Microservices Architecture

Twelve Factor App

The 12 Factor App

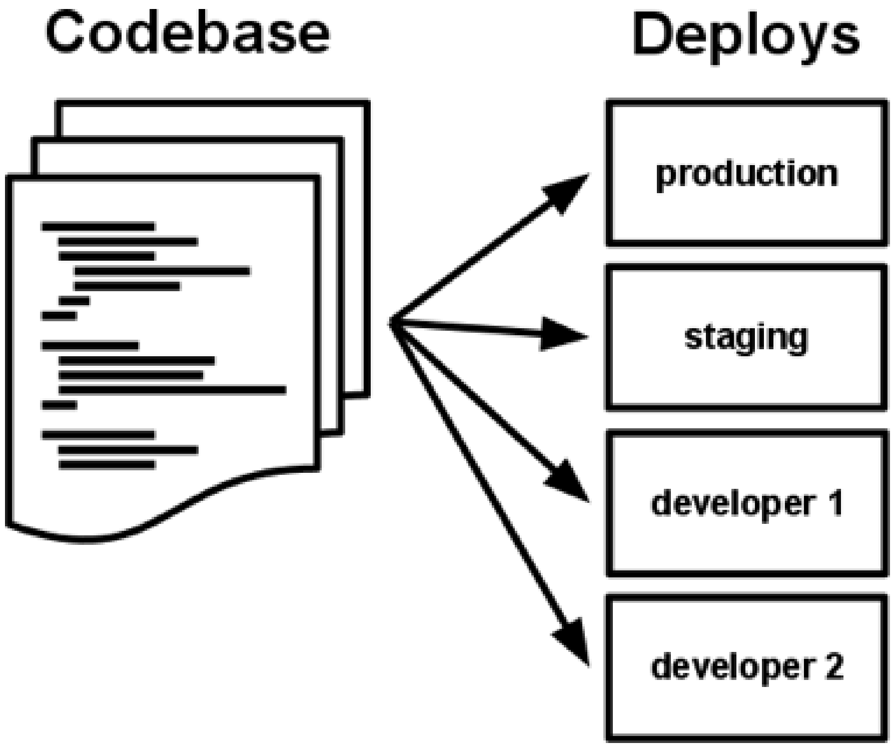
- Methodology for building SaaS apps (software as a service)
  - Drafted by developers at Heroku
  - First presented by Adam Wiggins circa 2011
- Applies to building SaaS apps that should:
  - Use declarative formats for setup automation
  - Have a clean contract with the underlying operating system (portable)
  - Are suitable for deployment on modern cloud platforms
  - Minimize divergence between development and production (DevOps)
  - Scale up without significant changes to tooling, architecture, or development practices
- Now considered a set of best practice guidelines for the design of microservices at the code architecture level

The 12 Factors

Factor	Description
I. Codebase	One codebase tracked in revision control, many deploys
II. Dependencies	Explicitly declare and isolate dependencies
III. Config	Store config in the environment
IV. Backing services	Treat backing services as attached resources
V. Build, release, run	Strictly separate build and run stages
VI. Processes	Execute the app as one or more stateless processes
VII. Port binding	Export services via port binding
VIII. Concurrency	Scale out via the process model
IX. Disposability	Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity	Keep development, staging, and production as similar as possible
XI. Logs	Treat logs as event streams
XII. Admin processes	Run admin/management tasks as one-off processes

