Introduction to DevOps & Docker

What is DevOps?

> *"DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity."*

— Amazon Web Services (AWS)

What is DevOps?

> *"DevOps is a software development methodology that unites development and operations teams to deliver software faster, more securely, and more efficiently."*

— GitLab

What is DevOps?

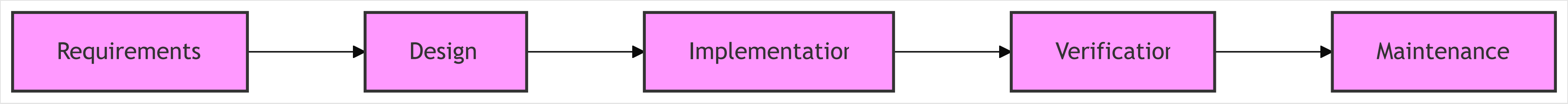
> *"DevOps is a set of practices that combines software development and IT operations to deliver software solutions more quickly, reliably, and stably. It fundamentally focuses on culture, automation, platform design, and constant feedback loops to enable faster, high-quality service delivery and greater business value."*

— Red Hat

The Evolution of Software Delivery

1. The Waterfall Model (The Old Way)

- **Linear Process**: Requirements → Design → Code → Test → Deploy.
- **The Problem**:
  - **Slow**: Testing happens at the end.
  - **Rigid**: Hard to change requirements once started.
  - **Risk**: "Big Bang" releases often fail.
  - **Silos**: Dev and Ops work separately.
  - **Feedback Delay:** Users see the product months later.
  - **Example**: Building a house without seeing it until it's done.



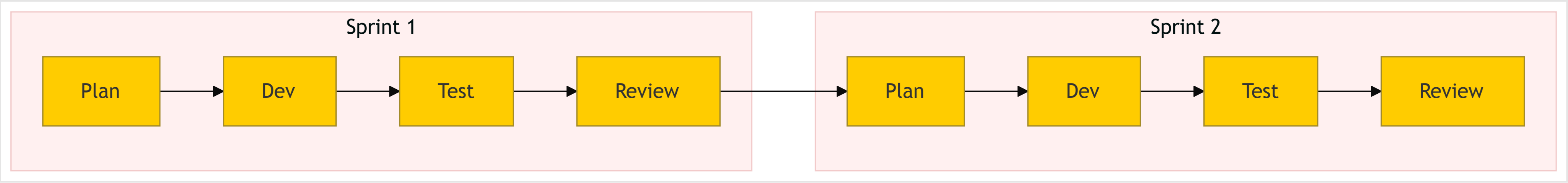
2. The "Wall of Confusion" (Silos)

- **Development**: Wants **Change** (New features, updates).
- **Operations**: Wants **Stability** (No bugs, uptime).
- **Result**: Conflict. Devs toss code "over the wall" to Ops. Ops struggle to run it.

> *"It worked on my machine!" — Every Developer, ever.*

3.1 Agile Methodology

- **Concept**: Break work into small, iterative cycles called **Sprints** (usually 2 weeks).
- **Goal**: Get feedback fast.
- **Process**: Plan → Dev → Test → Review → Repeat.
- **Benefits**:
  - Faster delivery of features.
  - More flexibility to change.
  - Continuous improvement based on feedback.

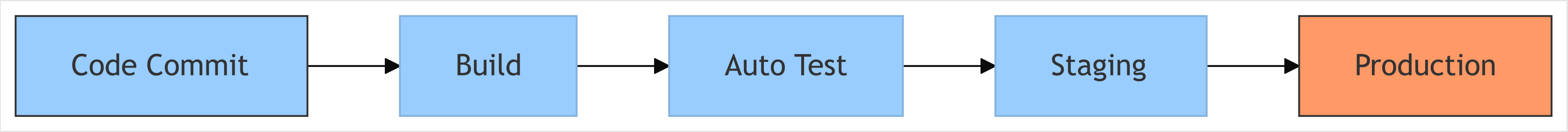


3.2 The "Agile Gap"

- **The Problem**: Agile teams focused on "Code Complete".
- **The Reality**: Code isn't valuable until it's **running in production**.
- **The Bottleneck**: Operations was still working in a "Waterfall" way (slow, manual deployments).
- **Solution**: We need to extend Agile principles **beyond** code, into Operations.
- **Enter DevOps**: Bridging Dev and Ops for faster, reliable delivery.

