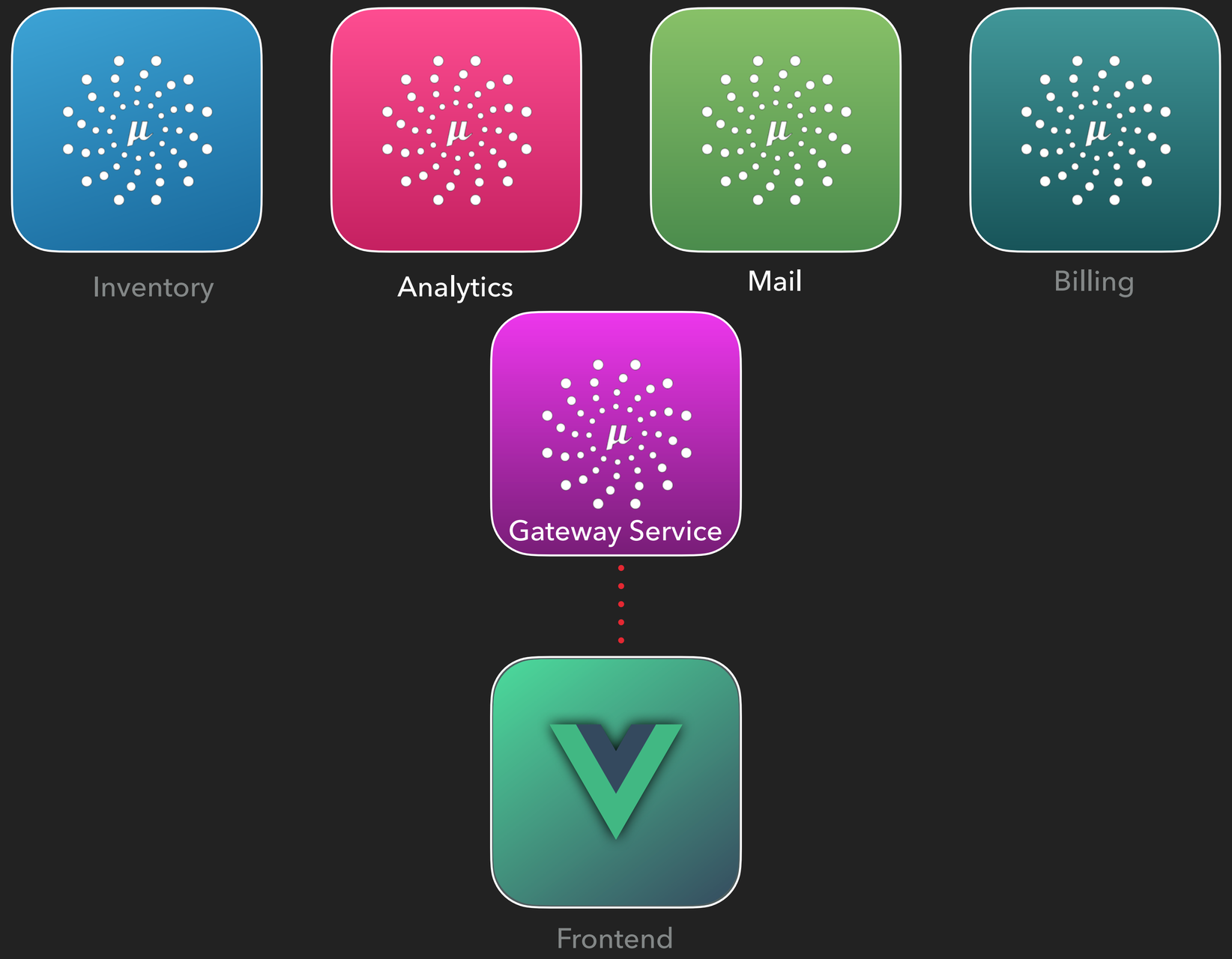
https://docs.microsoft.com/en-us/azure/architecture/microservices/design/gateway

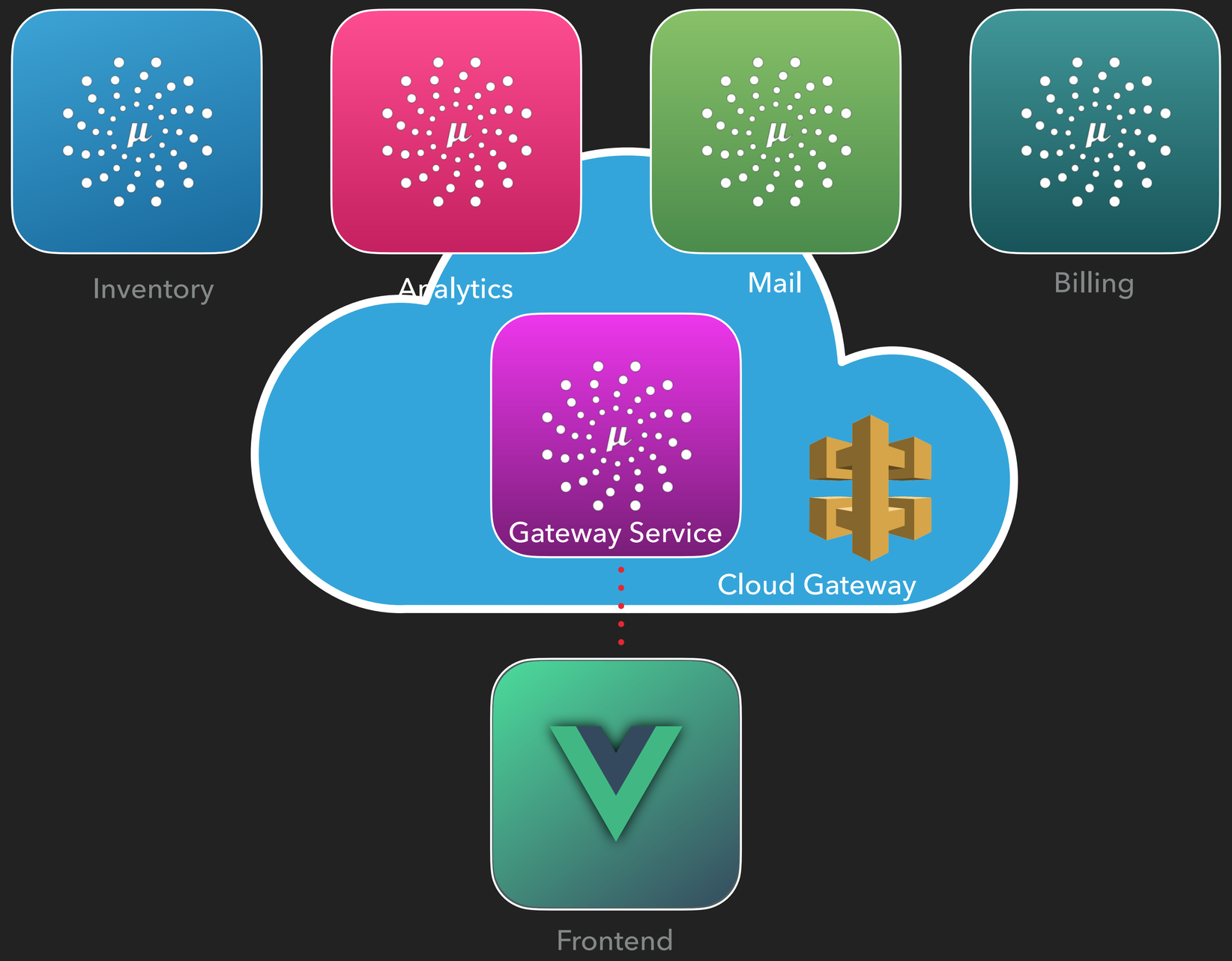
API GATEWAYS

- Many features can be implemented at the API Gateway level:
  - Rate-limiting
  - Logging/tracing
  - Request aggregation
  - API Versioning
- API Gateways should *not* be orchestrators!
- Open Source implementations (Netflix Zuul, Lyft Envoy, krakenD, etc)
- Cloud providers (e.g, AWS) often supply their own API Gateway product
- API Gateways can also be implemented as standalone services

