Topic 6: Microservices

Dr. Daniel Yellin

Overview

1. Monolithic applications
2. Microservice concepts
3. Containers and microservices **by example** (Docker Compose)
4. Docker volumes
5. Docker data management
6. Appendix: The ongoing development debate

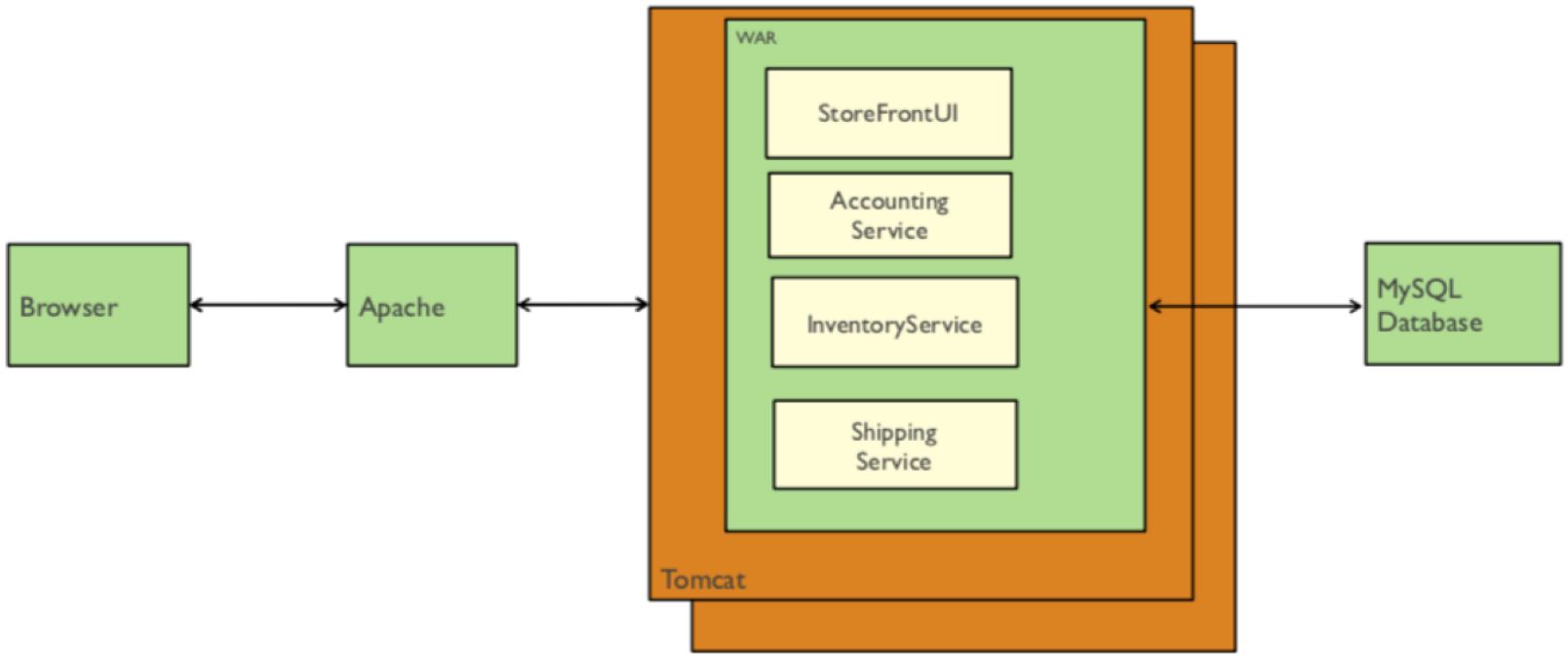
Monolithic Applications

To understand microservices, its best to first look at monolithic apps

Monolithic applications

- Monolithic applications, still in wide use today, are *in contrast to* microservices based applications
- By understanding monolithic applications, one can better understand why today more applications are developed using a microservices architectural pattern

Monolith Application

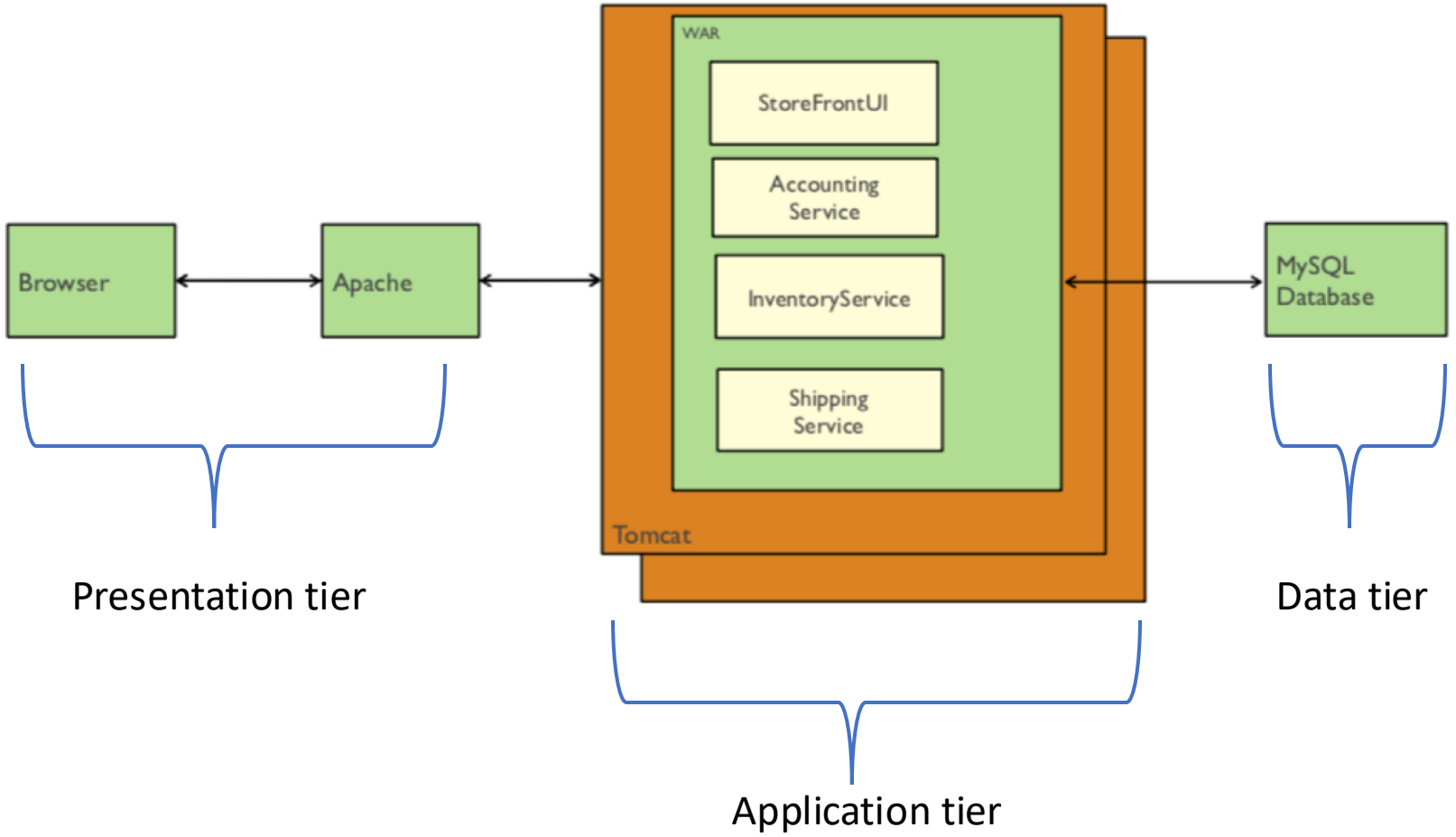


Characteristics of monolithic applications

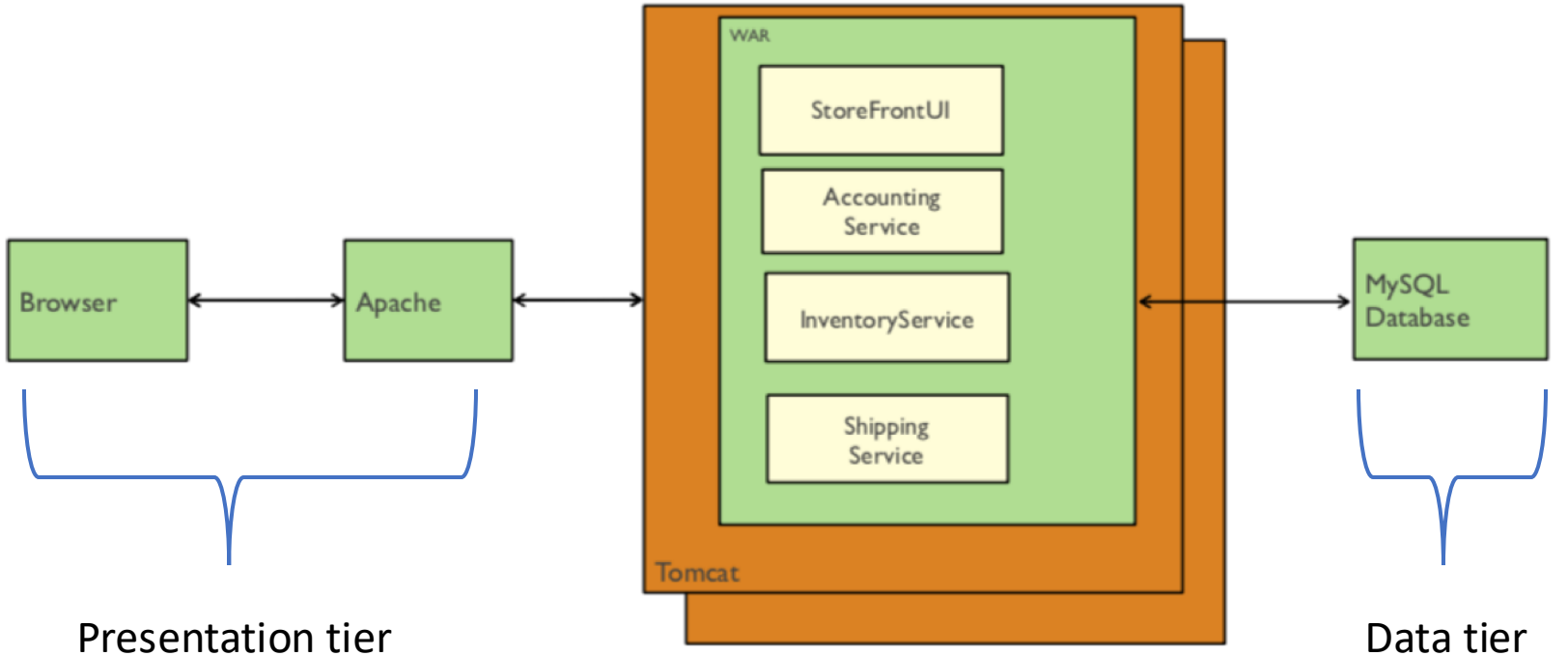
- **Single Codebase**: The application's entire codebase is centralized, allowing for collaborative development within a single, shared environment. Usually a single database for the entire app.
- **Unified deployment**: All the application code is deployed together.
- **Shared Memory Space**: The components of a monolithic application run in the same memory space and often on a single infrastructure (server or instance).
- **Cross-Component Communication**: functions can directly call each other without the need for inter-process communication.
- **Tightly Coupled**: A monolith's internal components are intricately interconnected. Modifications in one area can potentially have cascading effects across the entire application.