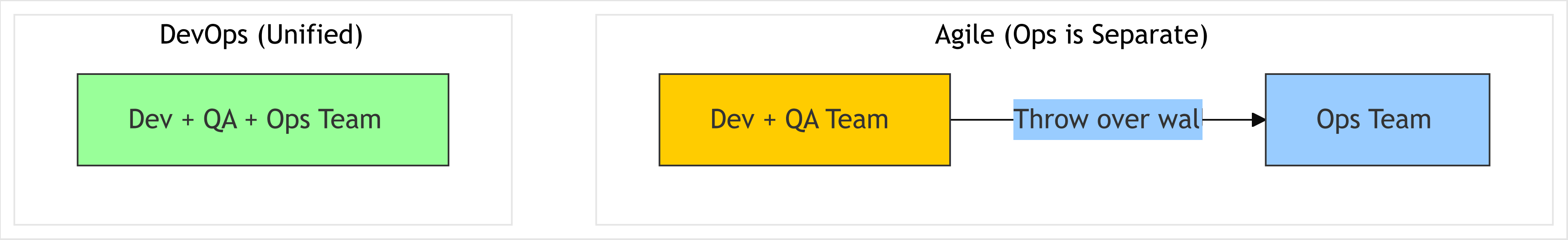
3.3 The Power of Automation

- **Manual**: Slow, Error-prone, Boring.
- **Automated**: Fast, Consistent, Scalable.
- **What do we automate?**
  - **Testing**: Run 1000s of tests in minutes.
  - **Infrastructure**: Build servers with code (IaC).
  - **Deployment**: Push to production with one click.
  - **Monitoring**: Automatically track performance & errors.
  - **Feedback**: Gather user data automatically.



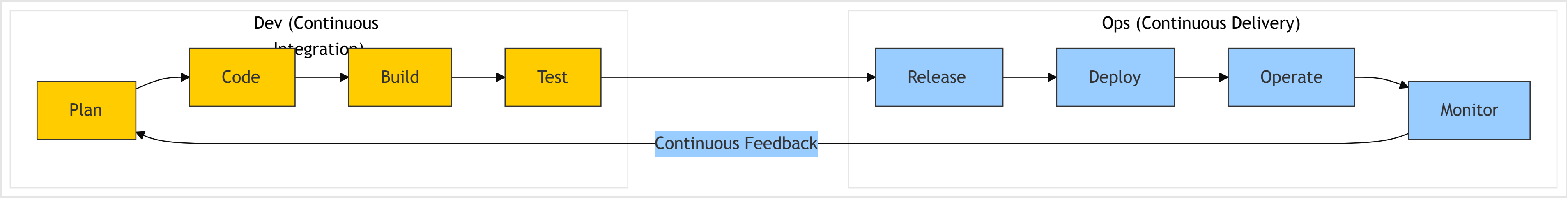
3.4 Agile vs DevOps: The Workflow Shift

- **Agile Focus**: Software Development speed. (Dev + QA)
- **DevOps Focus**: End-to-End Delivery speed. (Dev + QA + Ops)
- **The Shift**:
  - **Agile**: "Done" = Code is written & tested.
  - **DevOps**: "Done" = Code is running in production.