Backend

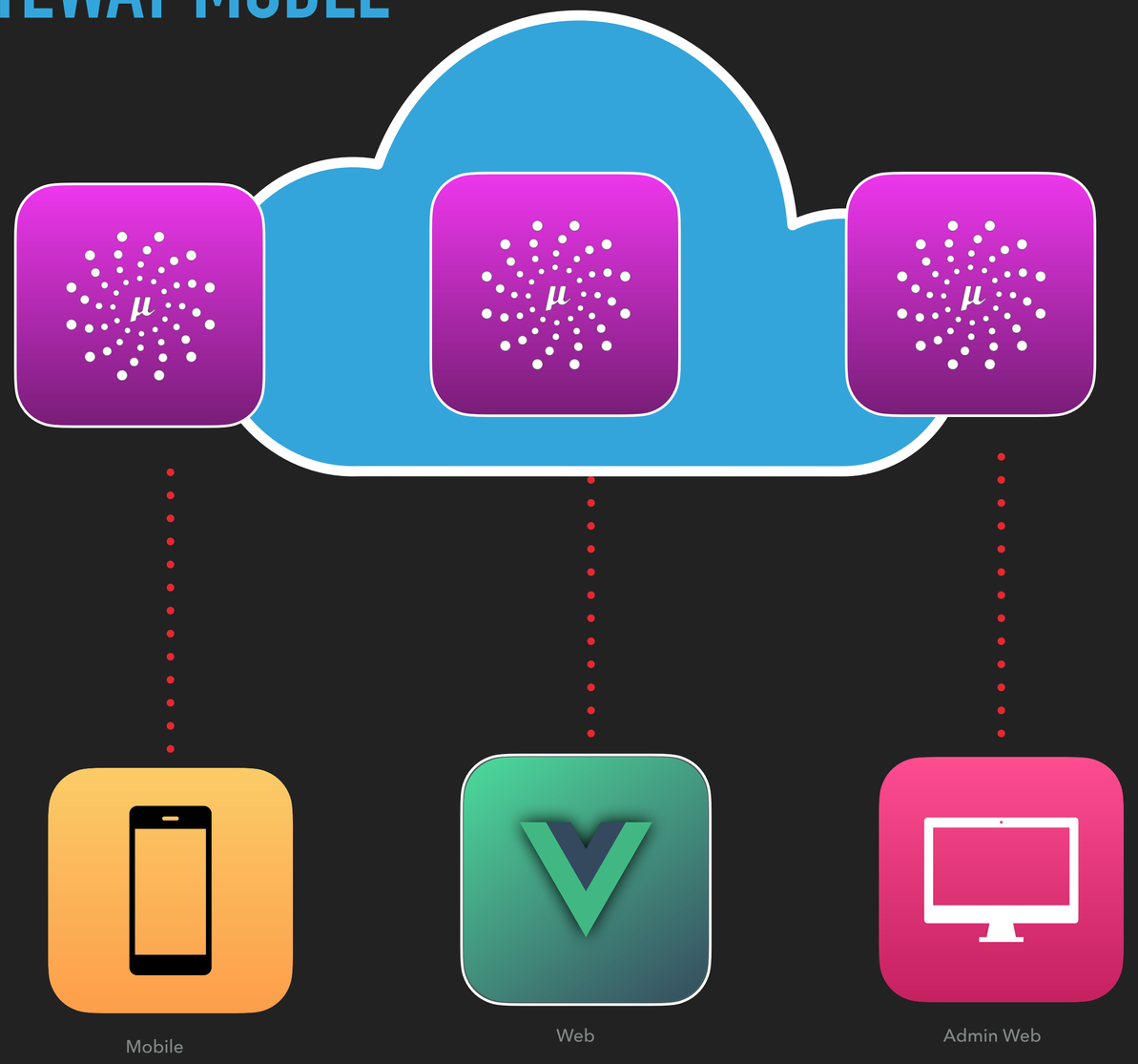
Frontend


Inventory
Analytics
Mail
Billing
Gateway Service
Frontend


Inventory
Analytics
Mail
Billing
Gateway Service
Cloud Gateway
Frontend

"BACKEND PER FRONTEND" GATEWAY MODEL

- Partition your API
- Support different client needs (web vs mobile etc)



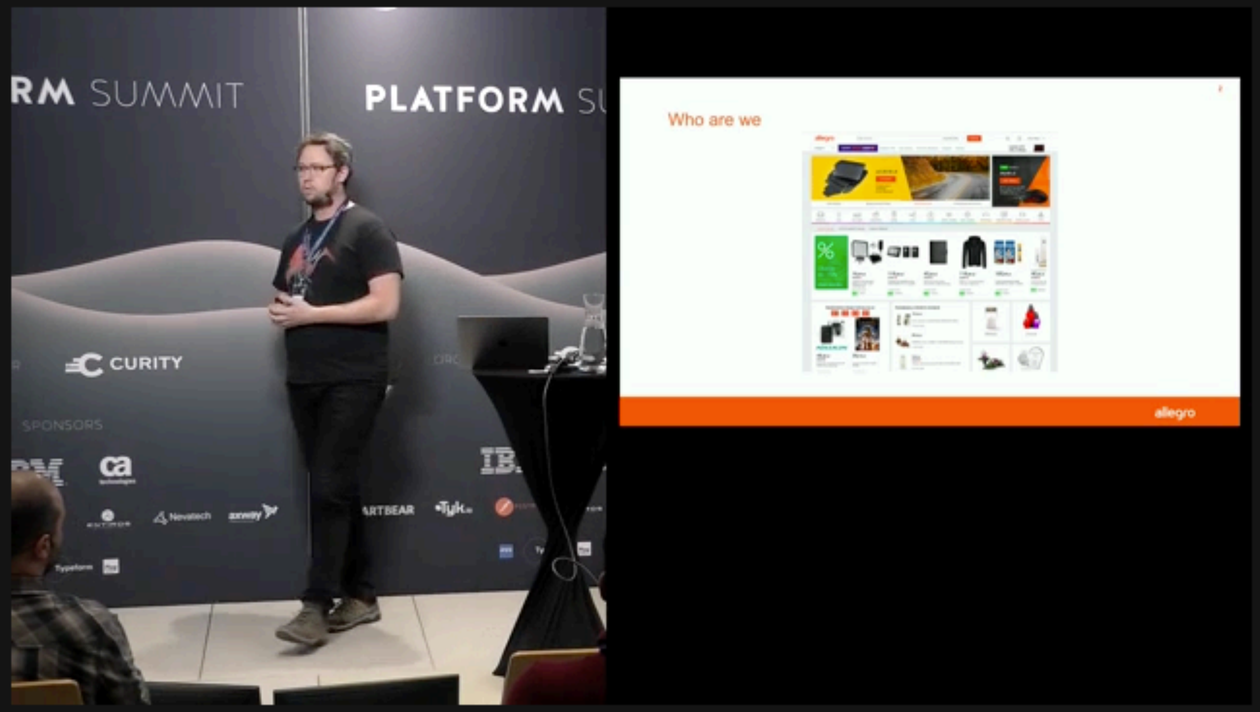
BUILDING AN API GATEWAY SERVICE WITH MICRONAUT