Single Codebase

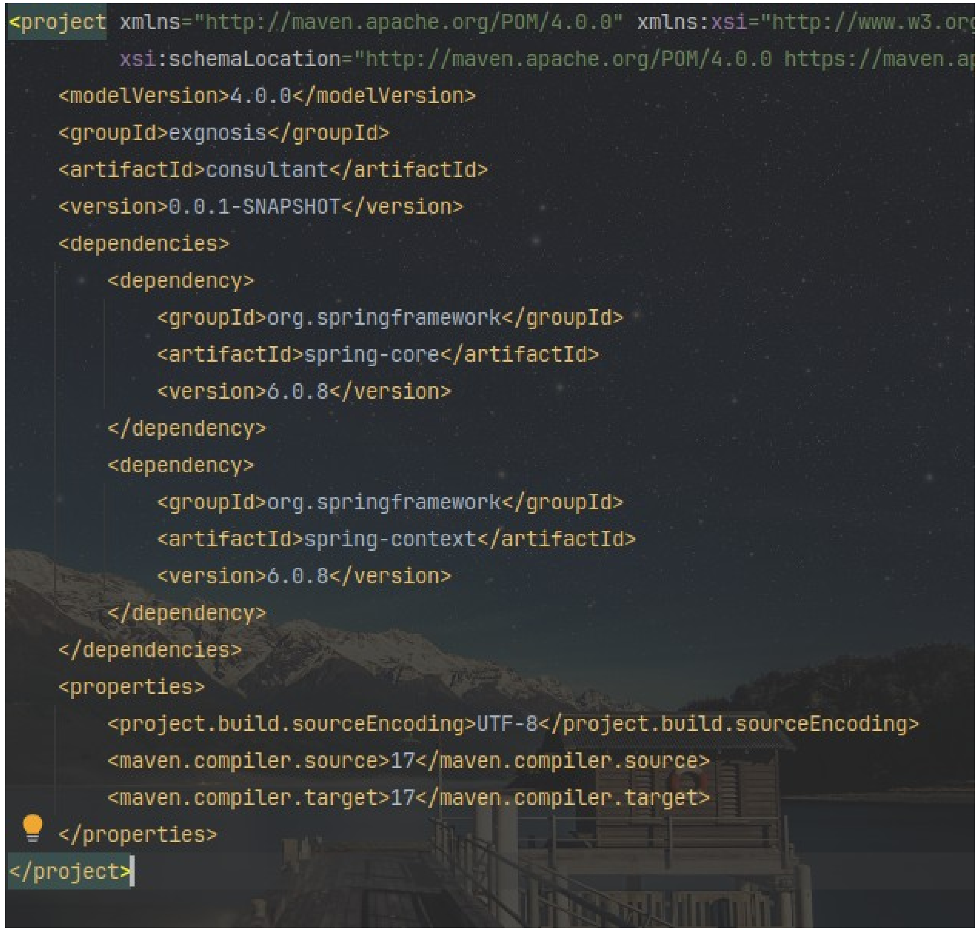
- A codebase is:
  - A set of repositories that share a root commit which means that they all derive from the same commit as a common starting point
- A deploy is a running instance of the app in a specific environment
  - Development, integration, testing, near production, production for example
- Multiple code bases are a distributed app
  - Each codebase should be refactored as microservice
- Multiple microservices sharing the same code
  - The shared code should be refactored into a library that becomes a dependency for the apps



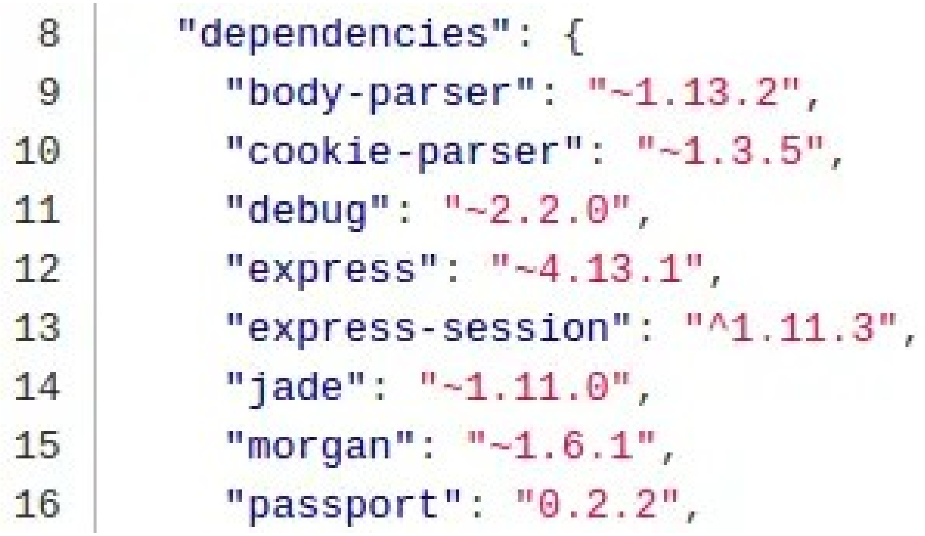
Externalize Dependencies

- All dependencies for a microservice are
  - Declared specifically and exactly in a dependency manifest
  - There are no implicit dependencies
  - This allows exact reproduction of specific versions of any deployment
- Uses a dependency isolation tool
  - Ensures that no implicit dependencies “leak in” from the environment
  - Each build executes in exactly the same way
  - Does not rely on the implicit existence of any system tools
- Many build tools provide dependency management

