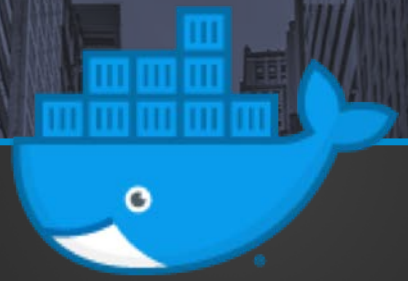
DOCKER FUNDAMENTALS

DEMYSTIFYING MICROSERVICES, CLOUD NATIVE INFRASTRUCTURE AND INFRASTRUCTURE AS CODE

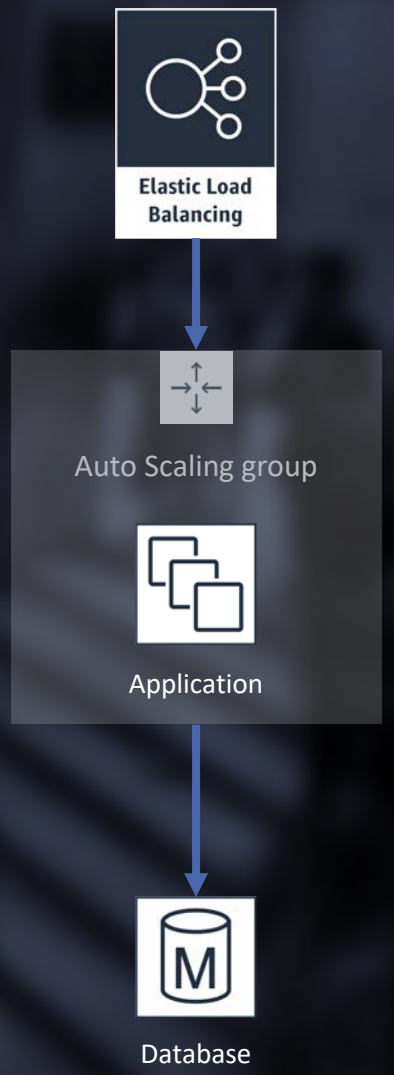
MICROSERVICES



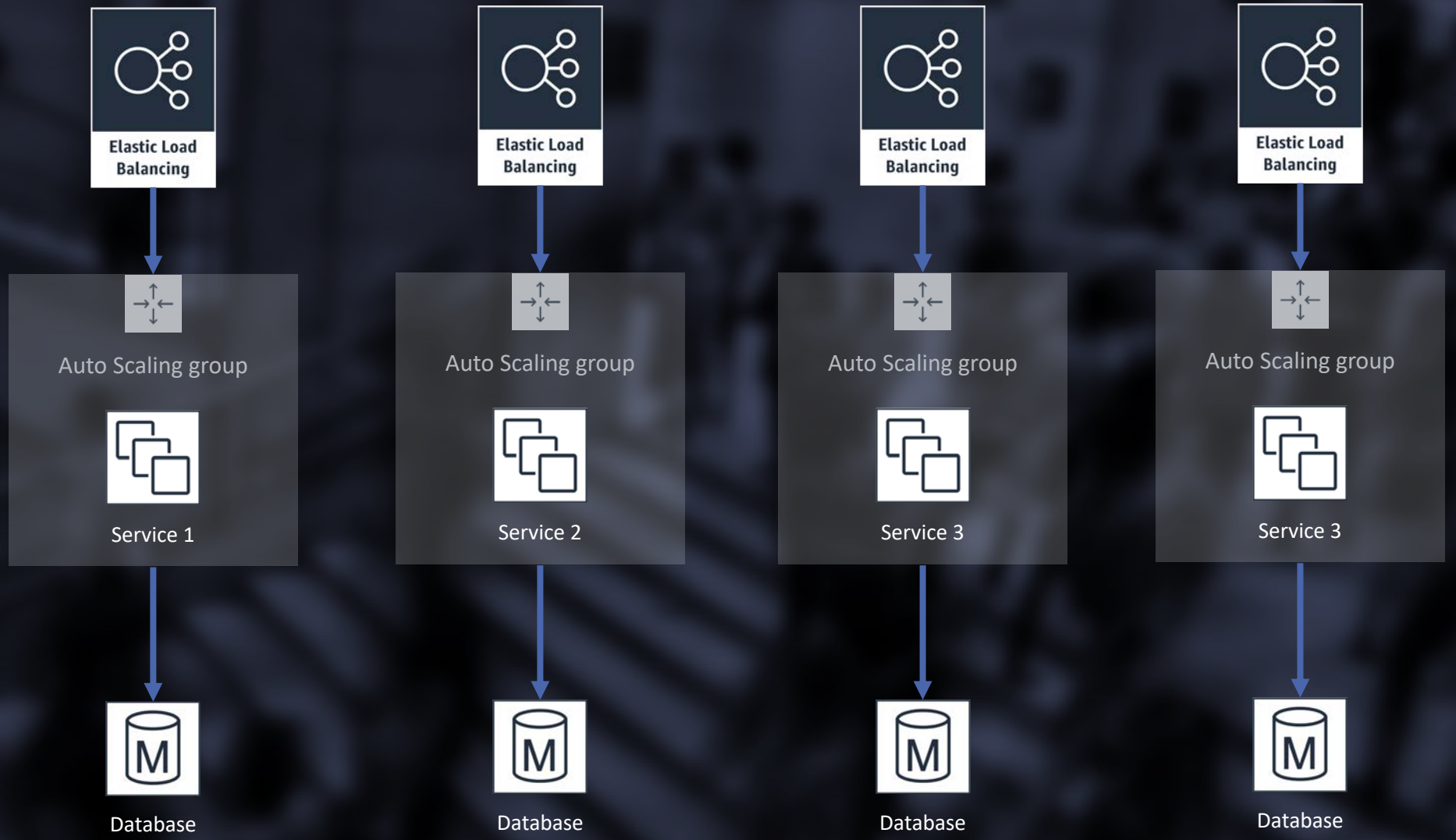
MICROSERVICES - OVERVIEW

- Small, fully independent and autonomous application. Isolated from other MS's, even with own DB.
- Responsible for a logical part of a larger application "ecosystem". Singular responsibility.
- Can be any language. Developers on the same project can have completely different skillsets if necessary.
- Loosely coupled to other MS's. Communicate with other MS's using standardized protocols. Often a REST API or a message bus.

MICROSERVICES – MONOLITH EXAMPLE



MICROSERVICES – MICROSERVICE EXAMPLE



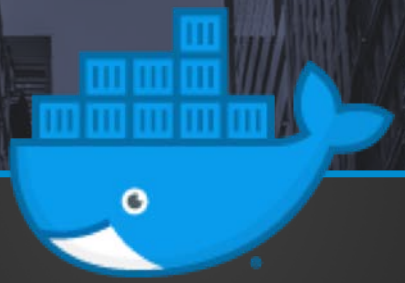
MICROSERVICES - BENEFITS

- Small and manageable pieces. No massive monoliths.
- No need to have every developer on the same language and framework.
- Testing is more limited since you don't have to test an entire monolith.
- Can have their own development and release cycle.
- Can be deployed independently without breaking other pieces. Can be done faster.
- Can be operated at a more granular level. Independent scaling of MS's. Faults are isolated. "Graceful degradation".
- Encourages encapsulation of business logic.
- Smaller size enables Agile development.

MICROSERVICES - DISADVANTAGES

- Much more complexity of the overall system. Way more moving parts. Often requires technologies that might not otherwise be used, such as message buses. Requires seasoned architects and leads to keep disaster at bay.
- Often requires more infrastructure to support, as each component has it's own overhead. With a monolith, overhead can often be shared.
- Coordination between dependent MS's can sometimes be more complex than a monolith.
- More parts to deploy.

MICROSERVICES: DISCUSSION



ECOMMERCE APP

MICROSERVICES – DISCUSSION

We have an ecommerce application. It has a product catalog, cart, payment processing, fulfillment, communications with customers, etc. Think amazon.com.

