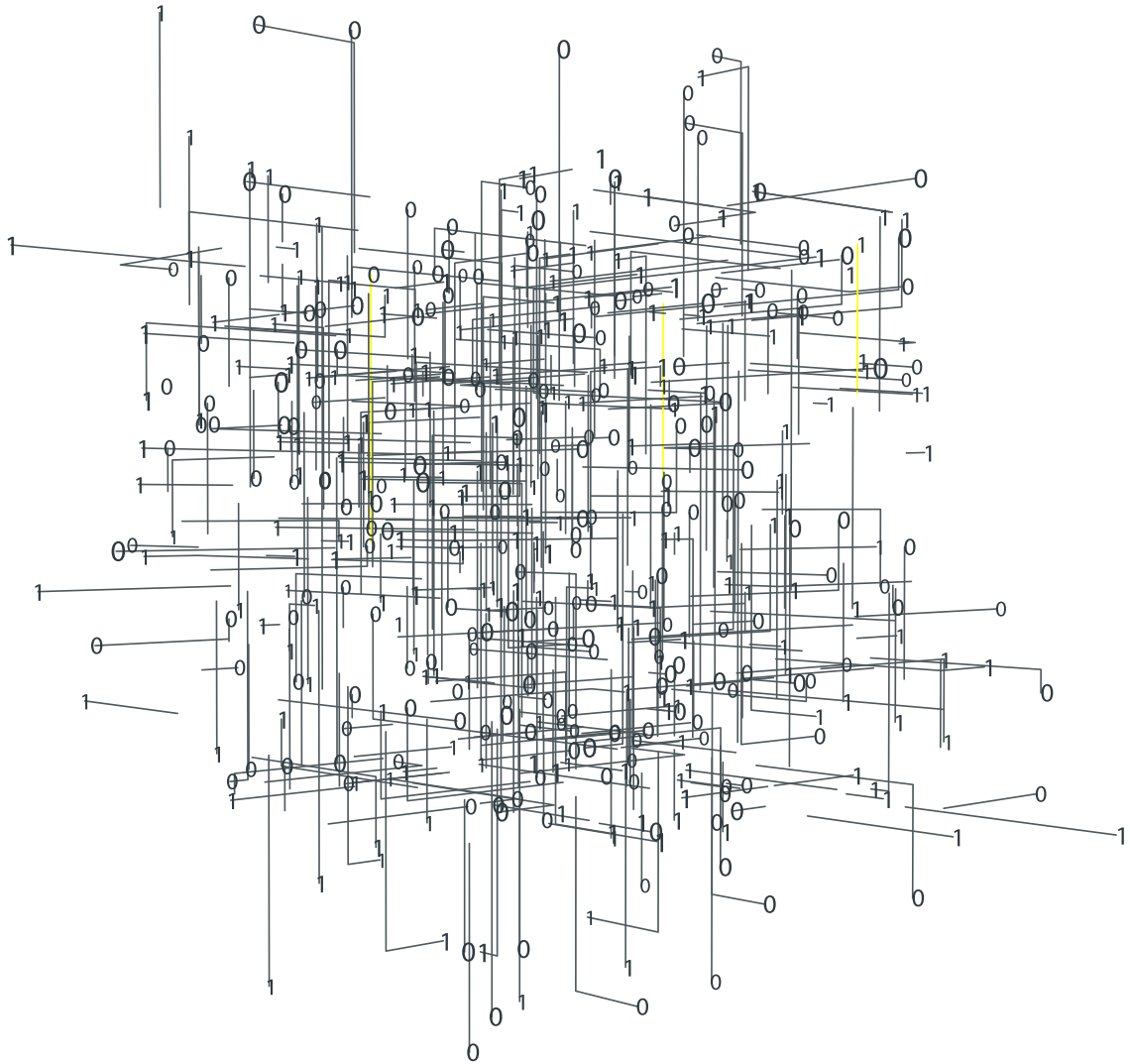
2DV604

APIs &.Deployment



Linnæus University

Today’s lecture

APIs

”composability”

Business agility

Open API

CD/CI

environments

”deployability”

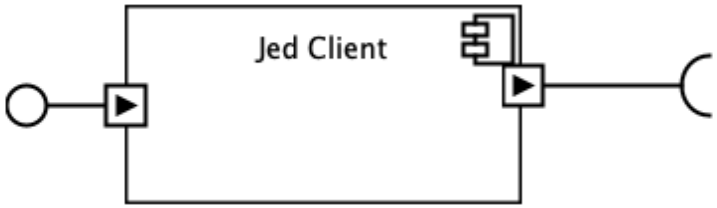
Patterns & tactics

“Composability,
Pieces fit, systems align,
Deploy with a breeze.”