The DevOps Lifecycle (The Infinite Loop)

- **Continuous Integration (Dev)**:
    - **Plan**: Define features & requirements based on user feedback.
    - **Code**: Write code & commit to version control (Git).
    - **Build**: Compile code & create artifacts (Docker Images).
    - **Test**: Automated Unit & Integration tests.
- **Continuous Delivery (Ops)**:
    - **Release**: Versioning & Change Management.
    - **Deploy**: Push to Staging/Production environments.
    - **Operate**: Manage infrastructure & scaling.
    - **Monitor**: Track performance, logs, & **User Feedback**.
- **The Key**: **Feedback** from "Monitor" goes back to "Plan".
- **Architecture Note**: DevOps works best with **Microservices** (small, independent services) rather than Monoliths, as they can be deployed and scaled independently.
-

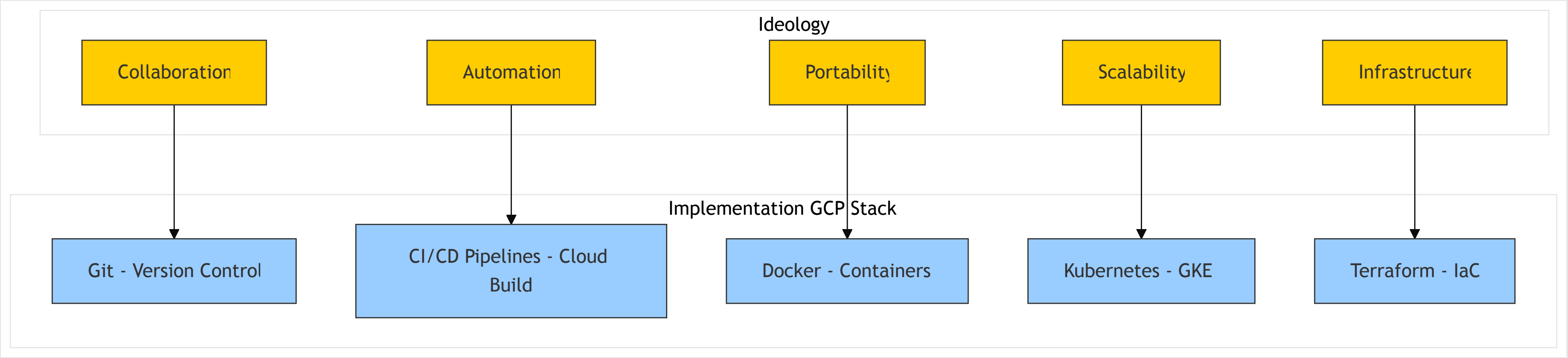

Dev (Continuous Integration)
Plan
Code
Build
Test
Ops (Continuous Delivery)
Release
Deploy
Operate
Monitor
Continuous Feedback

From Ideology to Implementation

- **The "As Code" Movement**: We treat **everything** like software.
- **Infrastructure as Code (IaC)**:
    - Instead of clicking buttons in AWS/GCP, we write **Terraform** files.
- **Pipeline as Code**:
    - Instead of manual builds, we write **GitHub Actions** or **Cloud Build** (YAML).
- **Benefit**: Version control, Peer Review, and Rollbacks for **Infrastructure**.

The DevOps Toolchain (Mapping)

- **Ideology → Tool (Implementation)**
- **Collaboration → Git** (Shared History)
- **Automation → CI/CD Pipelines** (Cloud Build)
- **Portability → Docker** (Standard Package)
- **Scalability → Kubernetes** (GKE)
- **Infrastructure → Terraform** (IaC)