- Consistent APIs between controller (service) and client (gateway)
- Use of shared API libraries can simplify development
  - Shared API: `interface ProductAPI` - specifies API for product resource
  - Service: `ProductController implements ProductAPI` - specifies business logic
  - Gateway: `ProductClient extends ProductAPI` - consumes backend API on behalf of edge-clients
- Support for API versioning, tracing, load-balancing, API docs, etc

SHARED API

```
@Controller("/")
class HelloController {

  @Get("/hello/{name}")
  String hello(String name) {
    return "Hello " + name;
  }

}
```

```
@Client("/")
interface HelloClient {

  @Get("/hello/{name}")
  String hello(String name);

}
```



Should I Make My Own API Gateway?

3,016 views • Nov 4, 2018 28 DISLIKE SHARE CLIP SAVE ...

Nordic APIs
8.51K subscribers

SUBSCRIBE

This is a session given by Michal Trojanowski at Allegro at Nordic APIs 2018 Platform Summit on October 24th, in Stockholm, Sweden.

- https://www.youtube.com/watch?v=YO6Sg4yaqC0

DEVNEXUS™

RESTFUL API

OCI

```
@Controller("/book")
class BookController {

    @Post
    HttpResponse<BookDetails> save(@Valid @Body BookDetails bookDetails) { /* .. */}

    @Put
    HttpResponse<BookDetails> update(@Valid @Body BookDetails bookDetails) { /* .. */}

    @Delete("/{id}")
    HttpResponse delete(Serializable id) { /* .. */}

    @Get("{?max,offset}")
    @Transactional(readOnly = false)
    HttpResponse<List<Book>> list(@Nullable Integer max, @Nullable Integer offset) { /* .. */}

    @Get("/{id}")
    @Transactional(readOnly = true)
    HttpResponse<BookDetails> get(Serializable id) { /* .. */}

    HttpResponse<Integer> count() { /* .. */}

}
```

MICRONAUT & REST

- Declarative Routes via method annotations:
  - `@Get`, `@Put`, `@Post`, `@Delete`
- JSON binding/rendering via Jackson
- Request Arguments via annotations:
  - `@Header`, `@Body`, `@CookieValue`, `@QueryValue`

JACKSON: JSON BINDING

```
public class Author {

    private String name;

    @JsonSerialize(MySerializer.class)
    private Date birthday;
}
```

```
@Post("/")
public HttpResponse<Author> save(
    @Body Author author) {

    if(bookRepository.save(author)) {
      return HttpResponse.ok();
    } else {
      /* handle error */
    }
}
```

https://www.baeldung.com/jackson-annotations

```
fetch("http://localhost:8080/
author/", {
    method: "POST",
    headers: new Headers({
        "Content-Type": "application/
json"
    }),
    body: JSON.stringify({
        name: "Author's Name",
        birthday: "01/31/1985"
    })
})
```

JAVASCRIPT

JACKSON: JSON RENDERING

```java
@JsonIgnoreProperties({"id", "version"})
public class Book {

    private Long id;
    private Long version;

    @JsonProperty("name")
    private String title;
    private Author author;
    private Integer pages;
    private List<String> tags;
}
```

```java
@Get("/{id}")
public Book show(Serializable id) {

    return bookRepository.get(id);
}
```

```json
{
  “name": "Title Here”,
  "author": {
    "name": "Author"
  },
  "pages": 150,
  "tags": [
    "tech",
    "bestseller"
  ]
}
```

JSON

https://www.baeldung.com/jackson-annotations

ENABLING CORS

- CORS support included in Micronaut
- Disabled by default
- Can specify allowed origins, methods, headers, max age, and more.

APPLICATION.YML

```yaml
micronaut:
    application:
        name: my-app
    server:
        cors:
            enabled: true
```

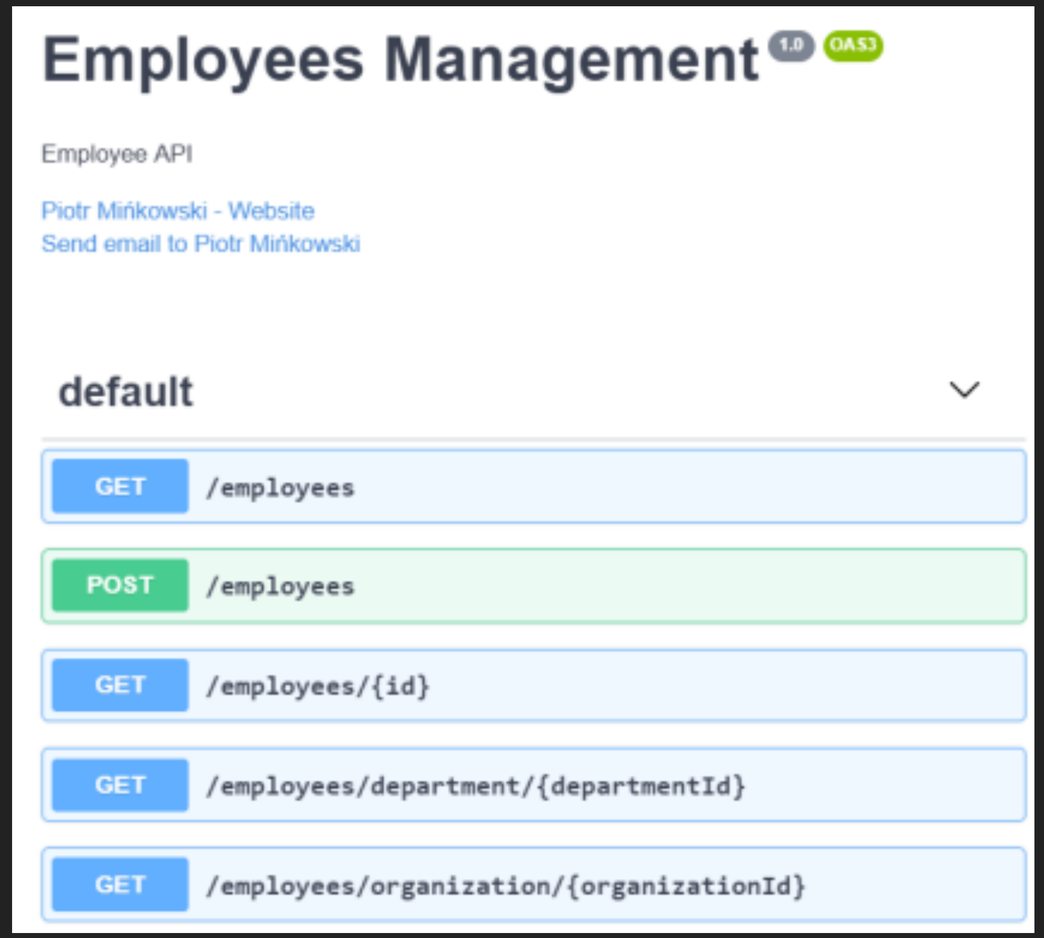
DEVNEXUS™

API DOCUMENTATION

OCI

OPEN API DOCUMENTATION

- OpenAPI is a specification for defining API docs
- Tools like Swagger can generate interactive API docs from the OpenAPI YAML
- Can show sample request/ responses, constraints, headers, etc