— ChatGPT 4o

API-centric

API—“application programming interface”



”Design by contract”

Orchestration of systems

What is a Contract?

```
preconditions
postconditions
invariants
```

At least two parties

Customer(s) requesting services

Provider(s) providing services

The contract is the agreement between the customer and the provider

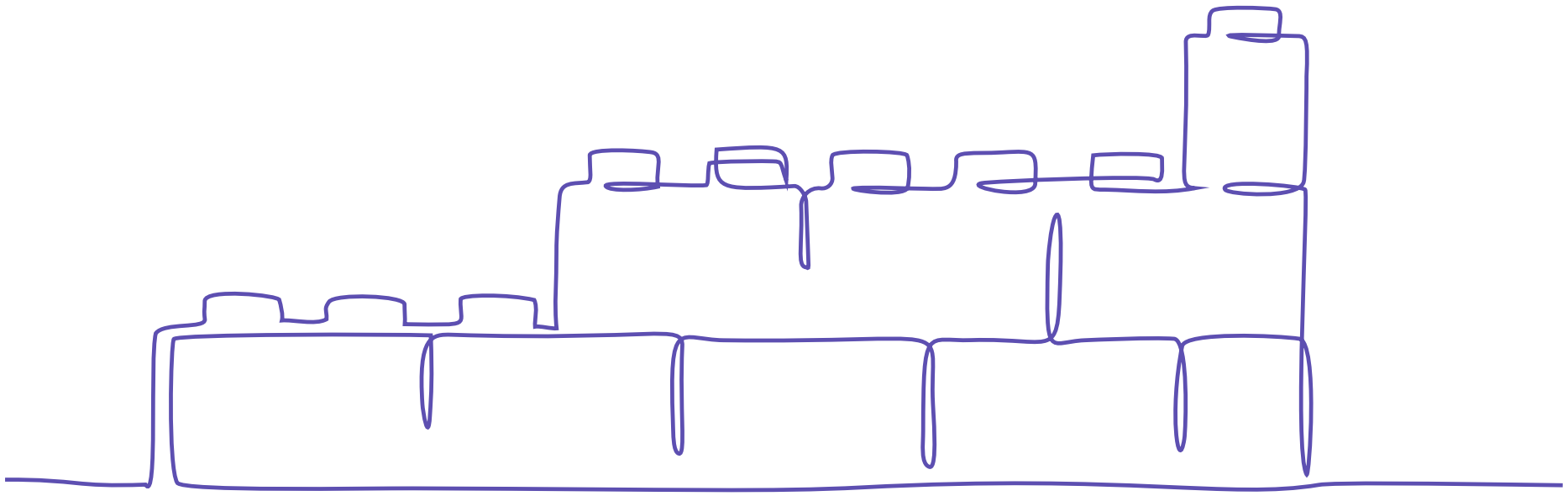
Two important characteristics of a contract

- Each party has expectations of the contract and is prepared to assume certain obligations to fulfil them.
- Benefits and obligations are documented in a contract document. The benefit to the customer is the supplier's obligation, and vice versa.

Composability

Composability a principle and a quality

"is a system design principle that deals with the inter-relationships of components. A highly composable system provides components that can be selected and assembled in various combinations to satisfy specific user requirements".