Externalize Dependencies



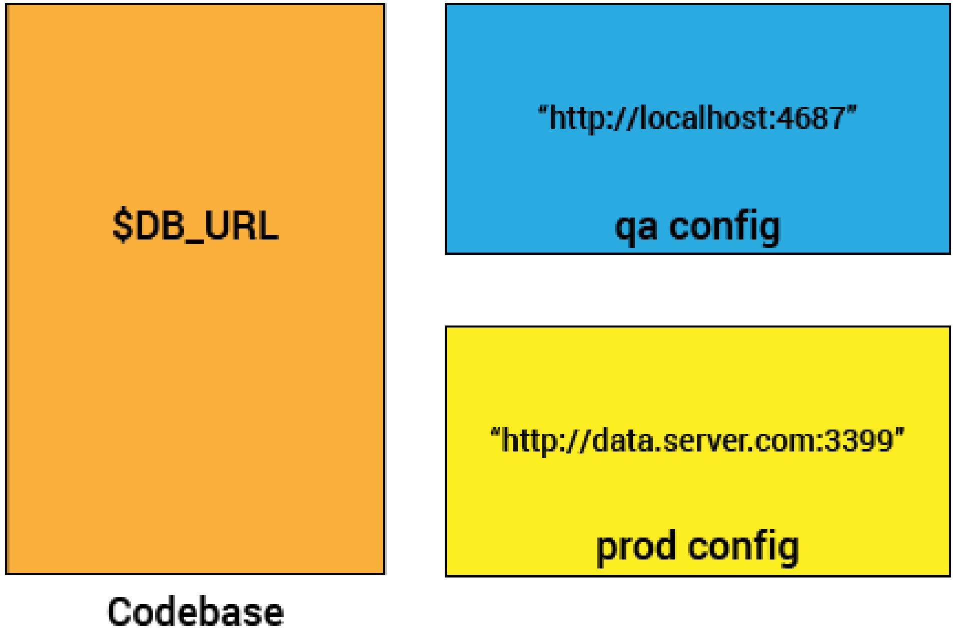
```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.ap
    <modelVersion>4.0.0</modelVersion>
    <groupId>exgnosis</groupId>
    <artifactId>consultant</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>6.0.8</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>6.0.8</version>
        </dependency>
    </dependencies>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
</project>
```



```json
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "express-session": "^1.11.3",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "passport": "0.2.2",
```

Store Config in Environment

- Configuration information is never kept in the code
  - Config info varies across deployments, code does not
- Config info includes:
  - Handles to databases or other services
  - Credentials to services like S3 buckets
  - Host names, external URLs or other identifying information
- All configuration information is
  - Kept externally as a set of configuration environment variables
  - Deployment tool sets the values of the environment variables in the deployed code



$DB_URL

Codebase

“http://localhost:4687”