OPENAPI INITIATIVE

Swagger

OPEN API DOCUMENTATION

- Micronaut can generate OpenAPI (Swagger) YAML definitions at compilation time
- Standard Micronaut annotations (`@Controller`, `@Get`, `@Consumes`, etc) and method return types (POJOs) will be analyzed and corresponding Swagger YML written to the file
- Standard Swagger annotations can be used to customize/override the generated YAML
- Micronaut can handle merging of OpenAPI schemas from multiple modules (e.g., when using Micronaut Security)

OPENAPI INITIATIVE

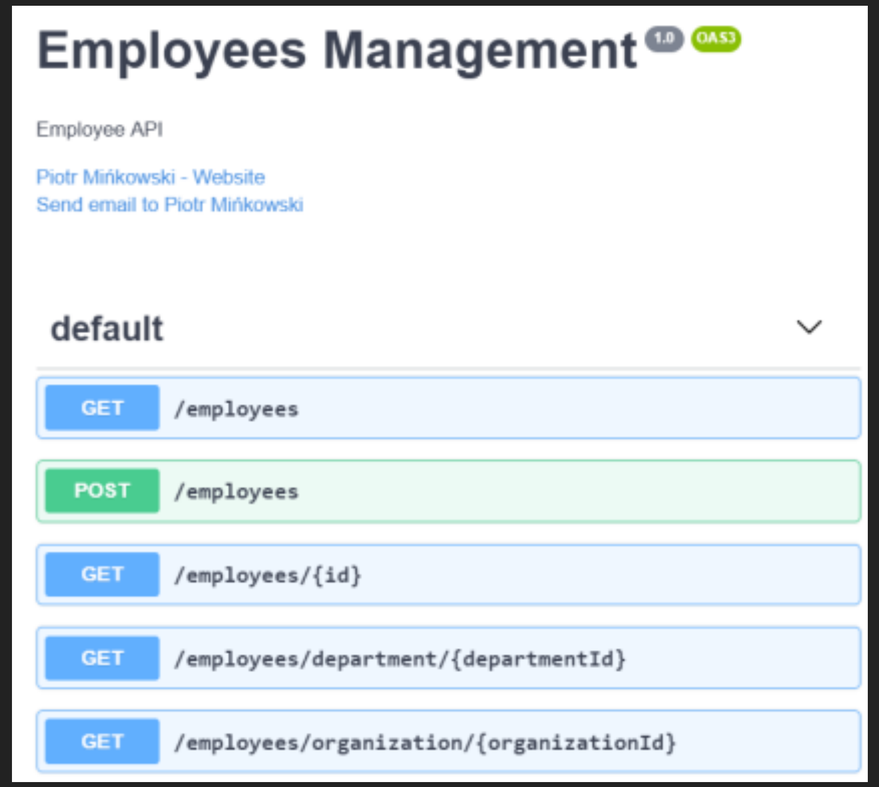
Swagger

OPEN API DOCUMENTATION

Configuration to expose Swagger YAML over the server:

src/main/resources/application.yml

```
micronaut:
  router:
    static-resources:
      swagger:
        paths: classpath:META-INF/swagger
        mapping: /swagger/**
```