Monolith Application: 3 tier architecture

Usually embraces a 3-tier architectural style



Monolith Application: 3 tier architecture



Presentation tier
Responsible for the user interface.

Application tier
Responsible for business logic

Data tier
Manages the application's data

What are the benefits of a monolithic application?

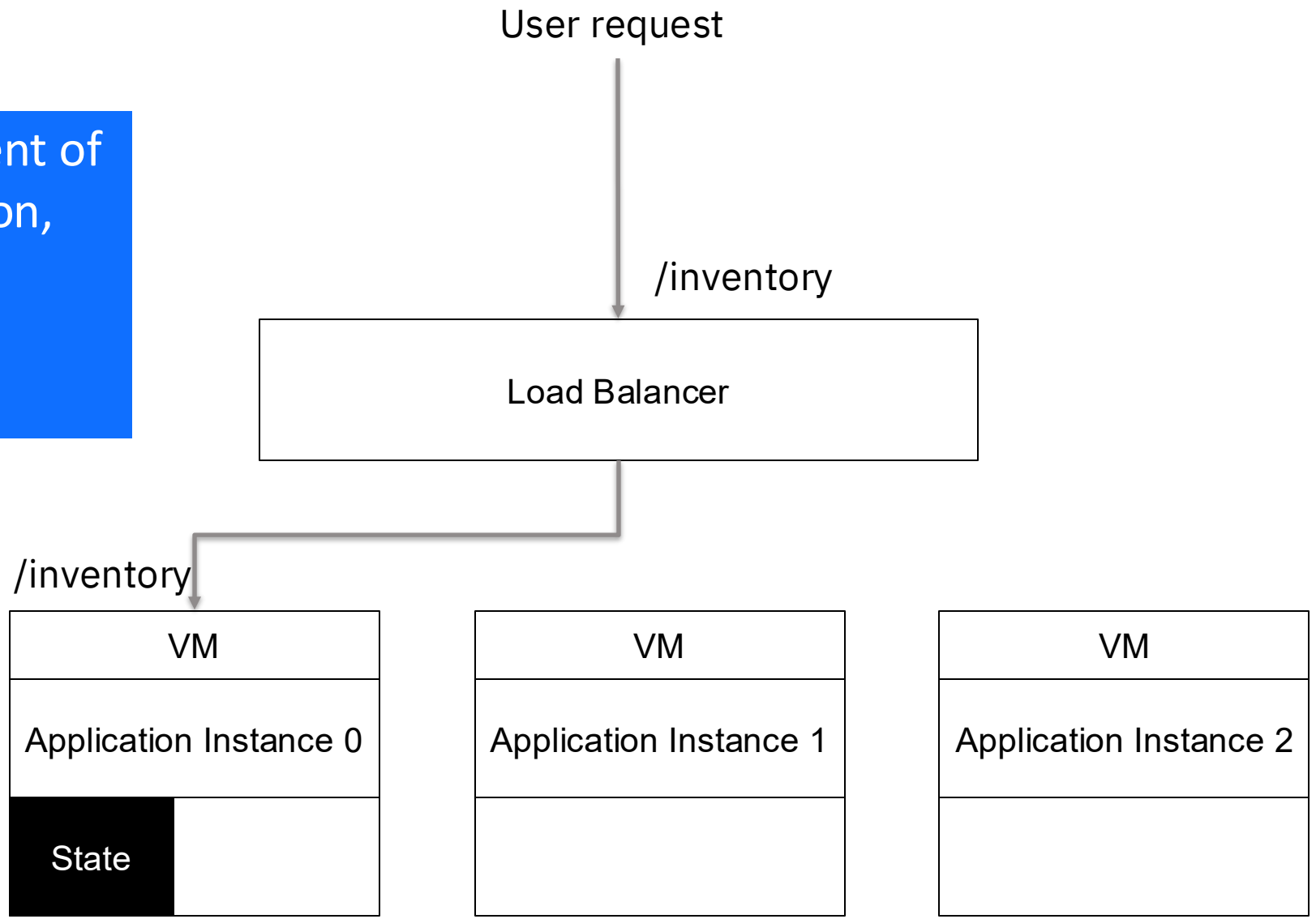
What are the benefits of a monolithic application?

For some applications, especially smaller ones, monolithic applications offer:

- **Simplicity**: simple to develop and test and deploy as it is based upon one code base. It has fewer components to manage and fix. It is mostly self-contained, not relying on external services.
- **Performance**: Component interaction is straightforward and fast because it (often) occurs within the same memory space.
- **Straightforward deployment**: deploy the entire entire app at once.
- **Consistency**: A monolithic application can provide a consistent look-and-feel, since all modules follow a unified development approach.

Monolith Application

To scale the deployment of a monolithic application, you would deploy the entire application on multiple VMs.



What are the drawbacks of a monolithic application?

- **Development Complexity:** The codebase can become large and complex, making development and maintenance more challenging. Tight coupling makes it hard to modify functions or to adapt new technologies.
- **Fragility:** One change can require the entire application to be rebuilt, retested, and redeployed, resulting in long delays in releasing new functionality.
- **Dependability:** Failure in a single component can render the entire application unavailable.
- **Scalability:** If one component needs more resources, the entire applications needs to be scaled up.

3 types of application cloud “development”

- Lift and shift
- Refactor
- Cloud native

Life and Shift

"*Lift and shift*" (also called *rehosting*), is a cloud migration strategy:

- It moves an application that runs on-premise to the Cloud, with minimal or no changes to its architecture.
- Often it involves taking apps that run in VMs on-prem and running them in VMs on the Cloud.
- This is a fast and straightforward approach. However, it has drawbacks:
  - Does not make use of Cloud features (such as auto-scaling)
  - May not perform well on the Cloud
  - Unaddressed application vulnerabilities become more of a risk on the Cloud
  - Can be more expensive long term

Cloud native

Cloud native applications are those that are built to run on the Cloud. They have the following characteristics:

- They use a microservice architecture.
- Each microservice runs in its own container.
- DevOps and CI/CD: they use automated pipelines for continuous integration and continuous delivery.
- Build to be resilient and fault-tolerant.
- Can scale up and down dynamically.

We will discuss all of these aspect in this and future lectures.

Refactored apps

Applications are sometime *refactored* to run on the cloud. It is a compromise between lift and shift and cloud native.

- Because it would be too costly (too long to develop), the application is not rewritten to be Cloud native.
- On the other hand, parts of the application are rewritten to take advantage of the Cloud benefits.

Microservice concepts

An architectural style for building loosely coupled services making up an application

Motivation for microservices

Many applications today

- are large and complex,
- are made up of many different capabilities,
- rely on external services,
- are updated frequently.

Monolithic applications are not a good fit for these applications.

The main drivers behind microservices

1. Decentralization of development

- *Separation of concerns:* enable each team to develop a single capability (microservice) independently of others
  - Sometimes called the *"single responsibility principle"* which states that "a module to be responsible to one and only one actor".
  - Fosters faster development, changes limited to smaller code base.
  - Each service gets the full attention of its own team. This enables quick and efficient independent development and maintenance.
  - Freedom to choose the best technology for each service.

The main drivers behind microservices (cont)

2. Independent software lifecycle

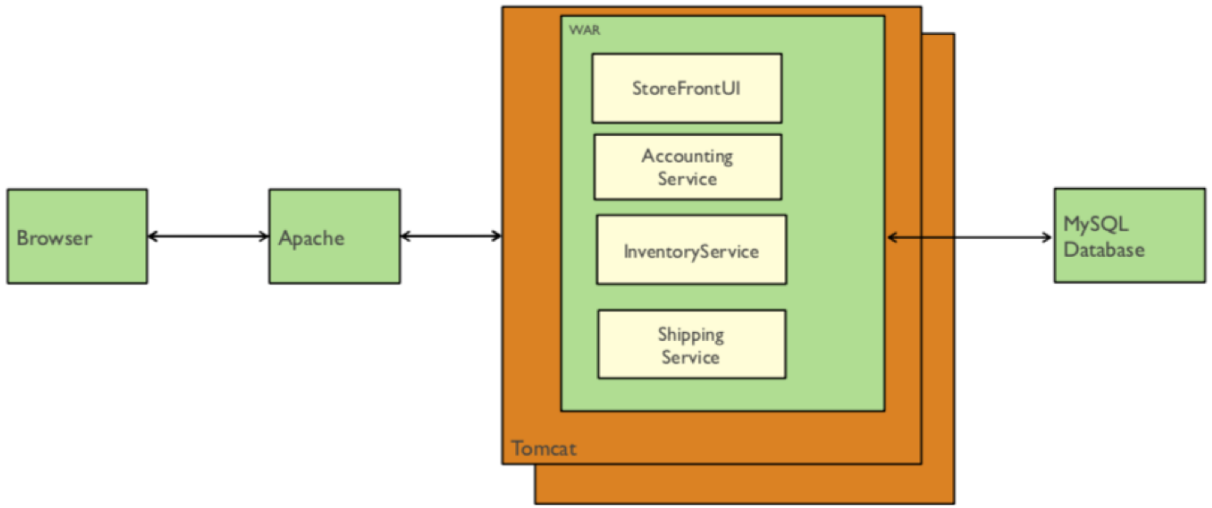
- *Deployment Independence*: each microservice can be deployed independently
  - Cloud requires ability to quickly deploy bug fixes, security patches and new features. Monolithic applications requires the entire app to be rebuilt in order to redeploy.
  - Faster to get code into production. Enhances team productivity as do not need to wait for others to deploy your code.
- *Reliability*: One service crashing does not affect other services.

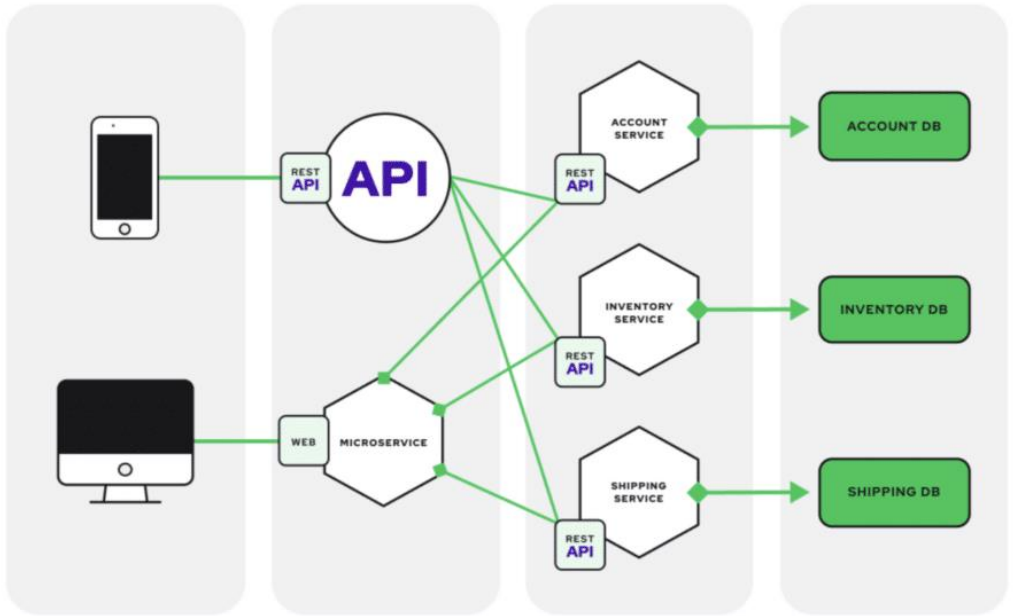
The main drivers behind microservices (cont)