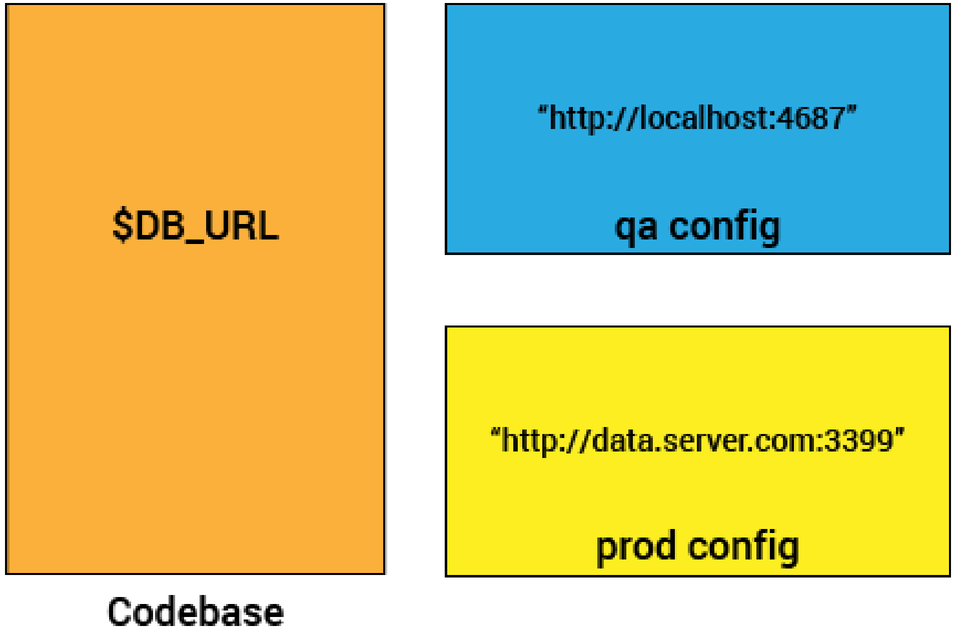
qa config

“http://data.server.com:3399”

prod config

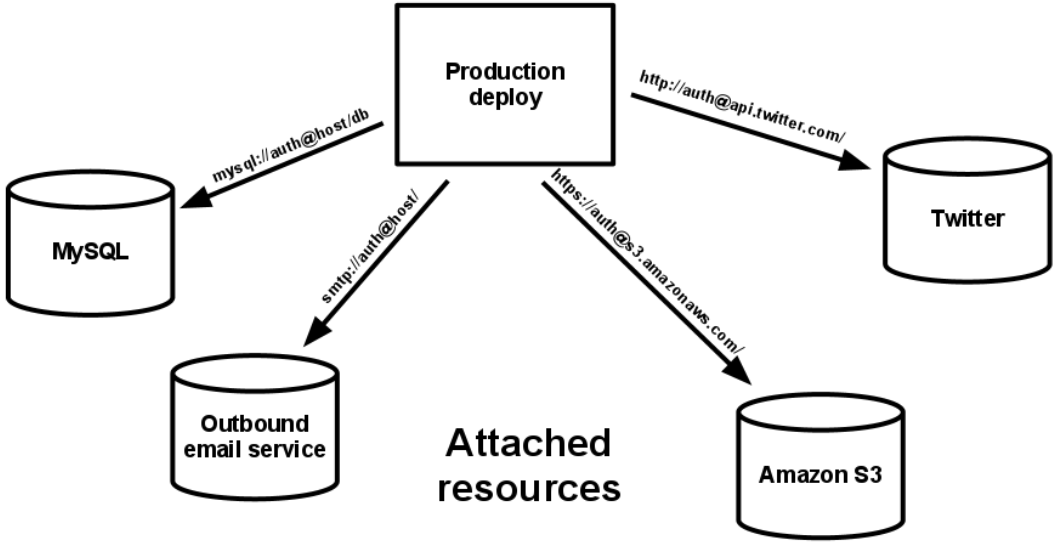
Store Config in Environment

- May be kept as config files
- Not recommended since
  - They might be added to code base
  - The may represent credential leakage if they are not stored properly
- Litmus test
  - Would any credentials be compromised if the codebase were made open source