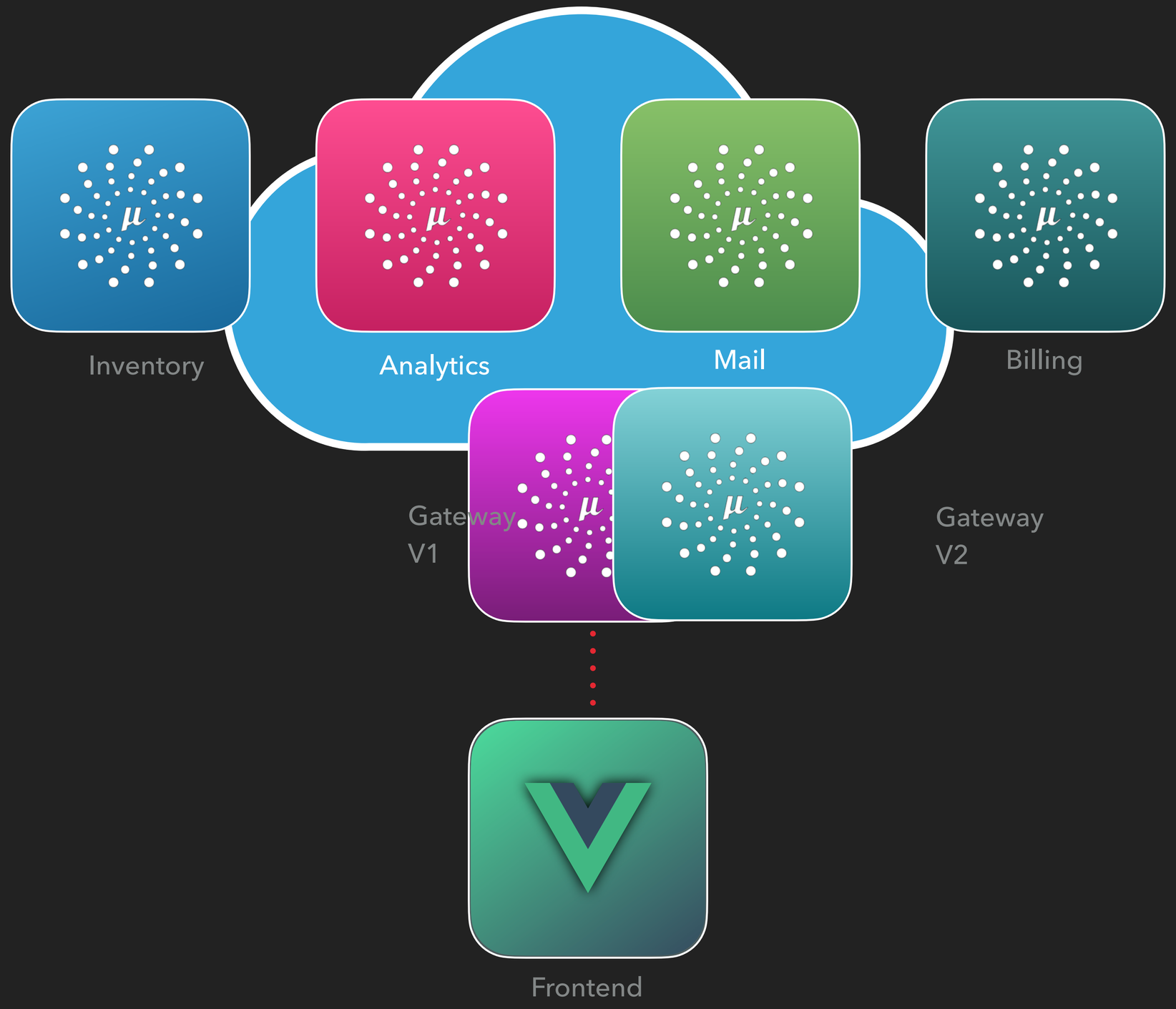


https://micronaut-projects.github.io/micronaut-openapi/latest/guide/index.html

DEVNEXUS™

API VERSIONING

OCI


Inventory
Analytics
Mail
Billing
Gateway
V1
Gateway
V2
Frontend

API VERSIONING WITH MICRONAUT

- Version annotation:

```
@Version("1")
@Get("/${version}/user/profile")
```

src/main/resources/application.yml

```
micronaut:
  router:
    versioning:
      enabled: true
      default-version: 3
```

- Can also be used to version HTTP client requests (same annotation API)
- By default, API version is resolved from the X-API-VERSION header, or the api-version request parameter (both of these are configurable)

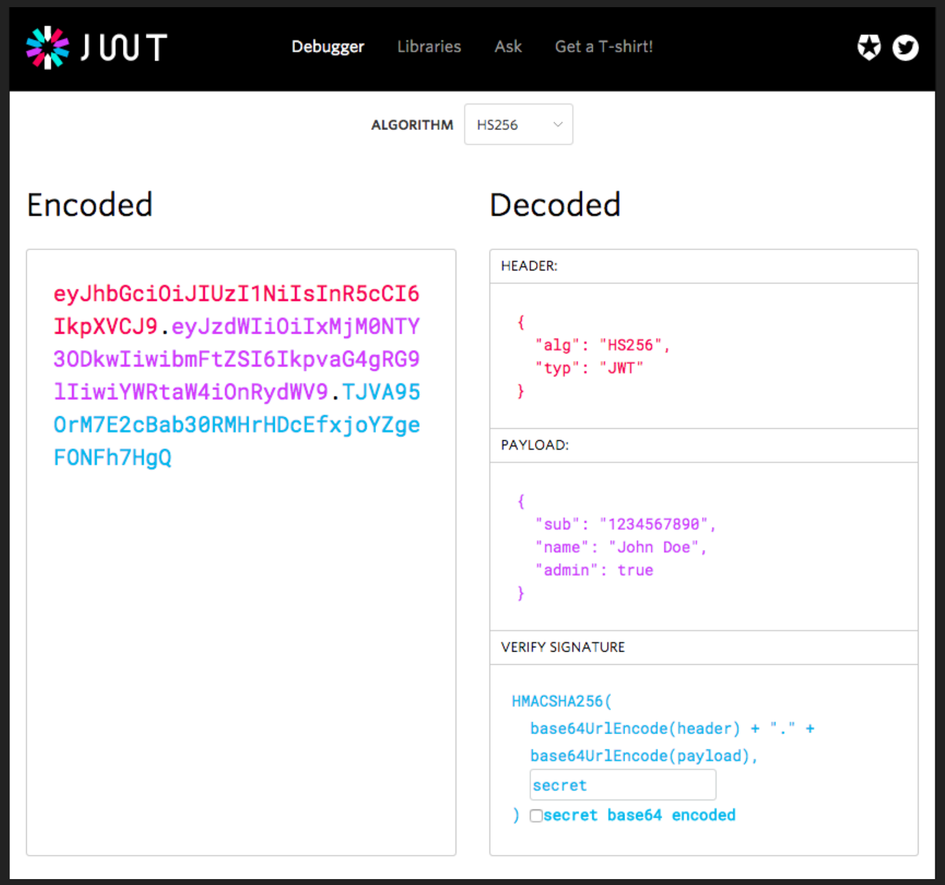
DEVNEXUS™

SECURITY WITH JWT

OCI

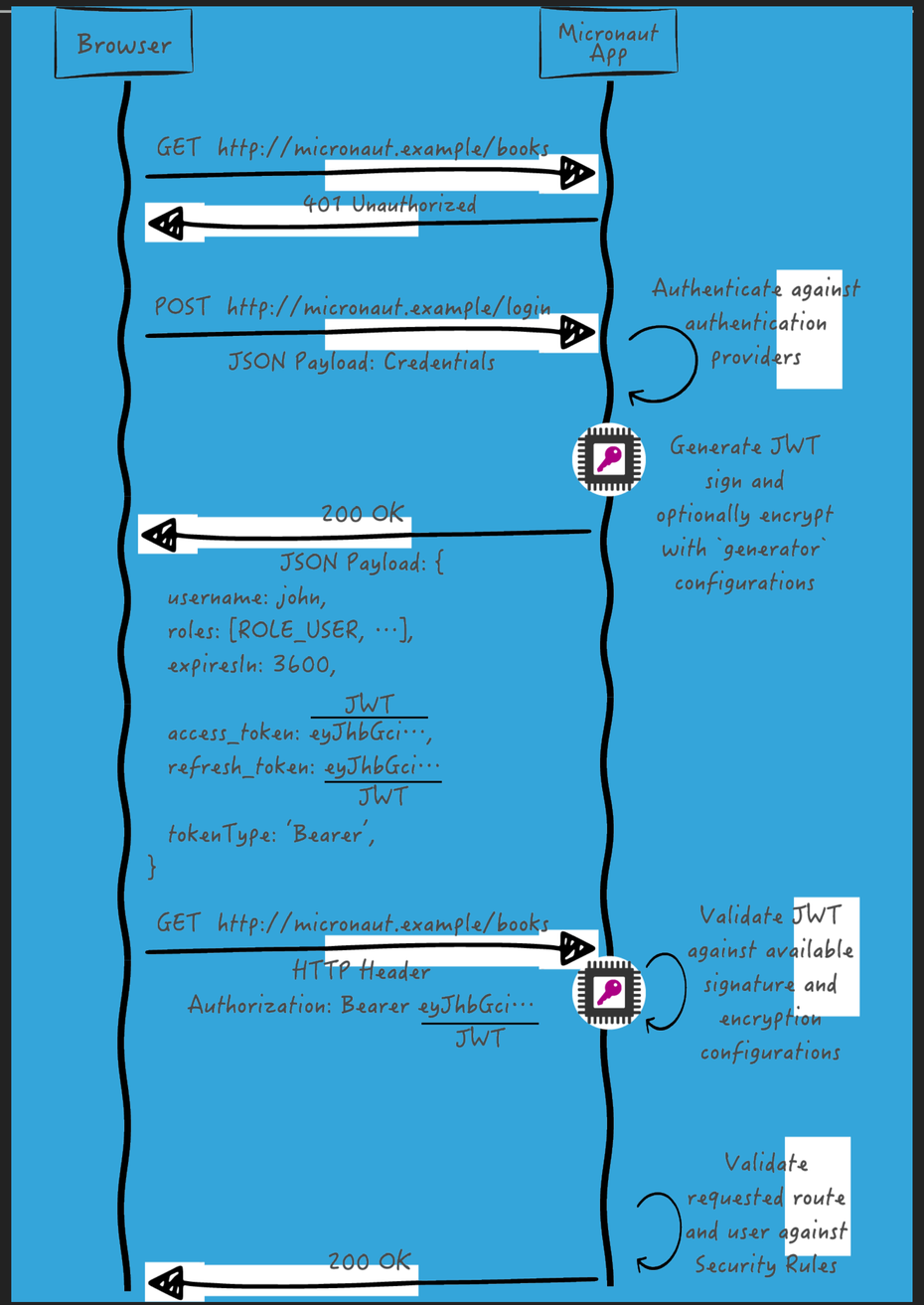
JWT: JSON WEB TOKEN

- Open standard for representing claims securely between two parties
- Tokens can be signed with either a secret or public/private key
- Standard approach for stateless authentication
- Ideal for transmitting authentication & authorization data between microservices and single-page-apps