3. Run-time scalability

- *Autonomous scaling*: the number of run-time instances of a microservice is, in general, independent of the number of run-time instances of other microservices
  - Better resource utilization, less manual run-time tuning , more responsive apps
  - Elasticity: the number of run-time instances of a microservice can be adjusted dynamically, based upon load

Monolith Application vs. Microservices App





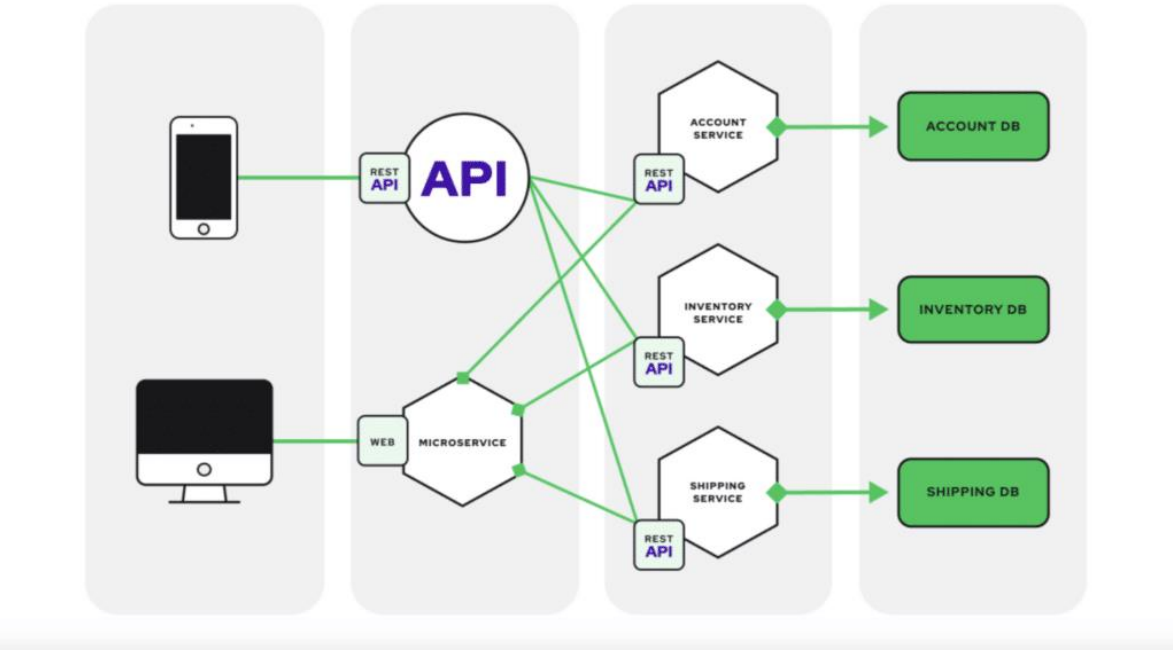
Source

Microservices App

Microservices is an *architectural style* where applications are broken into **smaller autonomous services that work together.**

Each microservice usually contains a single **business capability**. In the diagram these are the Account, Inventory and Shipping microservices.

The microservices work **independently**. Communication to microservices is through a well-defined interface.



Key features of microservices

Microservices are:

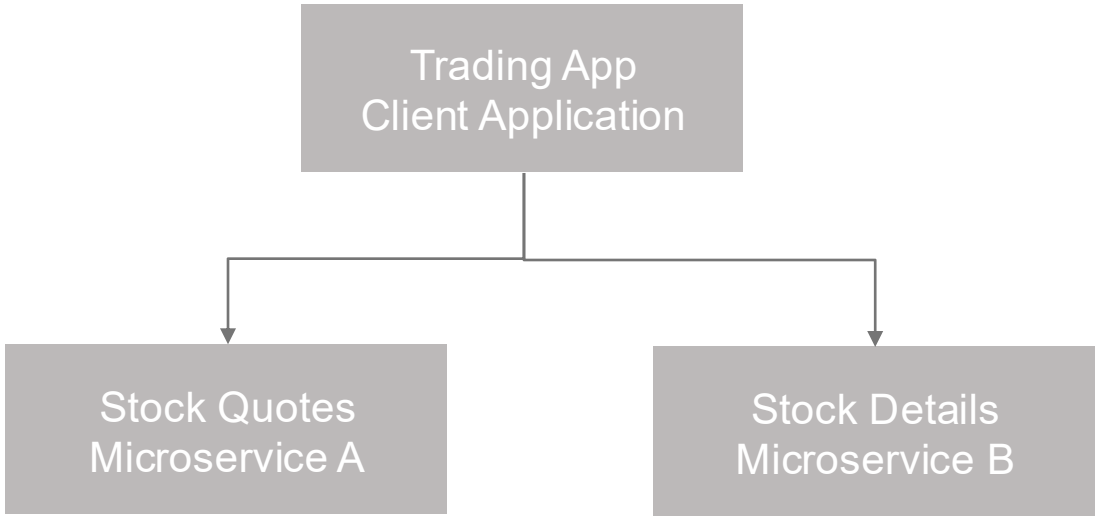
1. ***Self-contained*** with high *functional cohesion (*business capabilities)
   - Independently developed by a small team, easy to manage and maintain
   - Have high *static* and *synchronous dynamic* coupling
2. ***Loosely coupled*** to other microservices
   - Each microservice has its *own data store*
   - Reusable (at least in theory)
3. ***Independently deployable artifacts***
   - Independently scaled out
4. ***Technology agnostic*** (not restricted to any single programming language or framework– polyglot programming)

Microservices are composed via their APIs

An API is a set of clearly defined methods or components that enables different services and applications to communicate and share information with each other.

Example

- Microservice A offers real-time stock price
  - Operates independently and reusable
- Microservice B offers stock details
  - Operates independently and reusable
- Client Application – Trading App uses Microservice A and B



Communication with/between microservices

- Communication to services from **external users** is usually via REST, for reasons we have discussed.
- Communication to **between microservices** of an application will often not use REST but use either:
  - Remote Procedure Call (RPC) such as gRPC. Can be much more efficient. However, it requires tighter coupling between the microservices.
  - Message passing, such as Kafka. Asynchronous communication facilitates faster throughput, less latency, and improves high availability.

Running microservices in containers

Docker containers are a good fit for microservices

- *Decentralization of development - Separation of concerns:* Create a container for **each individual microservice**. Normally each microservice runs a **single process** and containers provide **task isolation**. One microservice per container.
  - Each microservice can choose its own language and libraries and frameworks.
  - Each microservice contains its own database for its data.
- *Independent software lifecycle.* Each container can be **deployed separately** in its own CI/CD process.
- *Run-time scalability and elasticity*: You can increase and decrease the number of container instances for each microservice at **run-time** independent of other containers.
- *Composition*: One can build applications by **integrating** microservices running in containers. Many tools facilitate this, including Docker Compose.

A 5th reason for encapsulation

We previously (topic 2) gave 4 reasons for encapsulation:

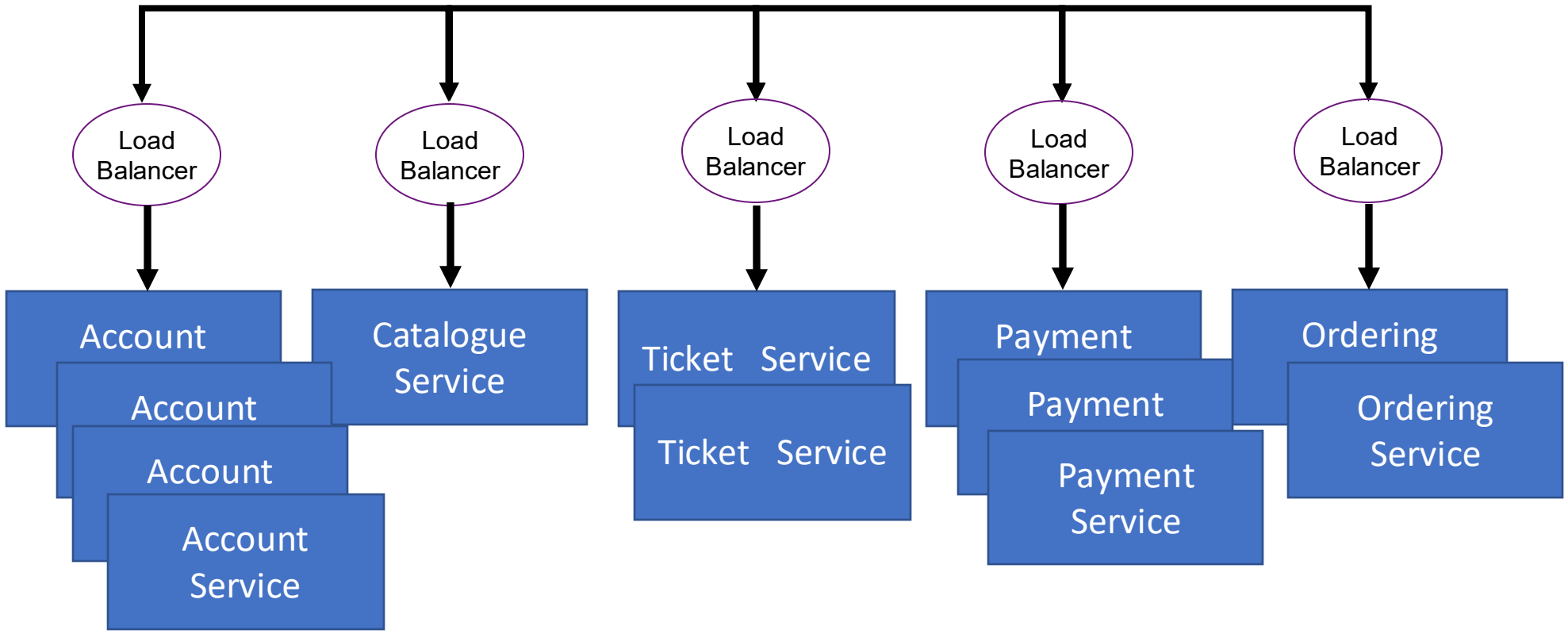
1. Independent development
2. Maintainability
3. Comprehensibility
4. Reusability

Microservices and the requirement for efficient scaling introduces a 5th reason:

5. ***Autonomous scalability*** - ability to scale each microservice independent of other microservices

Efficient independent horizontal scaling of microservices

Improved scale and resource usage by scaling each microservice independently and horizontally.



Each microservice will create the number of instances it needs to process its load.