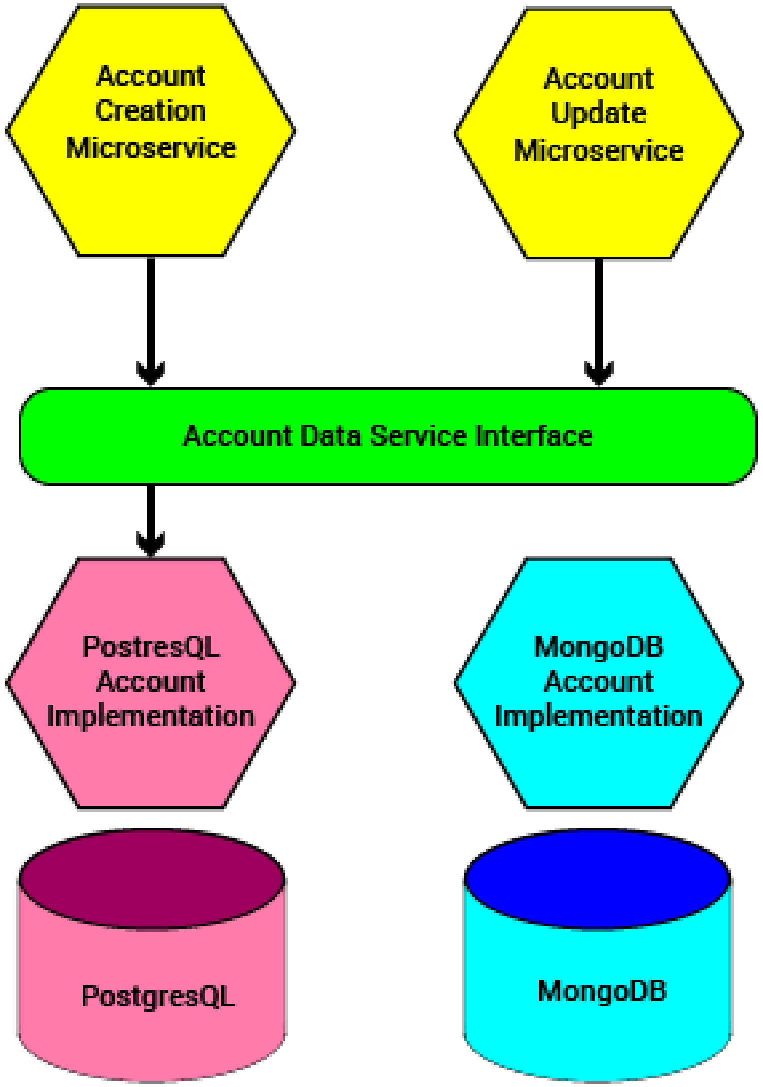
Backing Services

- A backing service is an infrastructure resource
  - Databases, queues, API services like geolocation, storage services, reporting and logging and others
- Backing services implement the service interface that microservices use
  - Dependency Inversion Principle
- The specific implementation of the service can be changed without needing to modify the microservice
- Reduces the coupling between the microservice and infrastructure



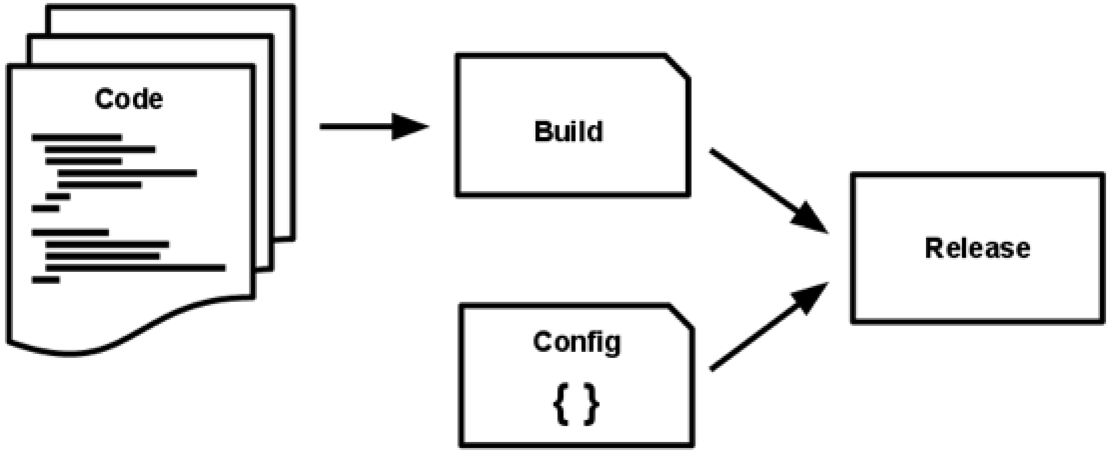
Backing Services

- Multiple microservices need to access persistent data
- Infrastructure layer provides a repository interface
  - May be implemented via different protocols
  - REST, stream, etc
- The repository service implements the interface using a specific resource
  - For example, can switch between different data storage implementations


Account Creation Microservice
Account Update Microservice
Account Data Service Interface
PostresQL Account Implementation
MongoDB Account Implementation
PostgresQL
MongoDB