STATELESS JWT SECURITY

- Unauthorized request is made to API
- Responds with 401
- Client POSTs to login endpoint
- Server responds with JWT
- Client includes access token in the Authorization header for subsequent requests
- Server validates the incoming token
- If authorized, server responds with resource



MICRONAUT SECURITY

- Core Micronaut Library - supports JWT, OAuth 2.0
- Annotation-based API & config-based URL mappings
- Support for **token propagation**
- Supports [RFC 6750 Bearer Token]
- JWTs can be read from cookie

BUILD.GRADLE

```
dependencies {
   implementation "io.micronaut:micronaut-security-jwt"
}
```

APPLICATION.YML

```yaml
micronaut:
  security:
    enabled: true
    token:
      jwt:
        enabled: true
        signatures:
          secret:
            generator:
              secret: changeMe
```

@SECURED ANNOTATION

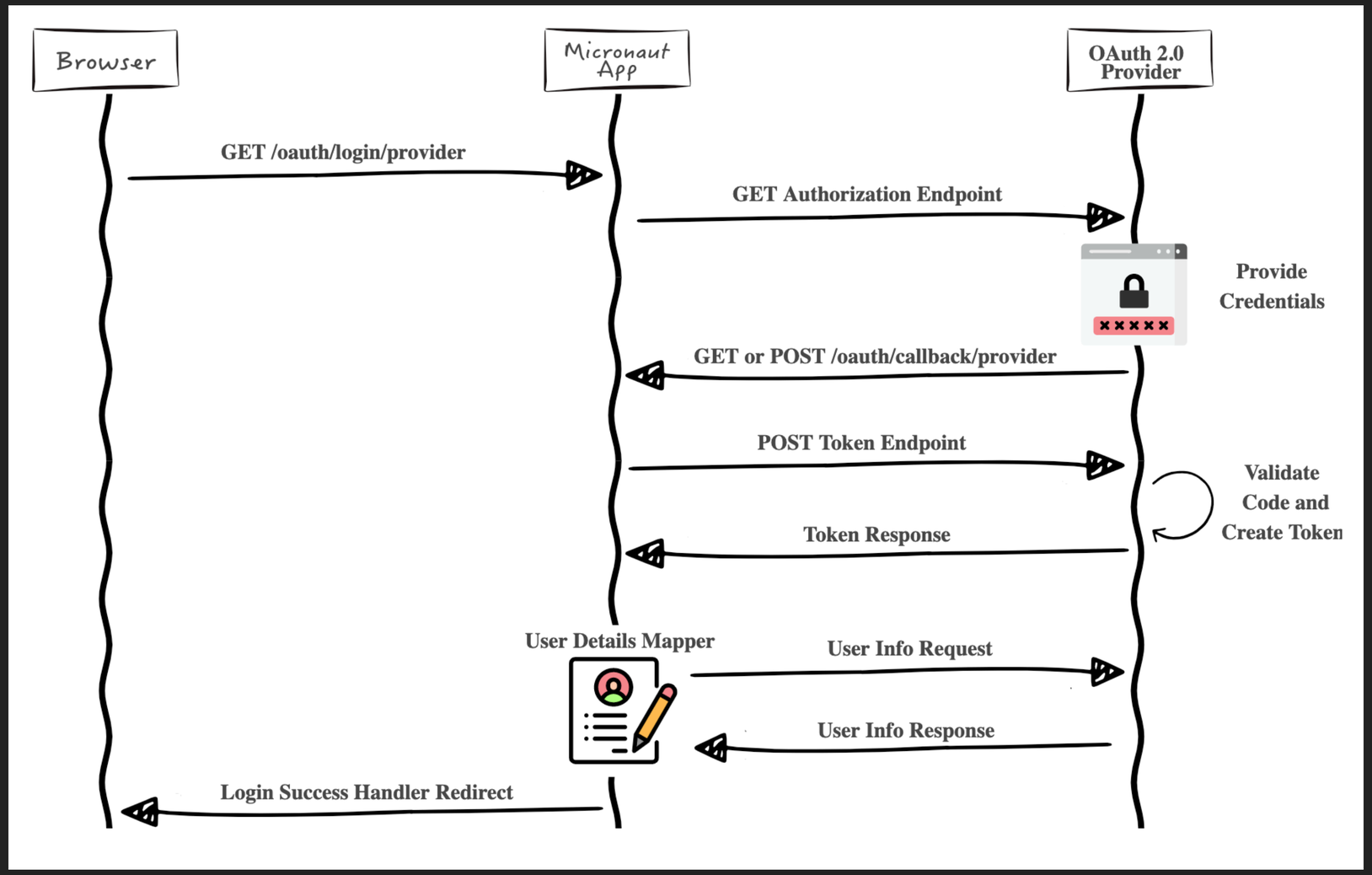
- @Secured annotation applied to controllers and methods
- All routes blocked by default
- Can require authentication and/or authorization (role-based)
- Alternative: JSR-250 security annotations are also supported: `@PermitAll`, `@RolesAllowed`, `@DenyAll`

```
import java.security.Principal;

@Secured("isAuthenticated()")
@Controller("/")
public class HomeController {

    @Get("/")
    String index(Principal principal) {
        return principal.getName();
    }

    @Secured({"ROLE_ADMIN", "ROLE_X"})
    @Get("/classified")
    String classified() {
        return /* REDACTED */;
    }
}
```

OAUTH 2.0

- Delegate authentication to a third-party provider (or custom provider)
- Requires a mapping between the provider's auth and user identity and authorization (within your application)
- Typically configured via a client ID/secret pair and a callback URL

OAUTH 2.0


Browser
Micronaut App
OAuth 2.0 Provider
GET /oauth/login/provider
GET Authorization Endpoint
Provide Credentials
GET or POST /oauth/callback/provider
POST Token Endpoint
Validate Code and Create Token
Token Response
User Details Mapper
User Info Request
User Info Response
Login Success Handler Redirect