Examples of companies adopting a microservices architecture

Netflix: One of the first to move to microservices. Here are some of the design principles they used:

- Create a Separate Datastore For Each Microservice
- Keep Code At a Similar Level Of Maturity
- Do a Separate Build For Each Microservice
- Deploy In Containers
- Treat Servers as Stateless

Examples of companies adopting a microservices architecture (2)

Capital One: 10 microservices best practices

1. The Single Responsibility Principle
2. Have a separate data store(s) for your microservice
3. Use asynchronous communication to achieve loose coupling
4. Fail fast by using a circuit breaker to achieve fault tolerance
5. Proxy your microservice requests through an API Gateway
6. Ensure your API changes are backwards compatible
7. Version your microservices for breaking changes
8. Have dedicated infrastructure hosting your microservice
9. Create a separate release train
10. Create Organizational Efficiencies

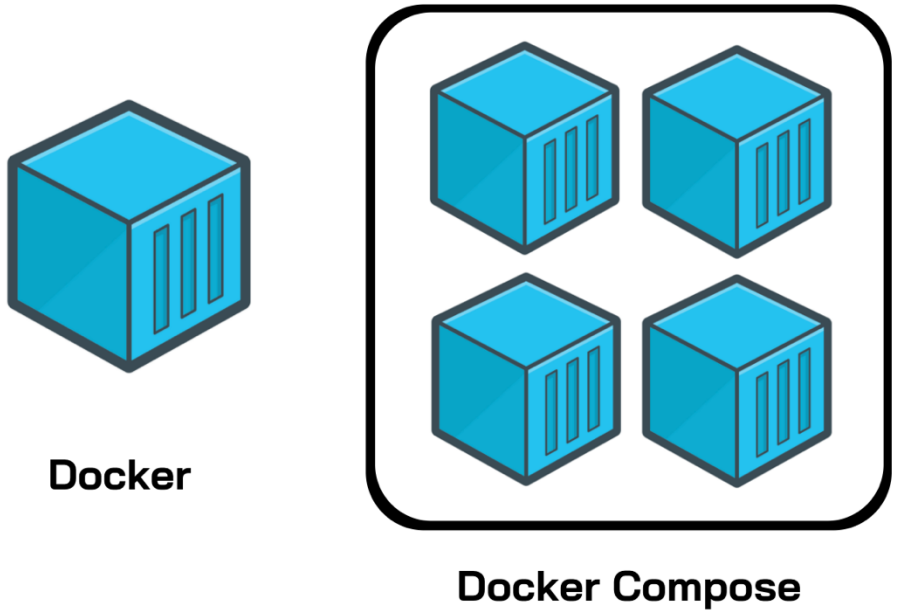
See also: 7 Microservices Benefits and How They Impact Development

Microservices and containers by example

The code for this example can be found in
https://github.com/dannyyellin/REST-toys-example

Docker and Docker Compose

- So far we have been focussed on deploying a single container.
- We used Dockerfiles to *declaratively* describe how to build a single Docker container.
- Most applications require more than one container - sometimes tens or hundreds of containers.
- Docker Compose provides a declarative mechanism for defining all the containers making up your application, how they communicate, and how they store/share data.



Source

Docker containers and microservices

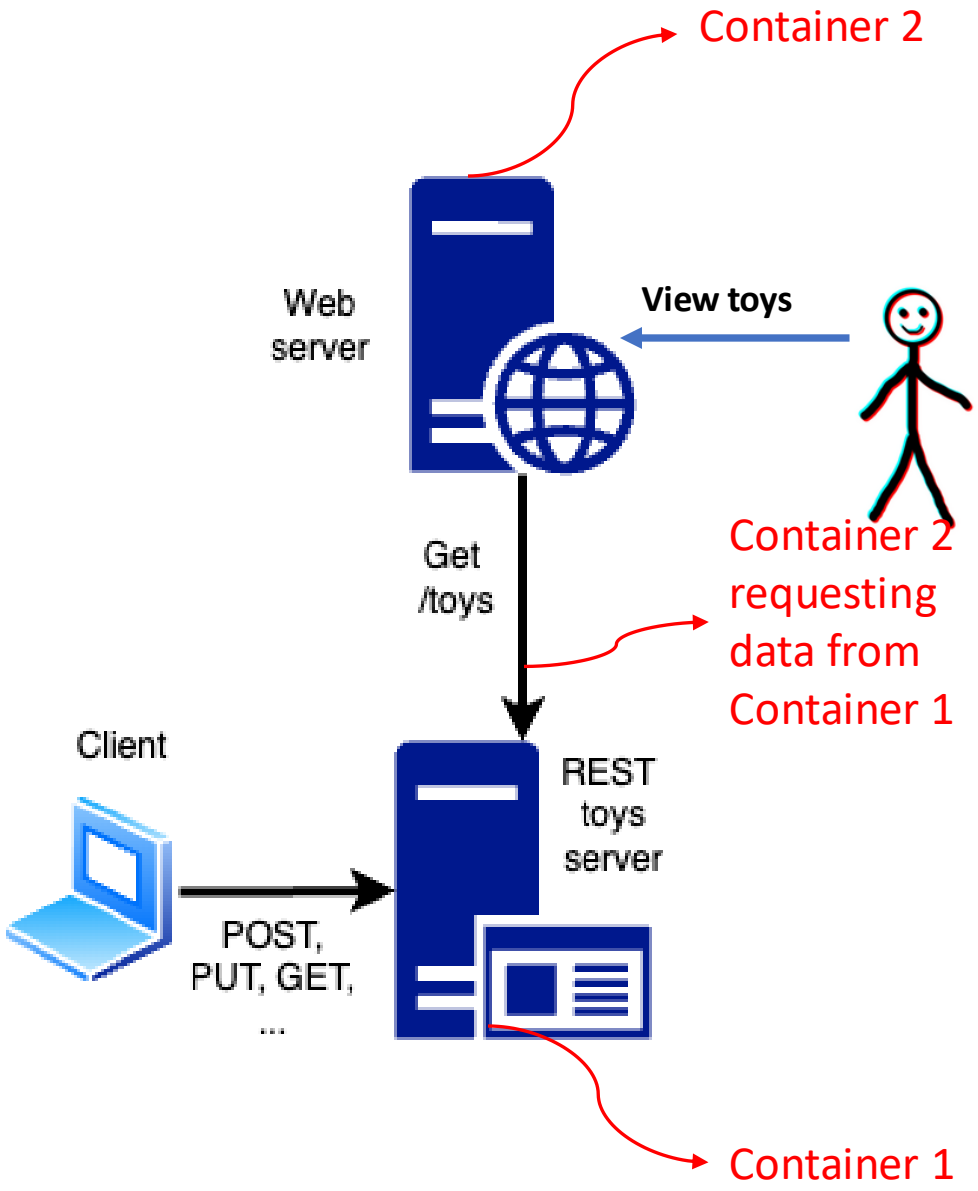
How would you:

- Manage a set of containers, each one implementing one microservice, that together make up an application ?
  - How would you ensure that they all run together when you start up the application?
  - How would you specify the order in which they must be started when the order is important?
  - How would you make it easy for them to communicate with one another, without having to know the IP address of each one (which might change each time they are started up)?
  - How do you ensure that they all use a specific microservice instance (e.g., DB)?
  - How would you let them share data that persists beyond the lifecycle of any specific container instance?

docker-compose

Docker-compose is a Docker tool that addresses these issues.

We will illustrate it by extending our example toys service in a simple way. We will add a web server frontend to display the toys that are currently in the toy sver inventory.

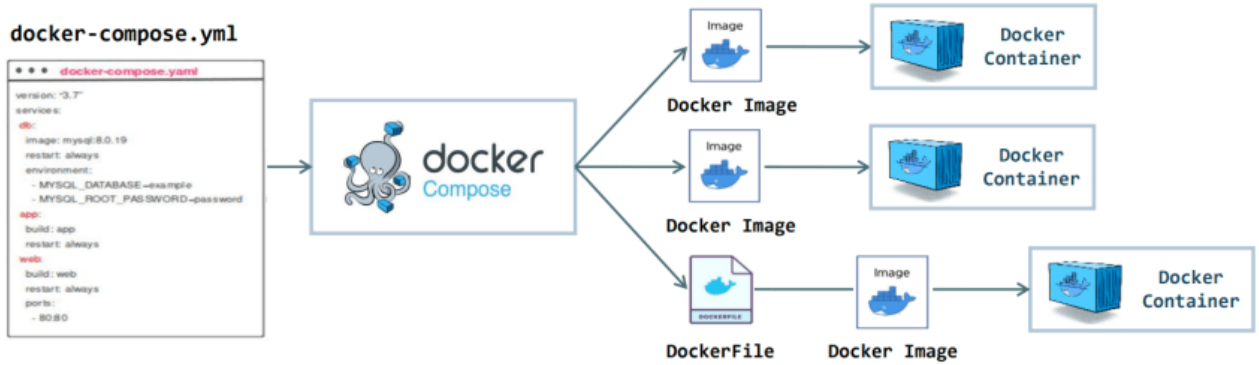


You must download Docker Compose to get started

The main docker compose concepts

- Service (≡ container)
  - Code running in a container, defined by an image or a Dockerfile
- Network
  - How services communicate, and who can communcate with whom
- Volumes
  - The persistent storage associated with a single container, or shared with multiple containers

Docker Networking & Volumes are actually independent of Docker compose, and can/are often used independ of it.



- docker-compose.yml
  - The declarative specification describing the services, networks, and volumes of the application.

The directory structure of our application

Our directory (on the host) structure will look like this:

webserver *(directory)*

- **app** *(subdirectory)*
  - **toys-robust.py**
  - **Dockerfile**
- **webSvr** *(subdirectory)*
  - **disply-toys.html**
- **docker-compose.yml**

This directory contains code for our toy-svr and the Dockerfile for this microservice

This directory contains code for a our web server microservice.

This is the yaml file containing our Docker Compose instructions.

The HTML (`disply-toys.htlm`) that the NGINX server will use

```
<!DOCTYPE html>
<html>
<head>
    <title>Toy List</title>
    <script>
        function fetchToys() {
            fetch('http://localhost:8001/toys')
                .then(response => response.json())
                .then(data => {
                    const toyList = document.getElementById('toyList');
                    toyList.innerHTML = ''; // Clear the list before appending

                    data.forEach(toy => {
                        const li = document.createElement('li');
                        li.innerHTML = `
                            Name: ${toy.name} (${toy.age}+) - $${toy.price}<br>
                            Description: ${toy.descr}<br>
                            ID: ${toy.id}<br>
                        `;
                        toyList.appendChild(li);
                    });
                })
                .catch(error => {
                    console.error('Error fetching toys:', error);
                    // Handle error, e.g., display an error message
                });
        }

        // Initial fetch on page load
        window.onload = fetchToys;
    </script>
</head>
<body>
    <h1>Toy List</h1>
    <button onclick="fetchToys()">Refresh</button>
    <ul id="toyList"></ul>
</body>
</html>
```

Performs a GET on toys

docker-compose.yml

```yaml
version: '3'  # version of compose format

services:
  toys-robust:
    build: app  # path is relative to current dire
    volumes:
      - type: bind
        source: ./app #host directory
        target: /app # container directory
    ports:
      - "8001:80"  # host:container

  website:
    image: nginx:alpine
    volumes:
      - type: bind
        source: ./webSvr/disply-toys.html  # host
        target: /usr/share/nginx/html/index.html #
    ports:
      - "8080:80"  # host:container
    depends_on:
      - toys-robust  # website is not going to wor
```

- Docker Compose version
- **services** specifies each micro-service in the application. Each one will run in its own container.
- In our case, we have two services:
  - One we are calling "**toys-robust**" and one we are calling "**website**".
  - You can give them any names you want.

docker-compose.yml

```yaml
version: '3'  # version of compose format

services:
  toys-robust:
    build: app  # path is relative to current dire
    volumes:
      - type: bind
        source: ./app #host directory
        target: /app # container directory
    ports:
      - "8001:80"  # host:container

  website:
    image: nginx:alpine
    volumes:
      - type: bind
        source: ./webSvr/disply-toys.html  # host
        target: /usr/share/nginx/html/index.html #
    ports:
      - "8080:80"  # host:container
    depends_on:
      - toys-robust  # website is not going to wor
```

- The **build** cmd for the toys-robust service tells docker-compose that if an image for this service does not already exist, it can be created using the Dockerfile in the subdir **./app**
- The website service does not have a Dockerfile to build the image. Instead, the **image** cmd tells docker-compose to use the image **nginx:alpine**, which it will download from Dockerhub if it does not exists on your host.
  - NGINX by default will display the html in the directory usr/share/nginx/html/index.html

docker-compose.yml

```yaml
version: '3'  # version of compose format

services:
  toys-robust:
    build: app  # path is relative to current dir
    volumes:
      - type: bind
        source: ./app #host directory
        target: /app # container directory
    ports:
      - "8001:80"  # host:container

  website:
    image: nginx:alpine
    volumes:
      - type: bind
        source: ./webSvr/disply-toys.html  # host
        target: /usr/share/nginx/html/index.html
    ports:
      - "8080:80"  # host:container
    depends_on:
      - toys-robust  # website is not going to wo
```

The **volumes** cmd sets up storage (files, directories) for a container.