Docker

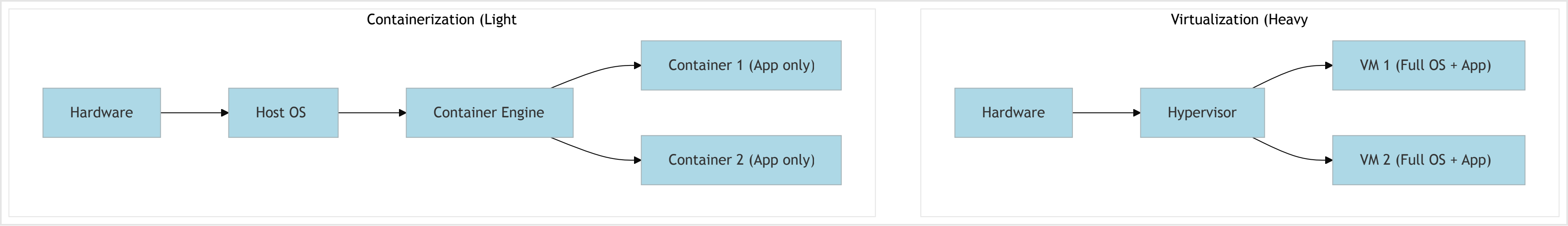
How we run apps? The Evolution

- **The Old Way (Bare Metal)**: Install everything directly on the server.
    - **Problem**: "Dependency Hell" - conflicts between apps.
- **The Better Way (Virtual Machines)**: Each app gets its own OS.
    - **Concept**: Running multiple "Virtual Machines" (VMs) on one physical server.
    - **Key Component**: **Hypervisor** (Software that manages VMs).
    - **Pros**: Complete isolation (OS level).
    - **Cons**: Heavy, slow to start, uses lots of RAM/CPU.
- **The Modern Way (Containers)**: Lightweight isolation, shared OS.
    - **Solution**: Fast, efficient, portable.

Containerization (The New Way)

- **Concept**: Packaging an app with **only** what it needs (code + libraries).
- **Key Difference**: Shares the Host OS Kernel (no full OS per app).
- **Pros**: Lightweight, starts in seconds, portable.

Visual Comparison



Comparison: VMs vs Containers

Weight	Heavy (GBs)	Light (MBs)
Speed	Slow (Minutes to boot)	Fast (Seconds to start)
Isolation	Strong (Full OS)	Good (Process level)
Best For	Different OS needs	Microservices, Apps

When to use what?

Use Virtual Machines (VMs) when:

- You need to run **different Operating Systems** (e.g., Linux on Windows).
- You need **strong security isolation** (e.g., running untrusted code).
- You have legacy apps that need a full OS environment.

Use Containers (Docker) when:

- You want to **deploy applications** quickly.
- You want consistency (works on my machine = works in production).
- You are building **Microservices**.
- You want to maximize server efficiency (run more apps on less hardware).

How do Containers Actually Work?

- **It's just a Process**: A container is a standard Linux process, but isolated.
- **Shared Kernel**: It uses the **Host OS** kernel (the brain of the OS).
- **Isolation (Namespaces)**: The container **thinks** it's the only thing running. It can't see other processes.
- **Limits (Cgroups)**: We can limit how much CPU/RAM a container uses.

> ***Analogy***:
>
> - ***VM***: *A separate house with its own infrastructure.*
> - ***Container***: *A room in a shared house (shared plumbing/electric), but with a locked door.*

Core Docker Concepts

1. Dockerfile (The Recipe)

- A text file with instructions.
- Tells Docker **how** to build your app.
- Example: "Start with Python, copy my code, run `app.py`".

2. Image (The Blueprint)

- The result of building a Dockerfile.
- Read-only template.
- You can't change it once built (immutable).

3. Container (The House)

- A runnable instance of an Image.
- You can start, stop, and delete it.
- You can run many containers from one image.

4. Volume (The Storage)

- Containers are temporary If you delete them, data is lost.
- **Volumes** let you save data **outside** the container so it persists.

What is Docker?

- **The Tool**: The most popular platform for creating and running containers.
- **The Promise**: "Build once, run anywhere."
- **Why it wins**:
  - Standardized format.
  - Huge ecosystem (Docker Hub).
  - Developer friendly.

Part 1: Containerize an Application

What We'll Build

- **Project**: A simple Todo List application
- **Tech Stack**: Node.js + React
- **Goal**: Package it into a Docker container
- **Why?**: Learn the basics of Dockerfiles and building images

Step 1: Get the Application

```
# Clone the sample app
git clone https://github.com/docker/getting-started-app.git
cd getting-started-app
```

- This is a pre-built Todo app
- We'll focus on **containerizing** it, not building it from scratch

Step 2: Create a Dockerfile

What is a Dockerfile?