MICRONAUT SECURITY & OAUTH GUIDES

GUIDES FILTERED BY #SECURITY

Micronaut Basic Auth
JAVA KOTLIN GROOVY

Session based authentication
JAVA GROOVY KOTLIN

Micronaut JWT Authentication
JAVA GROOVY KOTLIN

Micronaut JWT authentication via Cookies
JAVA GROOVY KOTLIN

LDAP and Database authentication providers

https://guides.micronaut.io/tags/security.html

GUIDES FILTERED BY #OAUTH2

Secure a Micronaut app with Okta

Secure a Micronaut app with Google

Secure a Micronaut app with Cognito

Secure a Micronaut app with Github

Secure a Micronaut app with Github

https://guides.micronaut.io/tags/oauth2.html

DEVNEXUS™

TOKEN PROPAGATION

OCI

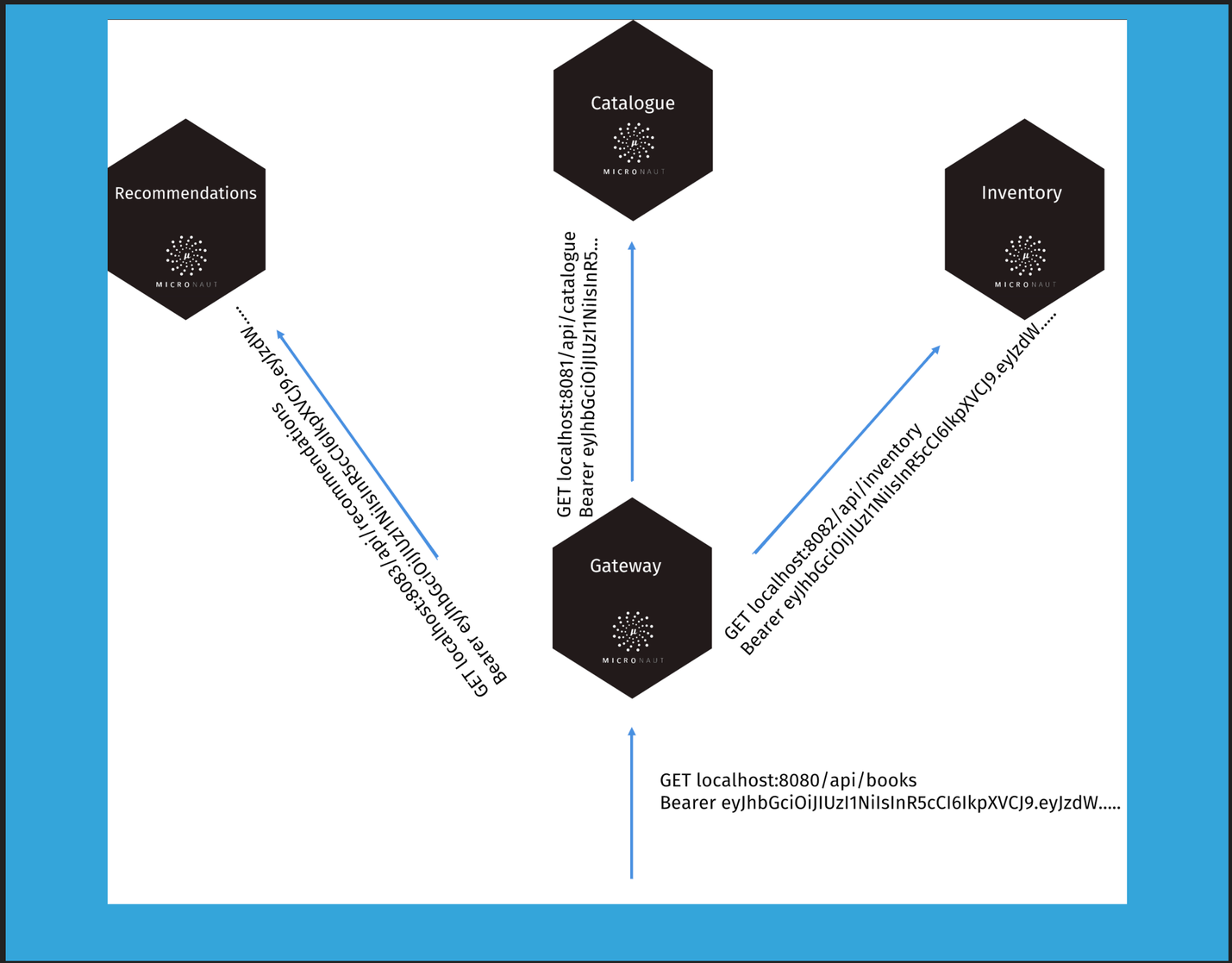
TOKEN PROPAGATION

- An access token is embedded within the request
- Token can be stored as a cookie, or within an HTTP Header
- Services to which tokens should be propagated can be specified via config
- Allows each service to enforce authentication/ authorization

APPLICATION.YML

```yaml
micronaut:
  security:
    enabled: true
    token:
      jwt:
        enabled: true
      writer:
        header:
          enabled: true
      propagation:
        enabled: true
        service-id-regex: "inventory"
```

SINGLE PAGE APPS FOR A MICROSERVICE ARCHITECTURE



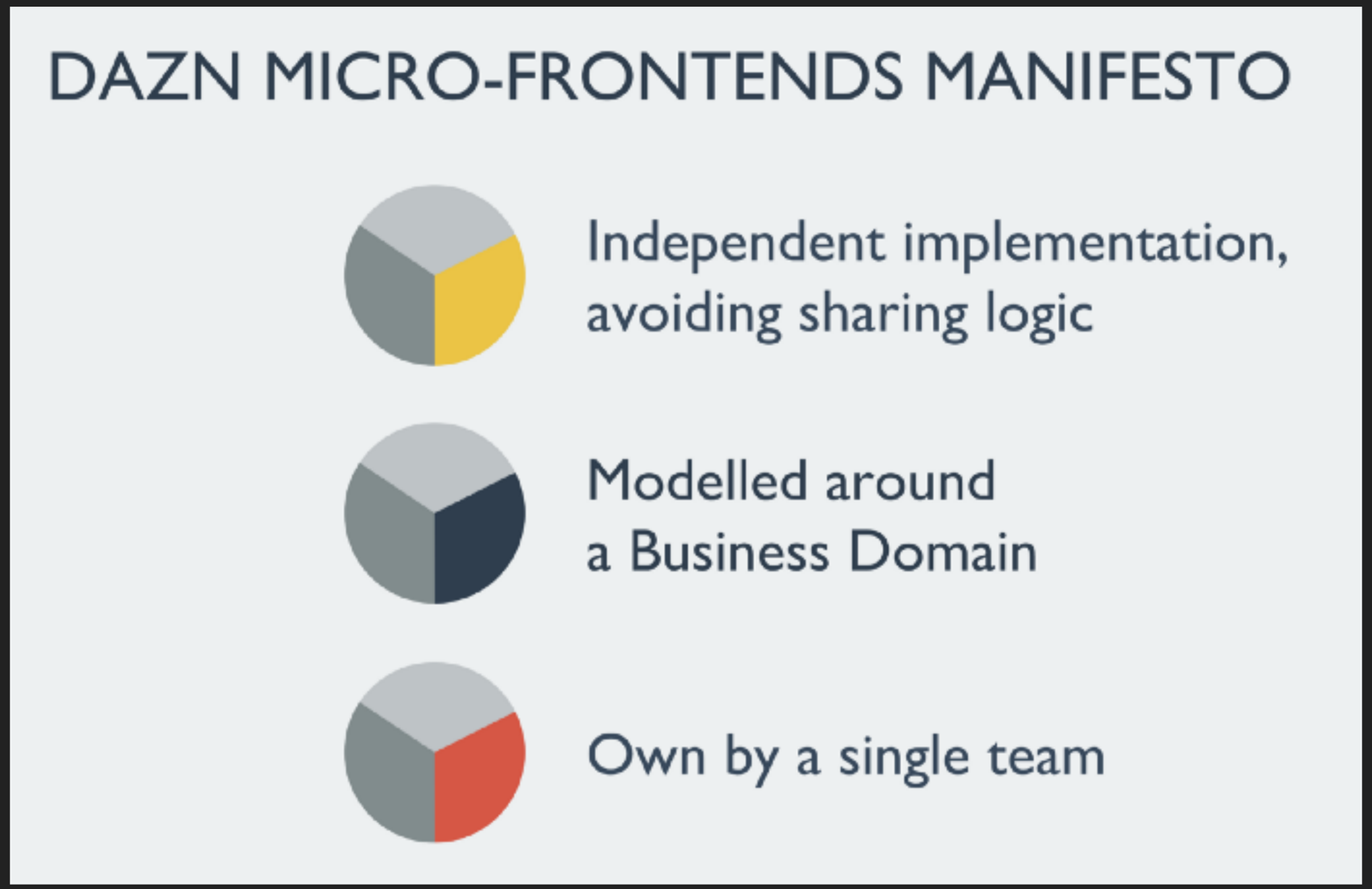
https://guides.micronaut.io/micronaut-token-propagation