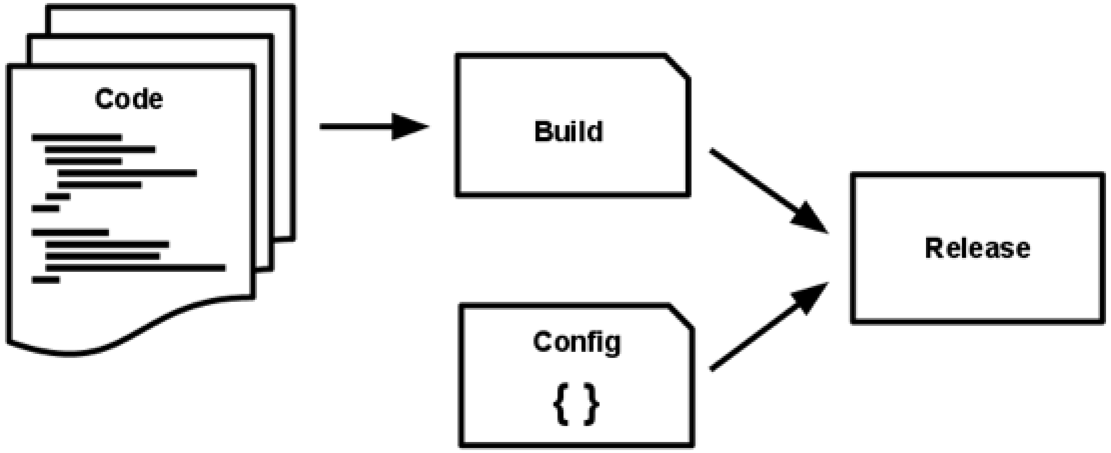
Build, Release and Run

- Codebase becomes a deploy via a three step process
- Build Stage
  - Code converted to an executable build
  - All dependencies resolved and integrated
- Release Stage
  - The build is combined with the config for a deploy to produce a release
  - The release is ready for execution in the deploy environment
- Run Stage
  - The release is run in the deploy environment
- Each step should be automated
  - CICD, continuous testing and DevOps



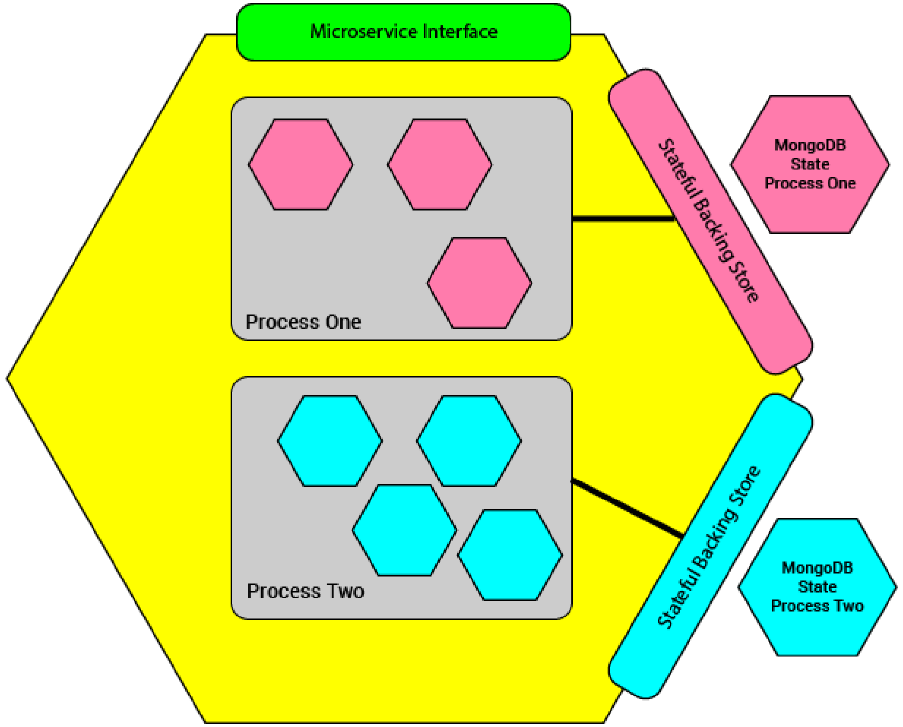
Build, Release and Run

- Each release should have a unique ID
- Releases cannot be changed after creation
  - Changes to the code or config require a new release to be created
  - A release is immutable
- Defective releases can be rolled back to a previous release
- Requires a process for release management