- For both services, we declare the *type* to be a *bind mount.* This means that the files are shared *between the host and container*, and changes by one is visible to the other
- For the toys-robust service, the sub-directory /app on the host will be shared with the /app directory in the container.
- For the website service, the file /webSvr/disply-toys.html on the host will be shared with the /var/www/html sub-directory in the container.
- This avoid the need to copy files. But should only be used for dev, not production.

docker-compose.yml

```yaml
version: '3'  # version of compose format

services:
  toys-robust:
    build: app  # path is relative to current dire
    volumes:
      - type: bind
        source: ./app #host directory
        target: /app # container directory
    ports:
      - "8001:80"  # host:container

  website:
    image: nginx:alpine
    volumes:
      - type: bind
        source: ./webSvr/disply-toys.html  # host
        target: /usr/share/nginx/html/index.html #
    ports:
      - "8080:80"  # host:container
    depends_on:
      - toys-robust  # website is not going to wor
```

- The **ports** cmd is the same as we have already seen in the Docker cmds.
  - In this case, it says that the toys-robust-service will be accessible on port **8001** in the host. From the container's point of view, it will be listening on port **80**.
  - Similarly, the web-service will be available on port **8080** of the host, which maps to port **80** of the container.
- **depends on:** tells docker-compose that the **website** service requires the **toys-robust service** to run, and that the toys-robust service must therefore be started before the website service.

docker-compose build & docker-compose up

1. We go to the parent directory (containing the docker-compose.yml) file and issue the command:

   `docker compose build`

This command will build the images or pull them from a repository, as specified in the docker-compose.yml.

2. Once docker-compose successfully completes, we issue the command:

   `docker compose run`

This command will create the containers from the images, configure them, and start them

3. The command

   `docker compose up`

Is equivalent to:

`docker compose build`
`docker compose run`

After stopping the containers, the command

`docker compose down`

removes the containers.

docker-compose: getting up-to-date images

Warning: if you already ran docker compose run (or up) and changed some source files, docker-compose may still use the existing images. This can be due to:

- You pulled and image from a remote repository and the image changed in the remote repository. Docker finds the (old) image in your local repository or cache. **`docker compose up`** does not check for changes in the remote repository.
- You built an image from a Dockerfile and changed a Dockerfile dependency. Docker finds the (old) image in your local repository or cache. docker-compose up does not check for changes and uses the old image.

Using

```
docker compose up
--force-recreate
```

should solve these problems. Or remove the images before issuing docker compose up.

docker-compose down --rmi all -v

The **-- rmi all** option not only removes the containers, it also removes all images that have *not* been tagged. The **–v** removes all volumes. Use with CAUTION. See docker compose options and here for additional advice

docker-compose build & docker-compose up (cont)

4. After docker compose us, we have two (actually 3, since we are using Mongo for the toys-robust app) containers up and running. We verify with the following cmd:

   `docker compose ps`

5. On **port 8001**, we issue HTTP requests to the toys-robust service to add toys to the service. (Use either *Postman* or *curl*).

```
danielyellin@Daniels-MacBook-Air webServer % docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED              STATUS              PORTS                     NAMES
bb750c600693   nginx:alpine           "/docker-entrypoint.…"   About a minute ago   Up About a minute   0.0.0.0:8080->80/tcp      webserver-website-1
56e05dfd96fb   webserver-toys-robust  "flask run --host=0.…"   About a minute ago   Up About a minute   0.0.0.0:8001->80/tcp      webserver-toys-robust-1
8dd758307676   mongo                  "docker-entrypoint.s…"   About a minute ago   Up About a minute   0.0.0.0:27017->27017/tcp  webserver-mongo-1
```

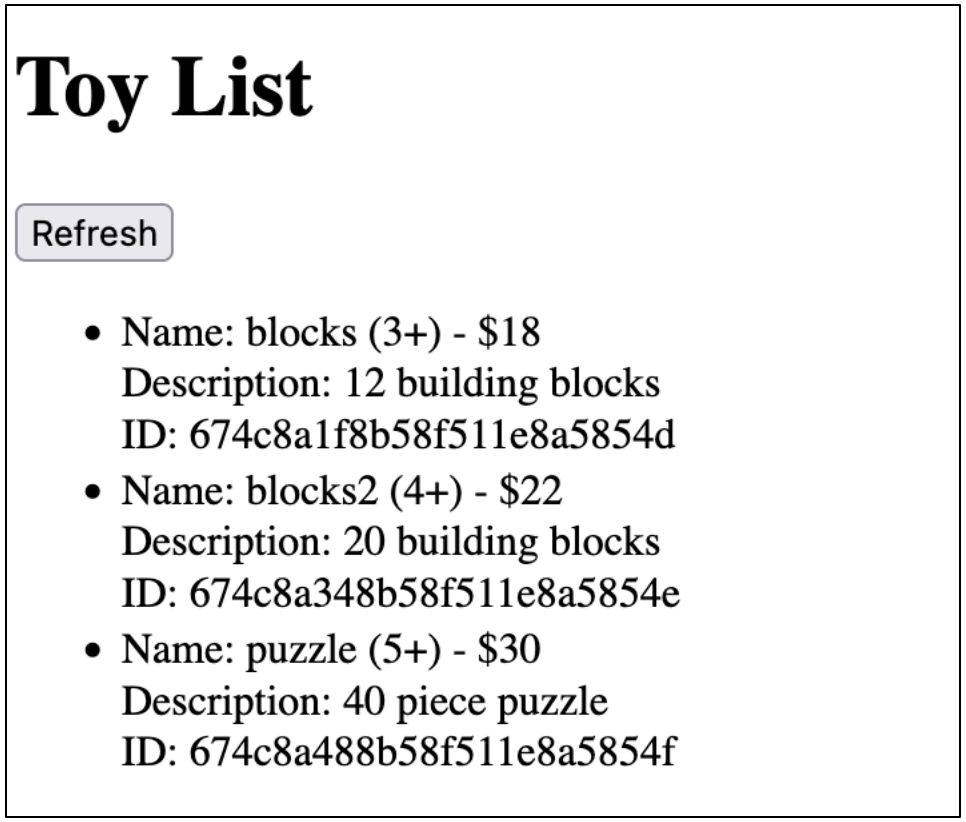
Note the name Docker Compose gives the different containers:
<compose-directory>-<service-name>-<num>

docker-compose build & docker-compose up (cont)

6. We open our browser and navigate to **localhost:8080** (or **0.0.0.0:8080**). That is the host port that NGINX listens on.

We see the rendering of the disply.html file.

- Experiment by adding/deleting words from the toys-robust service and use the refresh buttom to see the updates.



My toys-robust app uses Mongo

For my toys-robust application to talk to mongo, it needs to know the ip address of the Mongo container. This ip address is given to the container each time it starts by Docker, and will, in general, be different each time.

So how do I address the mongo container from my toys-robust application?
client = pymongo.MongoClient('mongodb://???:27017/')

Since in docker-compose.yml I define Mongo to have the service name “mongo”, I can just use the symbolic name ”mongo” in place of its IP address. Docker compose takes care of the rest.
client = pymongo.MongoClient('mongodb://mongo:27017/')

```yaml
services:
  toys-robust:
    build: app  # path is relative to current dir
    volumes:
      - type: bind
        source: ./app #host directory
        target: /app # container directory
    ports:
      - "8001:80"  # host:container

  website:
    image: nginx:alpine
    volumes:
      - type: bind
        source: ./webSvr/disply-toys.html  # host
        target: /usr/share/nginx/html/index.html
    ports:
      - "8080:80"  # host:container
    depends_on:
      - toys-robust  # website is not going to wo

  mongo:
    image: mongo
    ports:
      - 27017:27017
    volumes:
      - mongo_data:/data/db
```

More precise description

'mongodb://mongo:27017/'

The mongodb:// protocol tells the application that it should use the MongoDB driver to communicate with the database.

This specifies the **port number** on which MongoDB is listening for connections. The default port for MongoDB is 27017.

Docker Compose automatically creates a **network** for your containers to communicate with each other. Services within the same Docker Compose network can communicate with each other by using the service names as hostnames. So, when you specify mongo in the connection string, Docker resolves it to the IP address of the container running the MongoDB service.

Mongo and docker compose

Start mongo first and then the application (since it assumes DB is up and running). A “depends_on: B” means B must be started before A.

Use latest mongo image on DockerHub

Run on default port for Mongo, 27017

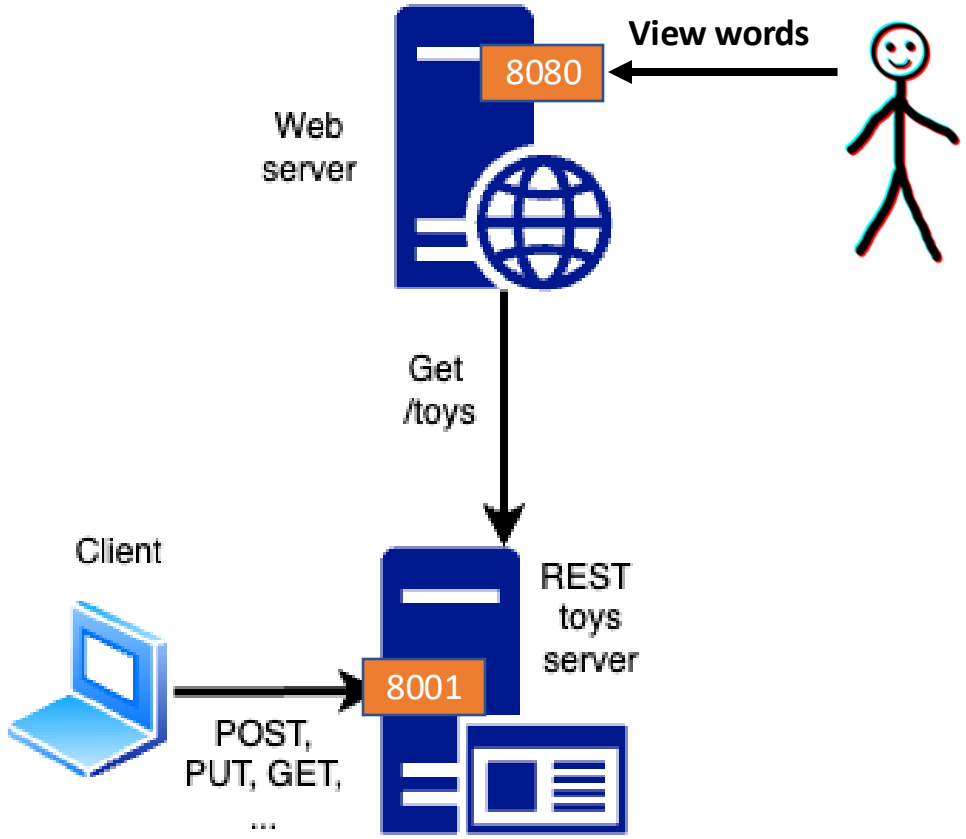
```yaml
services:
  toys-robust:
    build: app  # path is relative to current dir
    volumes:
      - type: bind
        source: ./app #host directory
        target: /app # container directory
    ports:
      - "8001:80"  # host:container

  website:
    image: nginx:alpine
    volumes:
      - type: bind
        source: ./webSvr/disply-toys.html  # host
        target: /usr/share/nginx/html/index.html
    ports:
      - "8080:80"  # host:container
    depends_on:
      - toys-robust  # website is not going to wo

  mongo:
    image: mongo
    ports:
      - 27017:27017
    volumes:
      - mongo_data:/data/db
```

Micro-services architecture

- docker-compose has instantiated our application architecture given previously
  - We have two services up and running, one being the REST toys service and one being the website service
  - The ports that the apps are listening are defined in the yml
  - Service names can be used as symbolic names for the IP addresses
  - Volumes are set up so that the source code on the host is shared with the container.