- A text file with instructions to build an image
- Think of it as a **recipe** for your container

Create a file named `Dockerfile` in the project root

Dockerfile Example

```dockerfile
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "src/index.js"]
```

Understanding the Dockerfile

`FROM node:18-alpine`	Start with Node.js 18 (lightweight Alpine Linux)
`WORKDIR /app`	Set `/app` as the working directory
`COPY package*.json ./`	Copy package.json files first (for caching)
`RUN npm install`	Install Node.js dependencies
`COPY . .`	Copy all app files
`EXPOSE 3000`	Document that the app uses port 3000
`CMD ["node", "src/index.js"]`	Command to run when container starts

Step 3: Build the Image

```
# Build the image
docker build -t getting-started .
```

What's happening?

- `-t getting-started` = Tag (name) the image "getting-started"
- `.` = Use the current directory (where Dockerfile is)

Output: Docker reads the Dockerfile and creates an image layer by layer

Step 4: Run the Container

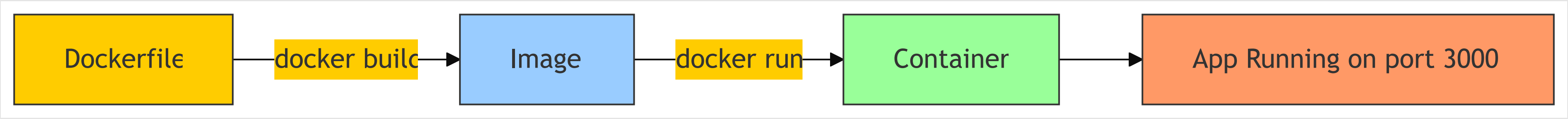
```
# Run the container
docker run -d -p 3000:3000 getting-started
```

Flags explained:

- `-d` = Detached mode (run in background)
- `-p 3000:3000` = Map port 3000 on host to port 3000 in container
- `getting-started` = The image name

Test it: Open `http://localhost:3000` in your browser!

Visual: Build → Run Flow



Part 2: Update the Application

The Problem

- You made changes to your code
- The old container is still running the **old version**
- **Question**: How do we update it?

Step 1: Modify the Code

Edit `src/static/js/app.js` (line 56):

```
// OLD
<p className="text-center">No items yet! Add one above!</p>

// NEW
<p className="text-center">You have no todo items yet! Add one above!</p>
```

Step 2: Rebuild the Image

```
# Rebuild with the same tag
docker build -t getting-started .
```

- Docker creates a **new version** of the image
- The old container is still running the old image

Step 3: Stop and Remove Old Container

```
# List running containers
docker ps

# Stop the container
docker stop <container-id>

# Remove the container
docker rm <container-id>

# OR do it in one command
docker rm -f <container-id>
```

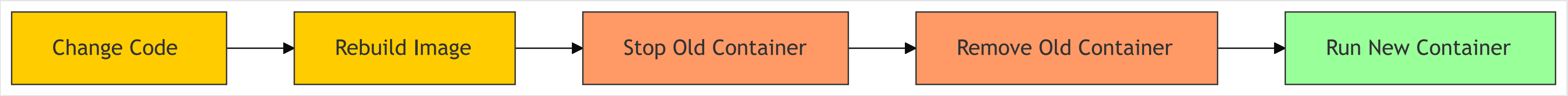
Why remove? You can't run two containers on the same port!

Step 4: Start the New Container

```
# Run the updated image
docker run -d -p 3000:3000 getting-started
```

Check the browser: You should see the updated text!

Visual: Update Workflow



Part 3: Share the Application

Why Share Images?

- **Collaboration**: Other developers can run your app
- **Deployment**: Run on servers, cloud, anywhere
- **Docker Hub**: Like GitHub, but for Docker images

Step 1: Create a Docker Hub Account

1. Go to **hub.docker.com**
2. Sign up for a free account
3. Remember your username!

Step 2: Create a Repository

- On Docker Hub, click **"Create Repository"**
- **Name**: `getting-started`
- **Visibility**: Public
- Click **"Create"**

Step 3: Login to Docker Hub

```
# Login from terminal
docker login
```

- Enter your Docker Hub username and password
- You'll see: `Login Succeeded`

Step 4: Tag Your Image

```
# Tag format: username/repository:tag
docker tag getting-started YOUR_USERNAME/getting-started
```

Example:

```
docker tag getting-started johndoe/getting-started
```

- This creates an **alias** for your image
- The tag matches your Docker Hub repository

Step 5: Push to Docker Hub

```
# Push the image
docker push YOUR_USERNAME/getting-started
```

What happens?