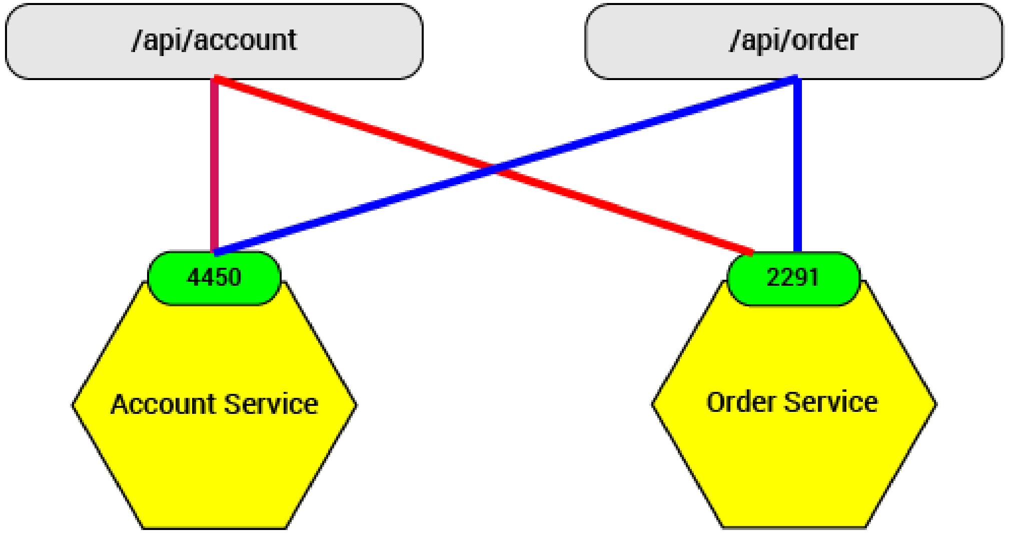
Processes

- An app is one of more stateless services
- State generally refers to the state of a transaction or job
  - These should be “re-entrant”
  - The state of the transaction is stored externally and persistently
- Loss of a single process does not mean that transaction state is lost
  - Another copy of the process can pick up and continue
- Local storage in a process should only be used for caching



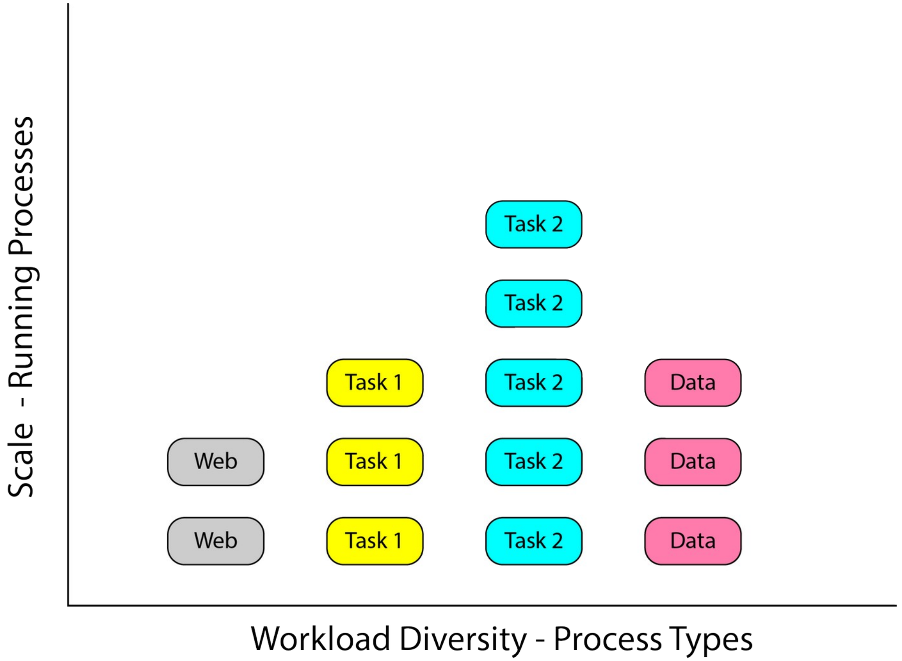
Port Binding

- The runtime app is completely self contained
- No knowledge of the run time environment should be injected into the app
- Functionality is exposed through an IP address and port
- The server routes messages to the app via its port
  - REST or streams like Kafka
- Service does not depend on the overall architecture
- This also includes backing services



Concurrency

- Processes are first class citizens
- Different tasks handled by different processes
  - HTTP requests handled by a web process
  - Persistence handled by a data process
- Direct benefit of the process model
- As load or demand increases
  - Processes can scale out horizontally