With your group, discuss how you would break this down into microservices. What are the components? How do they communicate? How is data stored? How is customer experience affected by your choices?

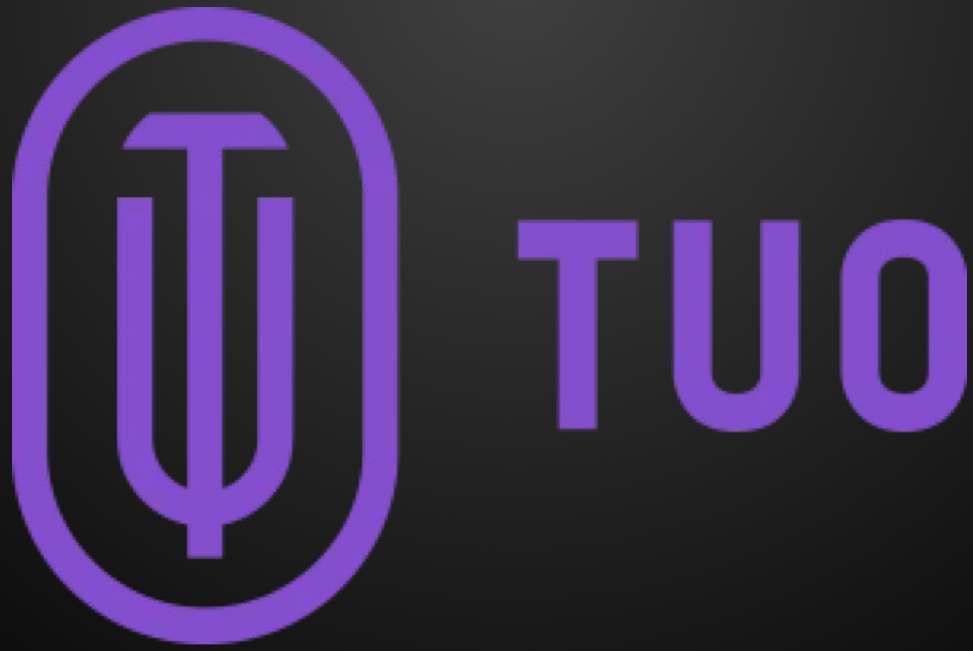
Design a microservice oriented system. Write notes, make diagrams, etc. Play the part of architect. Pick someone to present your solution to the class.

MICROSERVICES – DISCUSSION

Major functionality:

- Product catalog (admin functions, search, recommendations, related products, etc.)
- Shopping cart (and email reminders about items in cart)
- Order processing (payment processing, emails, etc)
- Fulfillment (send to warehouse, inventory allocation, picking, shipping, backorders, etc.)
- Reporting (internally and to the customer)
- Email marketing
- Social media integration
- Customer order cancel and refunds
- Shipment tracking
- Bonus: Digital content! Movies, music, etc.

MICROSERVICES: CASE STUDY



TUO

MICROSERVICES – CASE STUDY

TUO is a cloud connected LED smart light bulb that controls your body's circadian rhythm by stimulating melatonin producing cells in the retina.

TUO follows microservice principles using Serverless technologies. Services include:

- UI service – Serves up the Angular UI (also produced IOS/Android apps)
- Admin UI service – Serves up a web-based administration site
- Config service – Provides distributed configuration information
- Account API - User information and authentication
- IoT service – Communication with bulbs
- Device management API – Management of bulbs, firmware updates, etc.
- Group API – Management of groups of devices
- Mode API – Management of device modes
- Schedule API – Bulb schedule management and execution of schedules
- Content API – Dynamic content served in the app (like help info)

TUO

thetuolife.com

STATELESS APPLICATIONS - OVERVIEW

- No session state saved on a server
- Enables maximum use of server resources – no unevenly weighted servers
- Enables disposable infrastructure – servers can be trashed without concern for botched user sessions.
- Enables rapid scaling – both out and in
- Enables fast recovery from server failures. Users aren't stuck to a failed server.
- Enables easy and fast deployments.
- Session state stored on client and passed with each request. Example: cookies, JSON Web Tokens