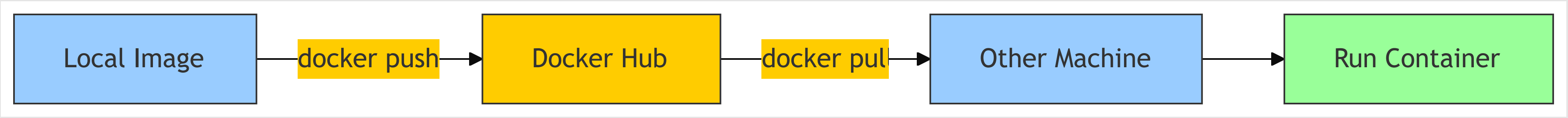
- Docker uploads your image layers to Docker Hub
- Anyone can now pull and run your image!

Step 6: Run on Another Machine

```
# On ANY machine with Docker installed
docker run -d -p 3000:3000 YOUR_USERNAME/getting-started
```

- Docker automatically **pulls** the image from Docker Hub
- No need to have the source code!

Visual: Push and Pull



Local Image

docker push

Docker Hub

docker pull

Other Machine

Run Container

Part 4: Persist the DB

The Problem

- Add some todo items to your app
- Stop and remove the container
- Start a new container
- **Result**: Your todos are GONE! 😱

Why? Container filesystems are **ephemeral** (temporary)

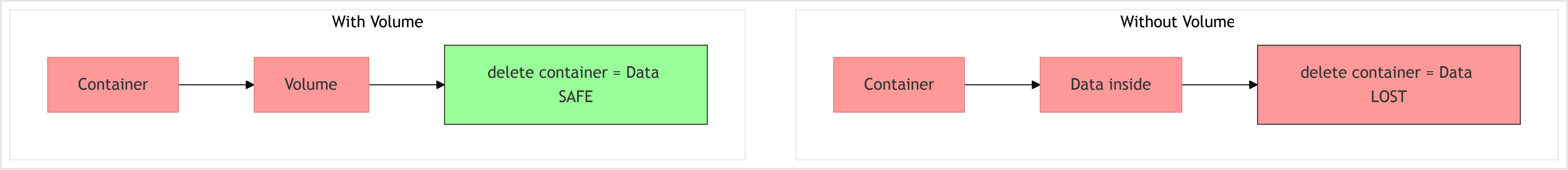
Understanding Container Filesystem

- Each container has its own isolated filesystem
- When you **delete** a container, its data is **lost**
- **Solution**: Use **Volumes** to persist data

What is a Volume?

- A special directory that exists **outside** the container
- Managed by Docker
- Data survives container deletion
- Can be shared between containers

Visual: Container vs Volume Storage



Step 1: Create a Volume

```
# Create a named volume
docker volume create todo-db
```

- This creates a volume called `todo-db`
- Docker manages where it's stored on your host

Step 2: Run Container with Volume

```
# Stop and remove old container first
docker rm -f <container-id>

# Run with volume mounted
docker run -d -p 3000:3000 \
  -v todo-db:/etc/todos \
  getting-started
```

What's `-v todo-db:/etc/todos`?

- Mount the `todo-db` volume to `/etc/todos` inside the container
- The app stores its database at `/etc/todos`

Step 3: Test Persistence

1. Add some todo items
2. Stop and remove the container:

   ```
   docker rm -f <container-id>
   ```

3. Start a new container with the same volume:

   ```
   docker run -d -p 3000:3000 -v todo-db:/etc/todos getting-started
   ```

4. **Check**: Your todos are still there!
   ✅

Inspect the Volume

```
# List all volumes
docker volume ls

# Inspect volume details
docker volume inspect todo-db
```

Output shows:

- Where Docker stores the data on your host
- Volume metadata

Part 5: Use Bind Mounts

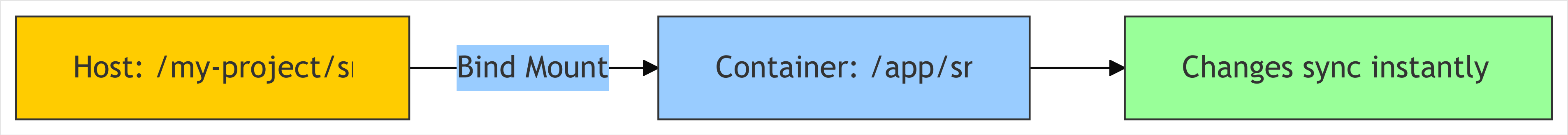
Volumes vs Bind Mounts

Managed by	Docker	You (manual path)
Location	Docker's storage	Any path on host
Best for	Production data	Development (live code)
Example	Database storage	Source code editing

Why Bind Mounts for Development?

- **Problem**: Every code change requires rebuilding the image
- **Solution**: Mount your source code directly into the container
- **Result**: Edit code → See changes instantly (no rebuild!)

Visual: Bind Mount Concept



Step 1: Run with Bind Mount

```
# Run container with bind mount
docker run -d -p 3000:3000 \
  -v "$(pwd):/app" \
  -v /app/node_modules \
  getting-started
```

Explanation:

- `-v "$(pwd):/app"` = Mount current directory to `/app` in container
- `-v /app/node_modules` = Prevent overwriting node_modules (keep container's version)

Step 2: Enable Live Reload

For Node.js apps, use `nodemon` to watch for changes:

Update your `package.json`:

```
{
  "scripts": {
    "dev": "nodemon src/index.js"
  }
}
```

Run with:

```
docker run -d -p 3000:3000 \
  -v "$(pwd):/app" \
  -v /app/node_modules \
  getting-started \
  npm run dev
```

Step 3: Test Live Reload

1. Edit `src/static/js/app.js`
2. Save the file
3. **Refresh browser** → Changes appear instantly!
4. No need to rebuild or restart container

Part 6: Multi-Container Apps

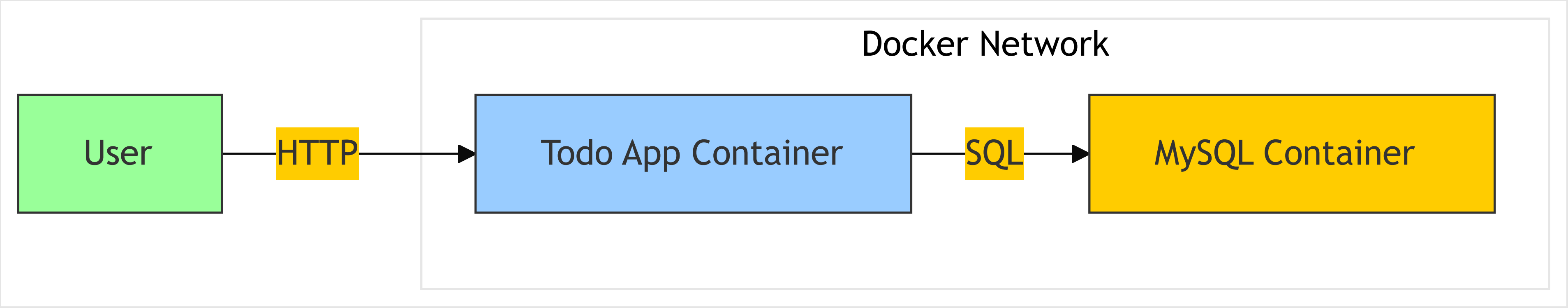
The Challenge

- Our app currently uses SQLite (file-based database)
- **Better approach**: Use MySQL in a separate container
- **Why separate?**
  - Scale independently
  - Update database without touching app
  - Follow microservices best practices

Container Networking

- Containers are **isolated** by default
- To communicate, they need to be on the same **network**
- Docker provides built-in networking