Disposability

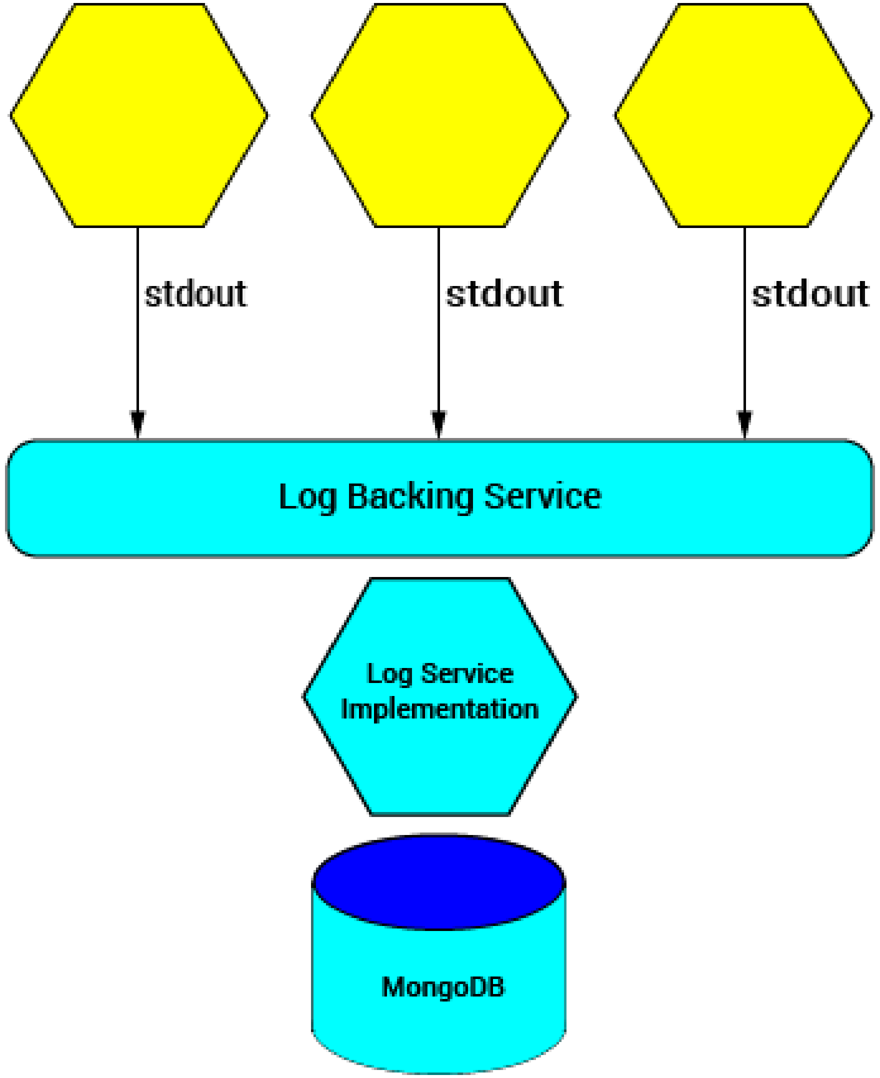
- Processes disposable
  - They can be stopped and started at a moment’s notice
- Startup time should be minimized
  - Supports rapid scaling on demand
- Should shut down gracefully
  - Stop listening on the port, allow any executing requests to finish, then shut down
- If the process is working on a job or transaction
  - Current state is updated in the persistent store and any locks released
- Processes should be robust against sudden death
  - For example a hardware failure
  - The orchestration environment should detect the process failure and return the job to the job queue

Dev-Prod Parity

- Keep deployment environments as similar as possible
  - Design for continuous development or deployment
- Backing services should be consistent
  - Implementation of the services differs between environments or deploys
  - But the backing service interface is constant
  - For example, a data service could be implemented by a local database in dev but a production database in the production environment
  - Implementation is determine by the config values for the deploy
- Code should never have to be modified just to run in the different deploys

Logs

- Treat logs as event streams
  - Time ordered stream of events collected from all running processes and backing services
- Individual processes should only have to write to a standard output
  - The streams are collated and routed for processing by the backing service
  - Processes have no knowledge of where their log output goes
- Event steams allow for
  - Long term data analytics
  - Real time monitoring and incident response
  - Forensic analysis of past events



Admin Processes

- Run admin/management tasks as one-off processes, for example
  - Database migrations
  - Running shells for live analysis of the process
  - Running one time maintenance scripts
- Admin code should
  - Run in the same environment as the application
  - Synchronized with the application code to avoid versioning issues
- Admin code may be part of a deploy
  - Provided the proper security measures are taken

End of Module