STATELESS APPLICATIONS - JWT

- A great resource is www.jwt.io. Get libraries and debug/see contents of a JWT you've generated.
- Three sections separated by a period

Header

```
{
 "alg": "HS256",
 "typ": "JWT"
}
```

Payload

```
{
 "sub": "1234567890",
 "name": "John Doe",
 "admin": true
}
```

Signature

```
HMACSHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

STATELESS APPLICATIONS - JWT

- Publicly readable! This is not secret information!
- Content is not alterable without the encoding key. The stronger/longer your key, the more secure your JWT
- If a malicious party gets the JWT, they can imitate your user. They must be stored in a secure place!


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3O
DkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflK
xwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

CLOUD-NATIVE APPLICATIONS- OVERVIEW

- Take advantage of flexibility of the cloud to consume and release resources at will without paying for them to be idle.
- Usually refers to containerizing workloads to abstract the application components on top of “plain vanilla” compute resources.
- Takes advantage of orchestration systems.
- Highly automated operations and DevOps. Take advantage of auto-healing and auto-scaling.
- Can often refer to the use of managed services by a cloud provider, such as Amazon Relational Database Service.

SINGLE RESPONSIBILITY PRINCIPLE - OVERVIEW

- Do one thing, and do it well
- Completely encapsulate the logic and persistence
- No intermingling of access to data or logic!
- Example: if one service needs order details, it must request this from the order service, which is wholly responsible for all operations related to orders, including retrieving it from the order database and serving it back to the client service as JSON.

AUTO-SCALING AND HEALING - OVERVIEW

- Services should scale horizontally, not vertically. What does this mean? Terms: scale in/out vs. up/down.
- What are some metrics that would trigger scaling?
- How does scaling differ from healing? How are they related?
- What geographic considerations are there?
- How quickly should you scale out? Scale in?

- Real-world example: Equator.com outage

REST API- OVERVIEW

- REpresentational State Transfer
- Stateless. No stored context on the server
- Uniform interface and commonality among different parts
- Key abstraction is the resource. Actions are taken on a resource.
  - For example, order is a resource. Actions are GET, POST, PATCH, DELETE, etc.

INFRASTRUCTURE AS CODE - OVERVIEW

- Code for the management of infrastructure (networks, servers, load balancers, volumes, orchestrators, etc)
- Provides automation and control
- Allows versioning since code can go into a code repo
- Solves environment drift
- Can prevent/overwrite manual interventions. Enforces consistency.
- Allows code review of infrastructure. Imagine the benefits to security conscious organizations (which should be all organizations).
- Allows fast build-out of identical environments. Also allows you to test environment changes before changing production.
- Several technologies exist such as TerraForm, Cloudformation, Ansible, Chef, Puppet, and many more.

INFRASTRUCTURE AS CODE - OVERVIEW

- Provisioning is different from configuration management (CM)
- With CM, you have static, long-lived resources that you're applying changes to in a controlled way. Examples of this are Chef and Puppet.
  - For example, upgrading a Chef managed server from one version of nginx to a newer one. The Chef recipe will run a package manager to upgrade it.
- With provisioning, resources are disposable and short-lived. Changing a resource often involves replacing it.
  - For example, with Terraform, to upgrade nginx, you'd just create a newer instance with the correct version, and throw away the old one.

SERVICE DISCOVERY - OVERVIEW

- With microservices, things are distributed. Services need to communicate with other services. But with the transient nature of microservices, how does an application know where another service is to communicate with it?
- Services could be available at a random IP, random port, random server. There's no way to know at runtime.
- Kubernetes solves this by automatically managing DNS for you so that you can work with named services and know that your traffic will be routed to the appropriate service. For example, a service named "my-service" can be accessed through that name from another service. The pods behind the service are automatically added and removed as necessary.
- We'll see more of this later.