Docker volumes (bind mounts)

Persisting container data

How to persist container data

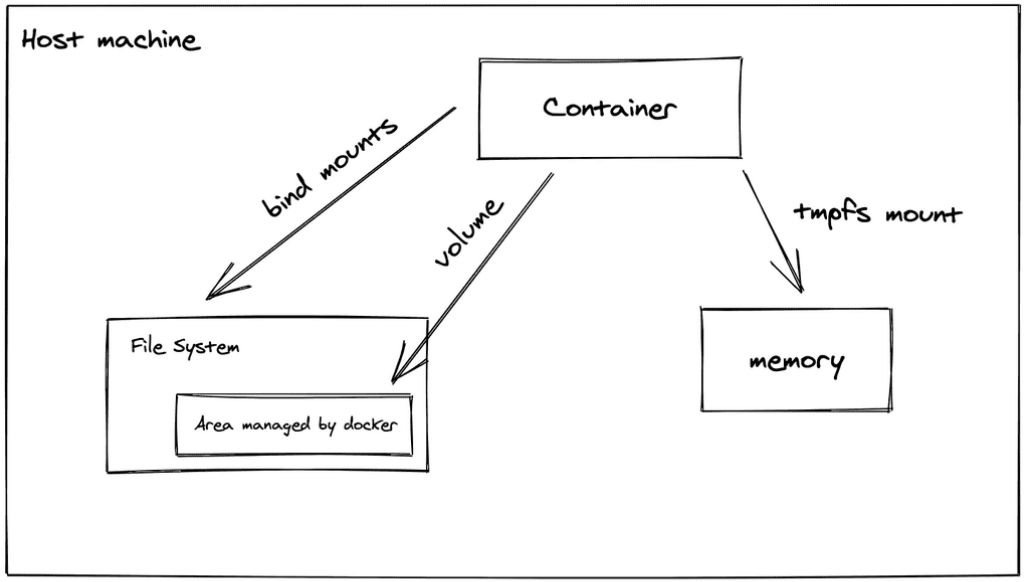
1. Bind mounts
   - Specify a host directory or file and a container directory or file.
   - The host directory or file is *shared* with the container directoryor file.
   - Changes on the host are persisted and seen by container and vice versa.
2. Docker volumes
   - Storage managed by Docker.
   - Can be given a name or can be anonymous.
   - Can be *shared across containers* but is not visible outside of Docker
   - Independent lifecyle – data persists in the volume even if the container is removed. This allows the container to save data even if it is killed, or crashes.
3. tmpfs mounts
   - To store non-persistent data. We will not focus on this.

Source

How to persist container data

1. Bind mounts
   Syntax: /host/path:/container/path
2. Docker volumes
   Syntax: volume_name:/container/path

   We will discuss Docker volumes more in the next lecture



An alternative syntax for bound mounts:

```
- type: bind
      source: /host/path
      target: /container/path
```

Source

A cool thing about using bind-mounts

While the micro-service containers are running:

- Initially, the html that NGINX serves looks like the screenshot given here.
- I then go into the html on my host and change the heading from “Toy List” to “Toy List Inventory”.
- Then refresh the browswer
- What happens?

A cool thing about using bind-mounts

The html is updated (without restarting NGINX).

NOTE: if it doen't work for you, it may be that your browser is caching the old HTML and you may have to clear the cache. Sometimes the file system can be caching the old HTML. (It may even be that NGINX is caching the old HTML, and you may need to adjust NGINX configuration).

This is a powerful tool for development and debugging – you can change your source files and do not have to restart the containers.

Using bind-mount, no need to copy toys-robust svc

In our docker-compose.yml:

```
toys-robust:
  build: app
  volumes:
    - type: bind
      source: ./app
      target: /app
```

This means that we do not need to copy the toys-robust.py file into the container. It can access it because it is bind mounted.

In our Dockerfile for /app dir:

```
…

RUN pip install requests
ENV FLASK_APP=toys-robust.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_RUN_PORT=80

# Because the Docker Compose bind
mounts the host app directory, the
host file toys-robust is accessible
to the service.  We do not need to
copy it into the container
# COPY toys-robust.py .


EXPOSE 80

…
```

Microservices and data management

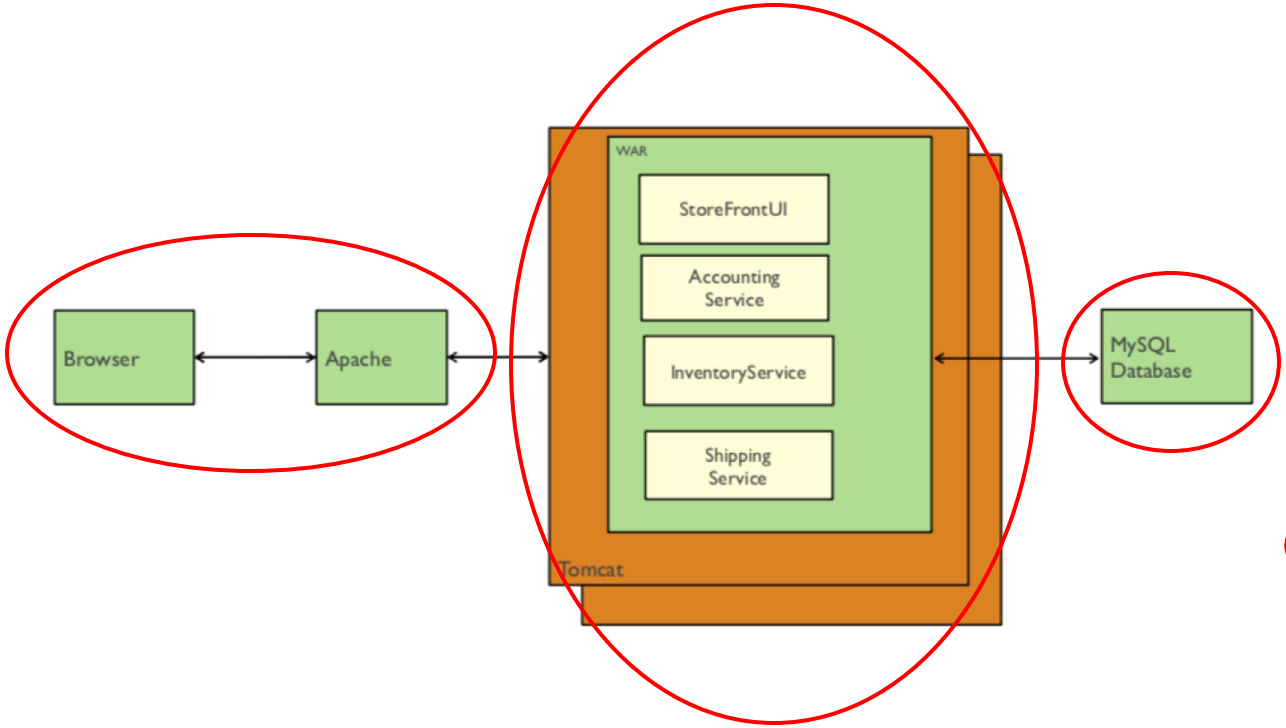
Design choices

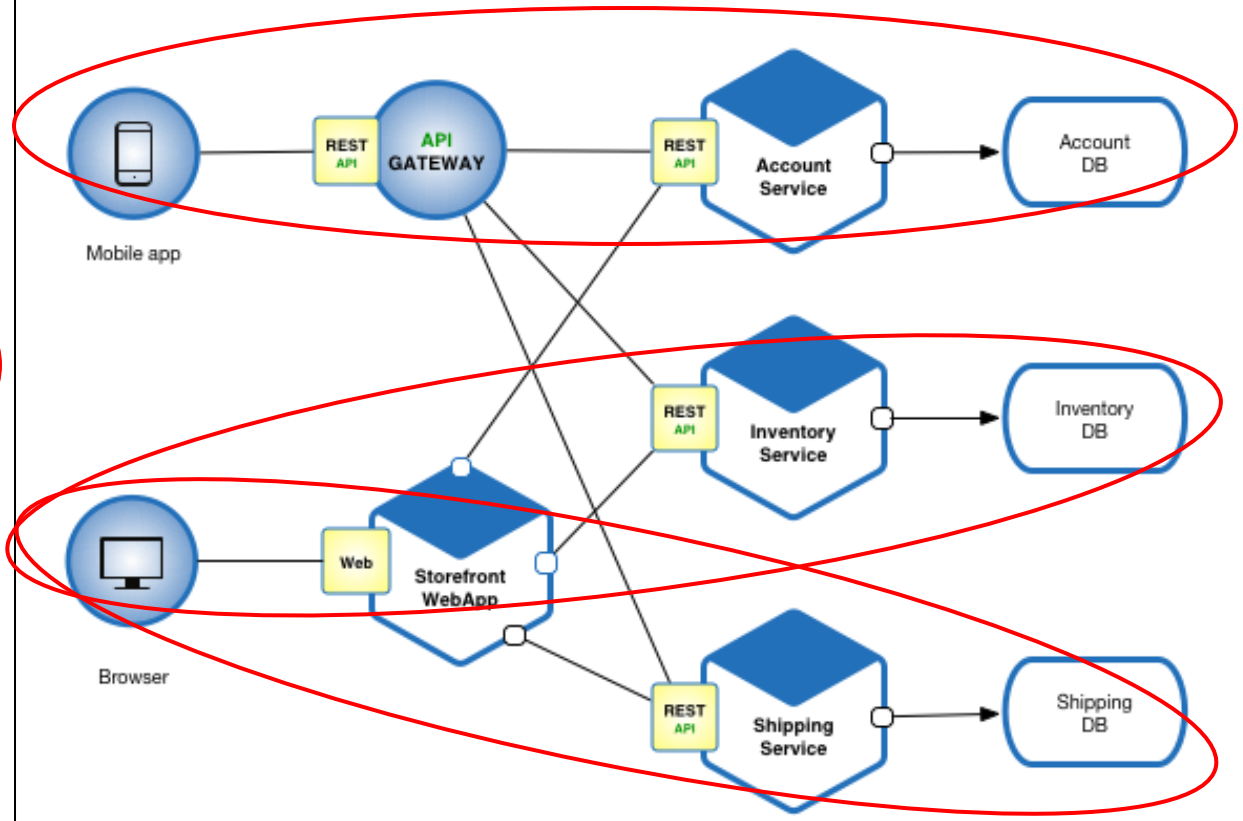
Designing microservices around business capabilities

The key principle is to design microservices around **business capabilities**

- Each microservice represents one business capability
  - For example, a shopping application may have catalogue, inventory, shopping cart, user profile, payment and shipping services.
  - Unlike a monolithic app, which is divided into tiers based upon technology (presentation, business logic, data tiers), a *single microservice will be responsible for all of its tiers*.
  - **Service scope and function**. Consider cohesion and size directly related to the single responsibility principle, where each service should be aligned with its responsibility, i.e., to do one thing well.