TWO APPROACHES

- 1. Traditional REST with API Gateway
- **2. Microfrontends**

APPROACH #2 MICROFRONTENDS

- An application of microservice architecture to frontend development
- Independent frontend applications with their own “bounded context” and (potentially) their own backend API
- Typically makes use of frontend tooling (e.g, Webpack) or IFrames to stitch together a cohesive UI from multiple apps
- Well known examples: Spotify, IKEA
- https://microfrontends.info/microfrontends/



- https://medium.com/dazn-tech/micro-frontends-the-future-of-frontend-architectures-5867ceded39a

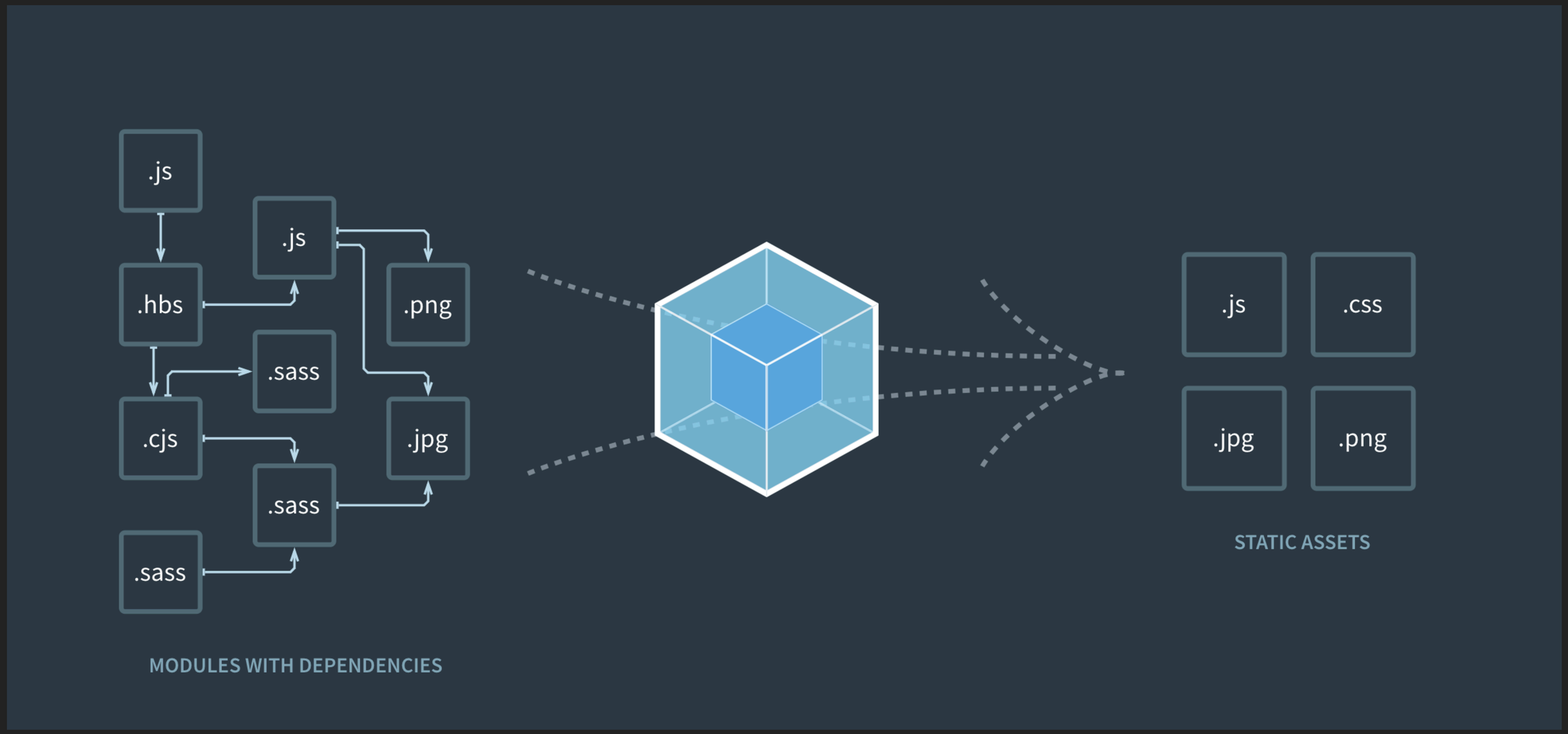
HOW TO MICROFRONTEND

- 1. Build-time integration; individual SPAs as libraries
- 2. Server-side routing between multiple SPAs
- 3. Composition - unifying multiple SPAs into a single UX using frontend tooling (Webpack)

WEBPACK

- JavaScript Module bundler
- Supports 100s of asset types
- Links/bundles dependencies in JavaScript apps
- Extremely configurable (too much?)
- Builds a dependency graph from one or more “entries”
- Outputs a (optionally minified, “chunked”) JS/CSS bundle
- Supports hot-reloading

WEBPACK


.js
.hbs
.cjs
.sass
.js
.sass
.sass
.png
.jpg
MODULES WITH DEPENDENCIES
.js
.css
.jpg
.png
STATIC ASSETS

WEBPACK MODULE FEDERATION PLUGIN

- Creates a single application from multiple builds
- Composes dynamic “remotes” that point to SPAs running in different contexts
- https://webpack.js.org/concepts/module-federation/
- Jack Herrington’s video: https://youtu.be/D3XYAx30CNc

DEVNEXUS™

MICROFRONTENDS: THE DEMO

OCI

DEVNEXUS™

THANK YOU!

- Twitter: @ZacharyAKlein
- LinkedIn: http://linkedin.com/in/zacharyaklein/
- https://objectcomputing.com

OCI