Visual: Multi-Container Architecture



Step 1: Create a Network

```
# Create a custom network
docker network create todo-app
```

- Containers on this network can talk to each other
- They can use container names as hostnames

Step 2: Start MySQL Container

```
docker run -d \
  --network todo-app \
  --network-alias mysql \
  -v todo-mysql-data:/var/lib/mysql \
  -e MYSQL_ROOT_PASSWORD=secret \
  -e MYSQL_DATABASE=todos \
  mysql:8.0
```

Breakdown:

- `--network todo-app` = Join the network
- `--network-alias mysql` = Other containers can reach it as "mysql"
- `-v todo-mysql-data:/var/lib/mysql` = Persist database data
- `-e MYSQL_ROOT_PASSWORD=secret` = Set root password
- `-e MYSQL_DATABASE=todos` = Create "todos" database

Step 3: Connect App to MySQL

```
docker run -d -p 3000:3000 \
  --network todo-app \
  -e MYSQL_HOST=mysql \
  -e MYSQL_USER=root \
  -e MYSQL_PASSWORD=secret \
  -e MYSQL_DB=todos \
  getting-started
```

Environment variables:

- `MYSQL_HOST=mysql` = Connect to the "mysql" container
- App automatically uses MySQL instead of SQLite

Step 4: Verify Connection

```
# Check MySQL logs
docker logs <mysql-container-id>

# You should see connection from the app
```

Test: Add todos in the app → They're stored in MySQL!

Part 7: Use Docker Compose

The Problem

- Running multi-container apps requires many commands
- Hard to remember all the flags and options
- **Solution**: Docker Compose

What is Docker Compose?

- A tool for defining multi-container apps
- Uses a YAML file (`docker-compose.yml`)
- Start everything with one command: `docker compose up`
- Stop everything with: `docker compose down`

Step 1: Create docker-compose.yml

Create `docker-compose.yml` in your project root:

```yaml
services:
  app:
    image: getting-started
    ports:
      - 3000:3000
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: secret
      MYSQL_DB: todos
    depends_on:
      - mysql

  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: todos
    volumes:
      - todo-mysql-data:/var/lib/mysql

volumes:
  todo-mysql-data:
```

Understanding the Compose File

Services = Containers to run

- `app` : Your todo application
- `mysql` : MySQL database

Key features:

- `ports` : Port mapping
- `environment` : Environment variables
- `depends_on` : Start order (mysql before app)
- `volumes` : Named volumes for persistence

Docker Compose automatically creates a network!

Step 2: Start the Application Stack

```
# Start all services
docker compose up -d
```

What happens?

- Docker Compose creates a network
- Starts MySQL container
- Starts app container
- All with one command!

Step 3: View Running Services

```
# List running services
docker compose ps

# View logs
docker compose logs -f

# View logs for specific service
docker compose logs -f app
```

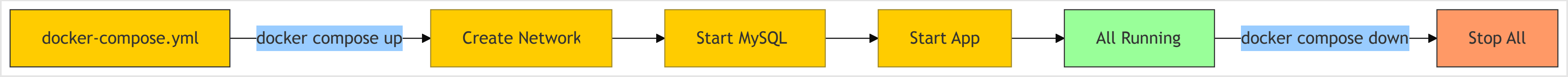
Step 4: Stop Everything

```
# Stop all services (keeps volumes)
docker compose down

# Stop and remove volumes
docker compose down --volumes
```

- One command stops and removes all containers
- Networks are automatically cleaned up

Visual: Docker Compose Workflow



Bonus: Development with Compose

Add bind mount for development:

```yaml
services:
  app:
    build: .  # Build from Dockerfile
    ports:
      - 3000:3000
    volumes:
      - ./:/app  # Bind mount for live reload
      - /app/node_modules
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: secret
      MYSQL_DB: todos
    command: npm run dev
    depends_on:
      - mysql
  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: todos
    volumes:
      - todo-mysql-data:/var/lib/mysql
volumes:
  todo-mysql-data:
```

Challenge

build your first docker image , run it and push it to docker hub
and share it with us

Questions?

🐳
Happy Dockering!