Technology tiers versus business capabilities



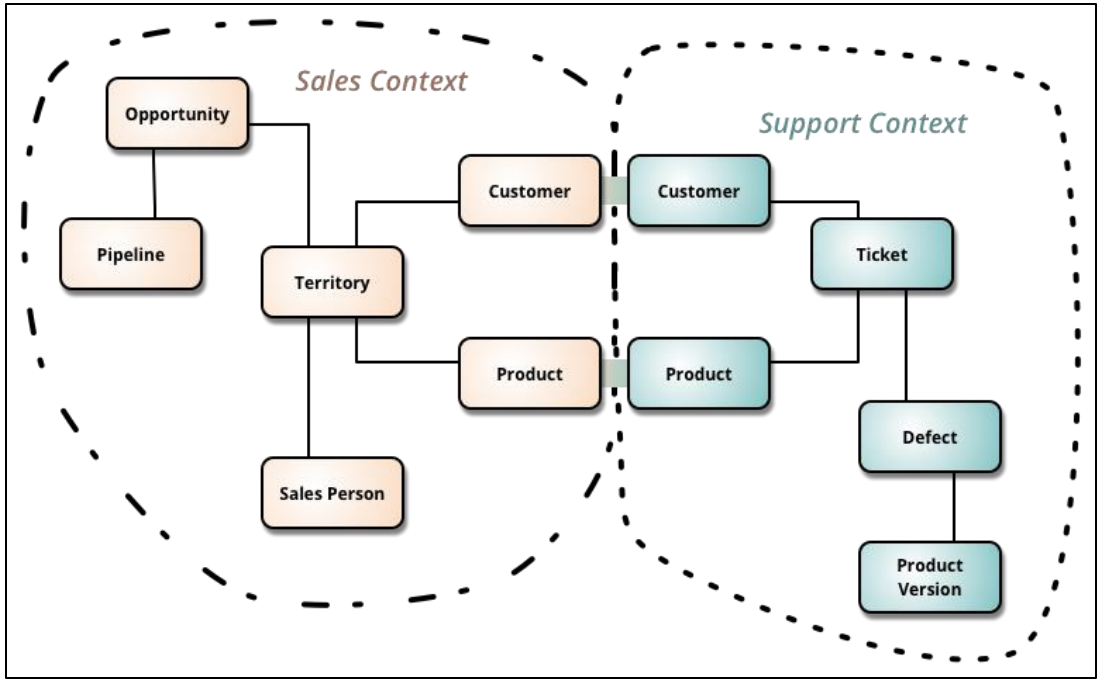


Decomposing the data model

Decomposing the data model helps to:

- understand the logical data the application is manipulating, and
- the relationships between the different data elements.

This facilitates an understanding on how to group data elements together into a single microservice.



Source

Other considerations on how to determine the scope of a single microservice

Code volatility. Consider splitting a component that changes more frequently than the rest to reduce the scope of testing and deployment.

Scalability and throughput. Consider this if some services need to scale more than others.

Fault tolerance. If one component crashes, this can impact other components.

Security. If we need to have different security concerns per component.

Data considerations for microservices

A basic principle for microservices is that every microservice manages its own data:

- Avoid tight coupling between services, where a change to the data schema must be coordinated across all microservices that use that data item.
- Allows each microservice to optimize data access for its own needs.

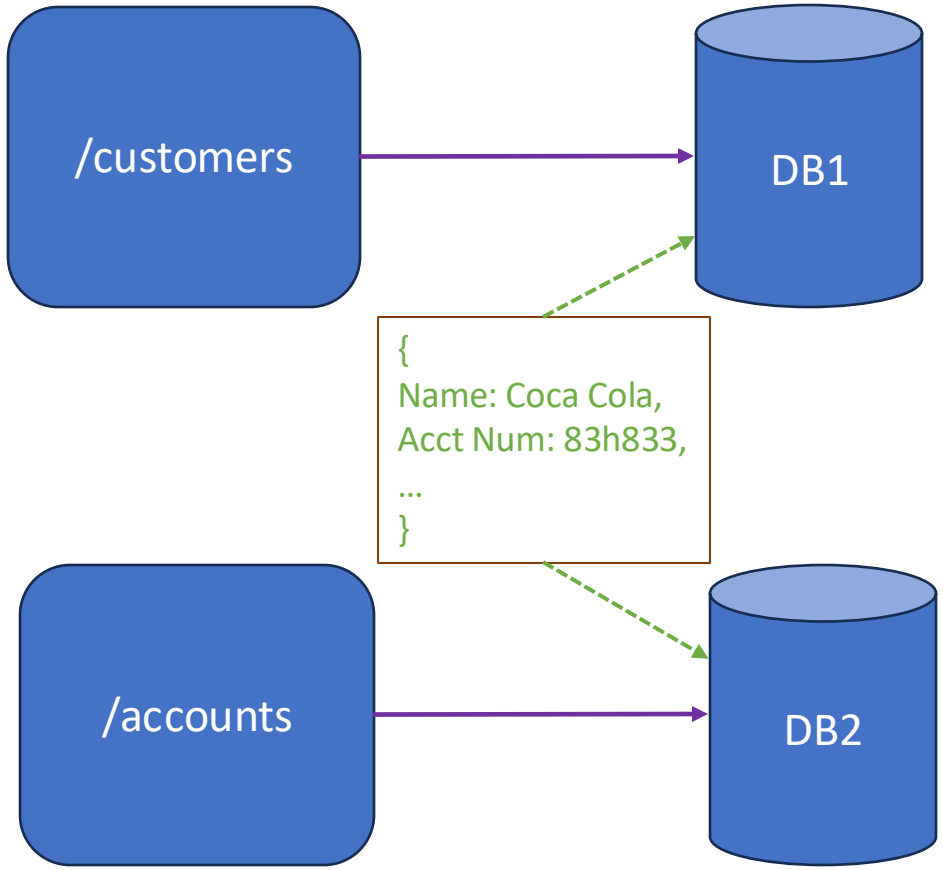
A challenge with this approach is how to deal with data that multiple services need to access?

Redundancy – have the same data item in multiple data stores managed by different microservices. However this can lead to inconsistencies.

- One approach is to use traditional DB synchronization techniques called *strong consistency* (ACID properties).
- Another approach to resolve inconsistencies: *eventual consistency.*

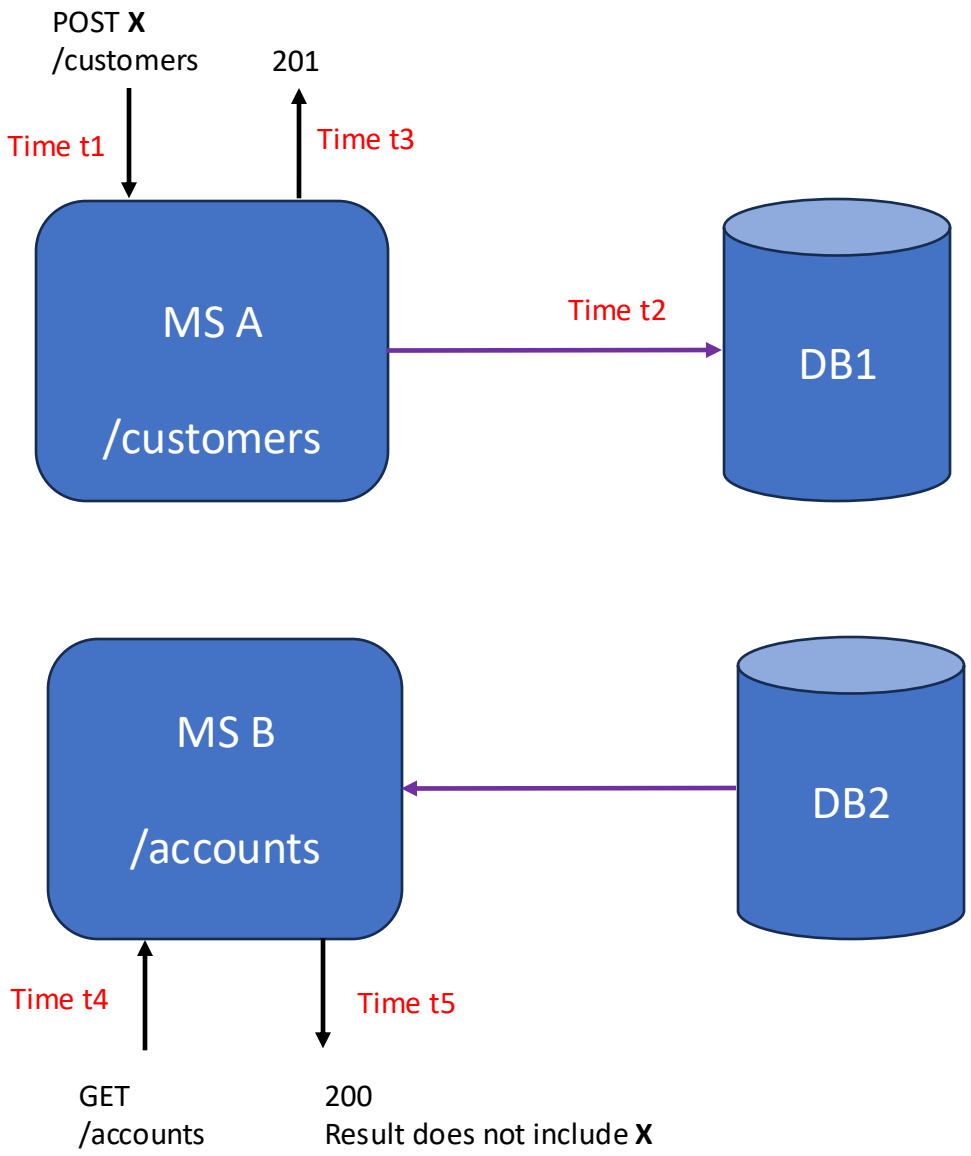
Eventual Consistency

- MS A has a resource called */customers*, stored in DB1.
- MS B has a resource called */accounts*, stored in DB2.
- Both of these resources contain the same information.



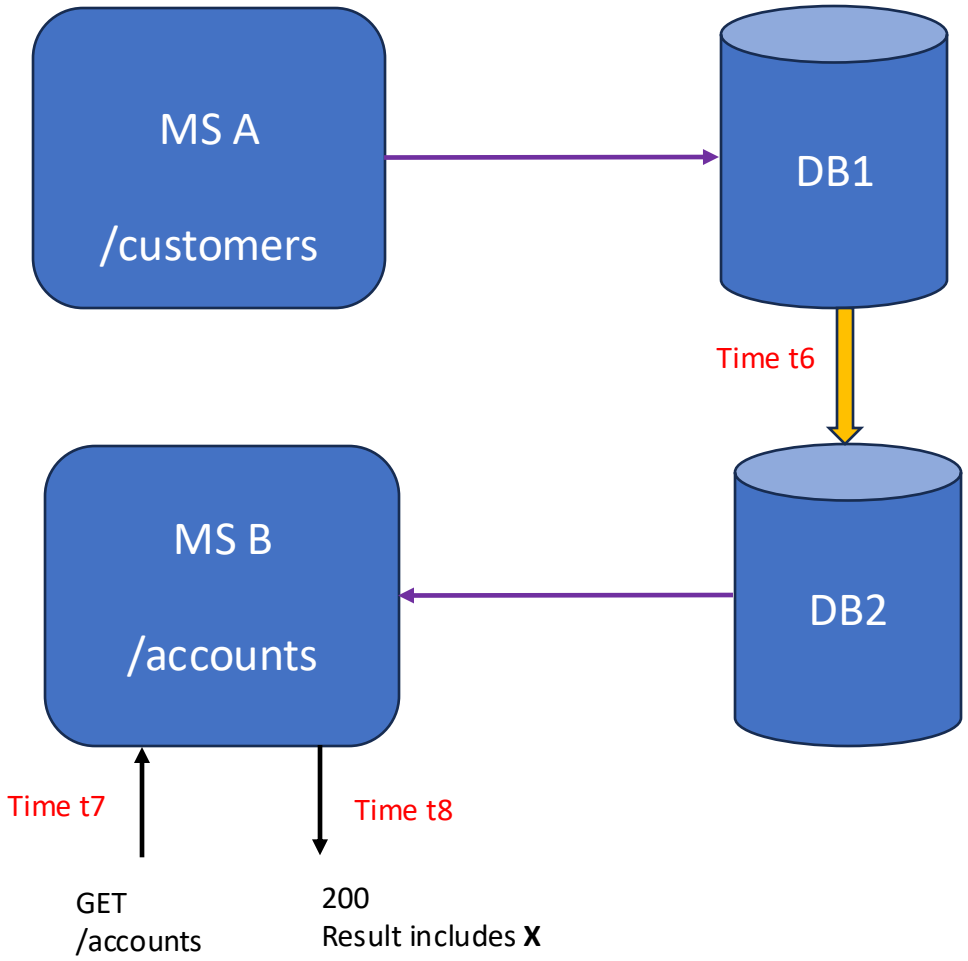
Eventual Consistency

- Time t1: An app P sends a POST X request to /customers.
- Time t2: A's DB is updated.
- Time t3: P gets back 201 response.
- Time t4: Another app PP sends GET request to /accounts
- Time t5: PP gets back 201 response. But the response does **not** include the customer X just added by P.



Eventual Consistency

- Time t6: DB1 sends updated data to DB2.
- Time t7: PP sends another GET request to /accounts
- Time t8: PP gets back 201 response. But this time the response **does** include the customer X added by P.



When should one use eventual consistency?

When should one use eventual consistency?

- **Eventual Consistency:** Well-suited for applications where real-time consistency is not vital and where system availability and scalability are more important, such as social media platforms, content distribution networks, and collaborative systems.
- **Strong Consistency:** Best suited for scenarios where data integrity and consistency are critical, such as financial systems, e-commerce platforms, and critical business applications.

Source

Summary

Design choices

Key microservices principles

- Independent development and deployment.
  - A microservice implements one capability, and is developed by its own team.
  - Can roll out each microservice independently. Requires CI/CD pipeline.
- Autonomous scaling.
  - Different microservices can have different number of running instances to meet its specific load requirements.
- Data separation.
  - Each microservice has its own database.
- Language and framework agnostic.
  - Each microservice can be written in the most appropriate language for its task.

Advantages of microservices

- Easier to develop and manage microservices than large monoliths. New developers can become productive quicker.
- Change in one microservice only requires to redeploy that microservice
- Can scale per microservice (as opposed to entire application)
- Each microservice can use a different technology based upon requirements for that microservice

Disadvantages of Microservices

- System complexity
  - Microservices introduce substantial operational overhead. Instead of monitoring, deploying, and maintaining one application, teams suddenly need to manage dozens or hundreds of microservices.
- Data consistency
  - Distributed transactions, eventual consistency, and partial failures. These complexities require different skills and patterns that many development teams aren't prepared for.
- Performance
  - Excess resource consumption as each microservice has its own containers, storage, logs, … Network latency slows down application as microservices communicate with one another.
- Higher Costs
  - The operational overhead of running numerous services (infrastructure, tooling, specialized DevOps talent) may result in higher cloud bills than a comparable monolith.
- Cognitive complexity.
  - As systems grow more distributed, understanding the complete flow of a request becomes increasingly difficult.

Finding the Middle Ground: Service-Based Architecture

While companies are still, in large, embracing microservice concepts, many are taking a more balanced approach.

References:

https://sysctl.id/microservices-to-monoliths-pendulum-swing/

https://medium.com/@sohail_saifi/the-end-of-microservices-why-companies-are-moving-back-to-modular-monoliths-7ccc9b0f6e32

https://dzone.com/articles/post-monolith-architecture-2025

https://www.youtube.com/watch?v=KSiW6lJr7O0

https://easy-software.com/en/newsroom/microservices-vs-monolith/

https://kitrum.com/blog/is-microservice-architecture-still-a-trend/

And many more. Just google it.

No one style fits all

An *architectural style* dictates certain principles that govern the solution being built. Example architectural styles we have discussed include Monolith and Microservices.

Microservices microservices encompass many of the best practices for software engineering, and it is therefore increasingly popular. However, like any architectural style, it does not fit every use case. Furthermore, microservices requires careful design.

References

References

Microservices

1. Definition and advantages of microservices
   - https://martinfowler.com/microservices/
   - https://developers.redhat.com/articles/2022/01/11/5-design-principles-microservices#five_design_principles_for_microservices
   - https://medium.com/platform-engineer/microservices-design-guide-eca0b799a7e8
   - "Cloud-Native Computing: A Survey from the Perspective of Services", https://arxiv.org/pdf/2306.14402.pdf
   - https://www.capitalone.com/tech/software-engineering/microservices-benefits/
2. Tradeoffs
   - https://martinfowler.com/articles/microservice-trade-offs.html
3. Good and bad practices in building microservices
   - "Migrating towards Microservices: Migration and Architecture", Smells", A. Carrasco B. van Bladel, S. Demeyer, IwoR '18, September 4, 2018, Montpellier, France. ACM
   - "Proposing a Dynamic Executive Microservices Architecture Model for AI Systems ", https://arxiv.org/pdf/2308.05833.pdf (Overview of microservices compared to other architectural styles).
4. Challenges & solutions in building microservices
   - https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/distributed-data-management
   - https://microservices.io/patterns/data/shared-database.html
   - https://arxiv.org/pdf/2306.15792.pdf "Sidecars on the Central Lane: Impact of Network Proxies on Microservices", HotInfra'23, June 18, 2023, Orlando, FL, USA
   - https://medium.com/geekculture/why-microservices-is-not-always-the-best-choice-df5b3b10babb

References

Microservices

4. Challenges & solutions in building microservices
   - https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/distributed-data-management
   - https://microservices.io/patterns/data/shared-database.html
   - https://arxiv.org/pdf/2306.15792.pdf "Sidecars on the Central Lane: Impact of Network Proxies on Microservices", HotInfra'23, June 18, 2023, Orlando, FL, USA
   - https://www.ibm.com/think/insights/microservices-advantages-disadvantages
   - https://medium.com/@emijiang6/rethinking-microservices-dcf9696af385
5. Communicating between microservices, evolving APIs
   - https://arxiv.org/pdf/2311.08175.pdf "Microservice API Evolution in Practice: A Study on Strategies and Challenges"

References

5. Replatforming a monolith legacy system to microservices
   - “Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review”, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 49, NO. 8, AUGUST 2023
   - https://blog.dreamfactory.com/microservices-examples
   - https://www.cloudamqp.com/blog/breaking-down-a-monolithic-system-into-microservices.html
   - https://blog.dreamfactory.com/microservices-examples
   - https://www.cloudamqp.com/blog/breaking-down-a-monolithic-system-into-microservices.html

Example microservice

The ”sock shop”, https://microservices-demo.github.io

Microservices and Docker

6. Microservices and Docker
   - Leveraging microservices architecture by using Docker technology, D. Jaramillo, et. al., SoutheastCon 2016, Norfolk, VA https://ieeexplore.ieee.org/document/7506647
   - https://gabrieltanner.org/blog/docker-compose/
   - https://cloudnweb.dev/2020/10/everything-you-need-to-know-about-docker-volume/
   - https://techmormo.com/posts/category/networking-in-docker/

References

5. Microservices and scaling
   - “ChainsFormer: A Chain Latency-aware Resource Provisioning Approach for Microservices Cluster”, https://arxiv.org/pdf/2309.12592.pdf
   - “Analytically-Driven Resource Management for Cloud-Native Microservices ”, https://arxiv.org/pdf/2401.02920.pdf
6. Testing Microservices
   - “Advanced White-Box Heuristics for Search-Based Fuzzing of REST APIs”, https://browse.arxiv.org/pdf/2309.08360.pdf
   - "Evaluating the Risk of Changes in a Microservices Architecture”, https://arxiv.org/pdf/2309.06238.pdf (Gives algorithm to estimate the risk that a change to a REST API will introduce a breakage)
   - “Exploring Behaviours of RESTful APIs in an Industrial Setting”, https://arxiv.org/pdf/2310.17318.pdf (uses OAS)

References

Migrating to a microservices architecture

7. Netflix. There is a lot written on Netflix and microservices. Here are a couple:
   - https://www.cs.binghamton.edu/~huilu/slidesSpring2022/Netflix-case-study.pdf
   - https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/
8. Amazon and Uber and others move to microservices architecture
   - https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/
   - https://newsletter.pragmaticengineer.com/p/real-world-eng-8
   - https://blog.khanacademy.org/go-services-one-goliath-project/
9. "Evaluating and Explaining Large Language Models for Code Using Syntactic Structures", https://arxiv.org/pdf/2308.03873.pdf. How to convert a monolith to microservices, with a focus on how to convert function invocations and communication between microservices.
10. "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 49, NO. 8, AUGUST 2023

Docker Volumes

- https://spin.atomicobject.com/2019/07/11/docker-volumes-explained/

References

New paradigms

1. Resource Allocation of Industry 4.0 Micro-Service Applications across Serverless Fog Federation, https://arxiv.org/pdf/2401.07194.pdf

Ongoing development debate

An often repeated debate between product management and software architecture

Evolving legacy to a microservices architecture

- There are many best practices that have been written on how this should be done.
  - A full discussion on this is beyond the scope of this course
- Suffice to say it is diffcult (and not always successful)
- The challenges are not only technical, but also financial, as portrayed by the following debate that often occurs within companies providing software solutions.

Product management, sales

Software architect, maintenance/support

The debate

- Focus on features that our customers want
- They do not care about the technology being used, they care about what they can do with the product
- We do not have the resources to develop new features and to redesign the technical platform.
- Focus on features!

Software architect, maintenance/support

Product management, sales

- The technical platform is outdated
- That causes new feature development to take longer and sometimes cannot be done at all.
- Customer support is flooded with problems due to the underlying technology
- We must update the technology base!

- Focus on features that our customers want
- They do not care about the technology being used, they care about what they can do with the product
- We do not have the resources to develop new features and to redesign the technical platform.
- Focus on features!

Software architect, maintenance/support

Product management, sales

Who is right?
What would you do?

atform is outdated
feature development
to take longer and sometimes cannot be done at all.
- Customer support is flooded with problems due to the underlying technology
- We must update the technology base!