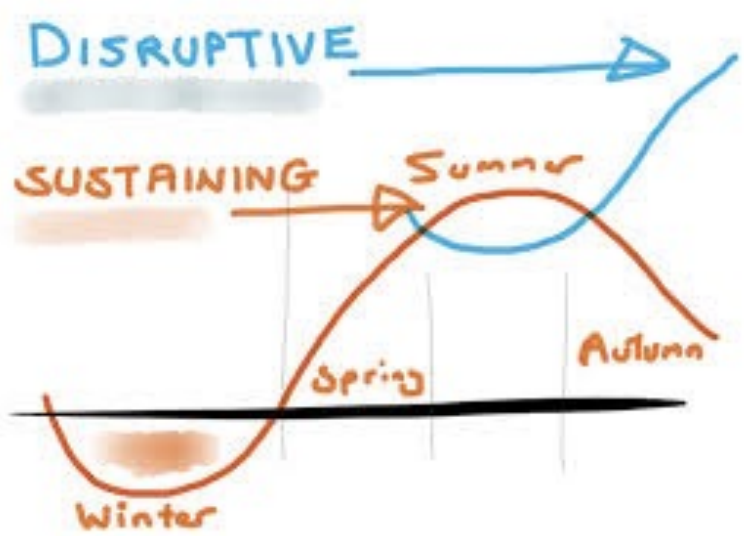


Source: wikipedia

Business agility

From projects to products

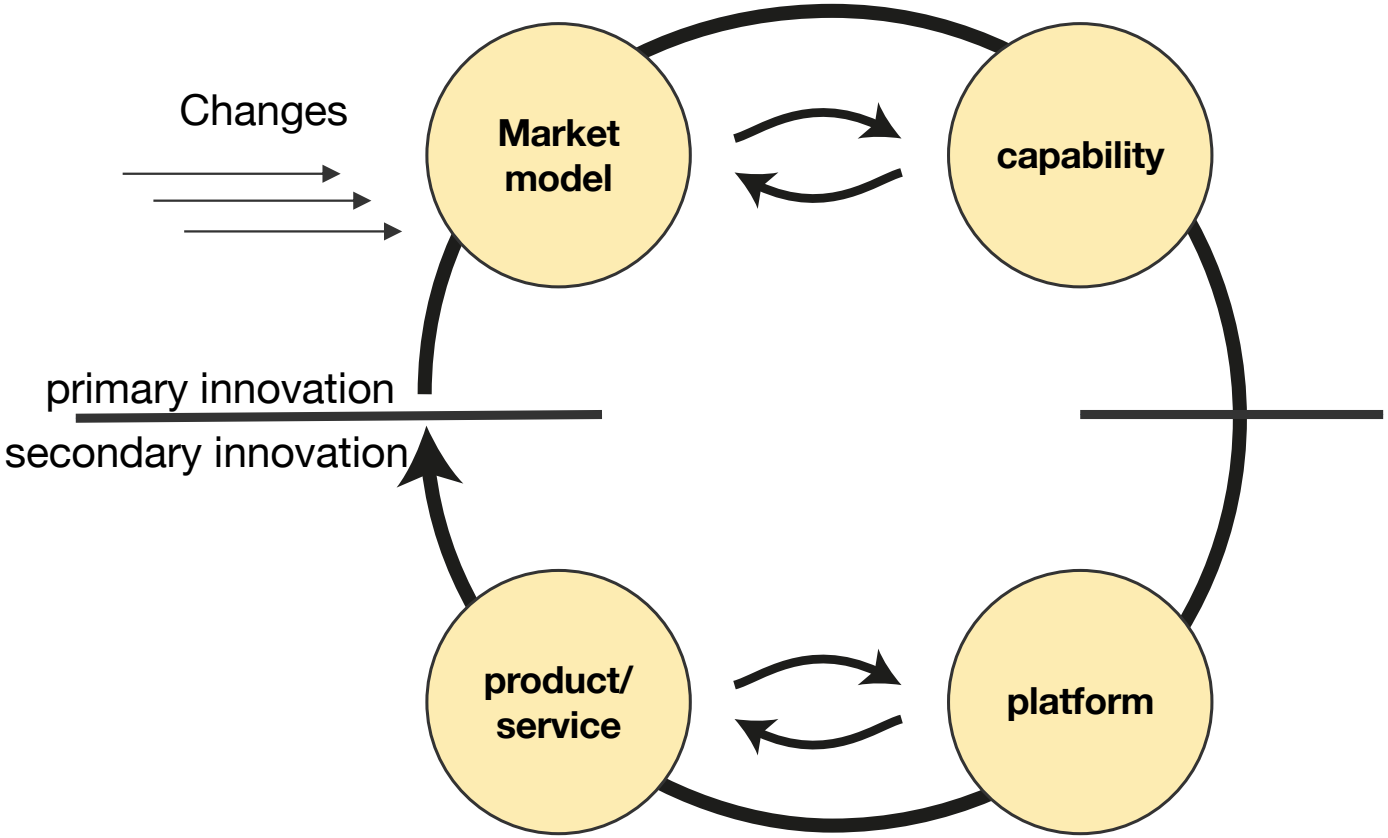
”Business agility is the ability of an organization to sense changes internally or externally and respond accordingly in order to deliver value to its customers.”



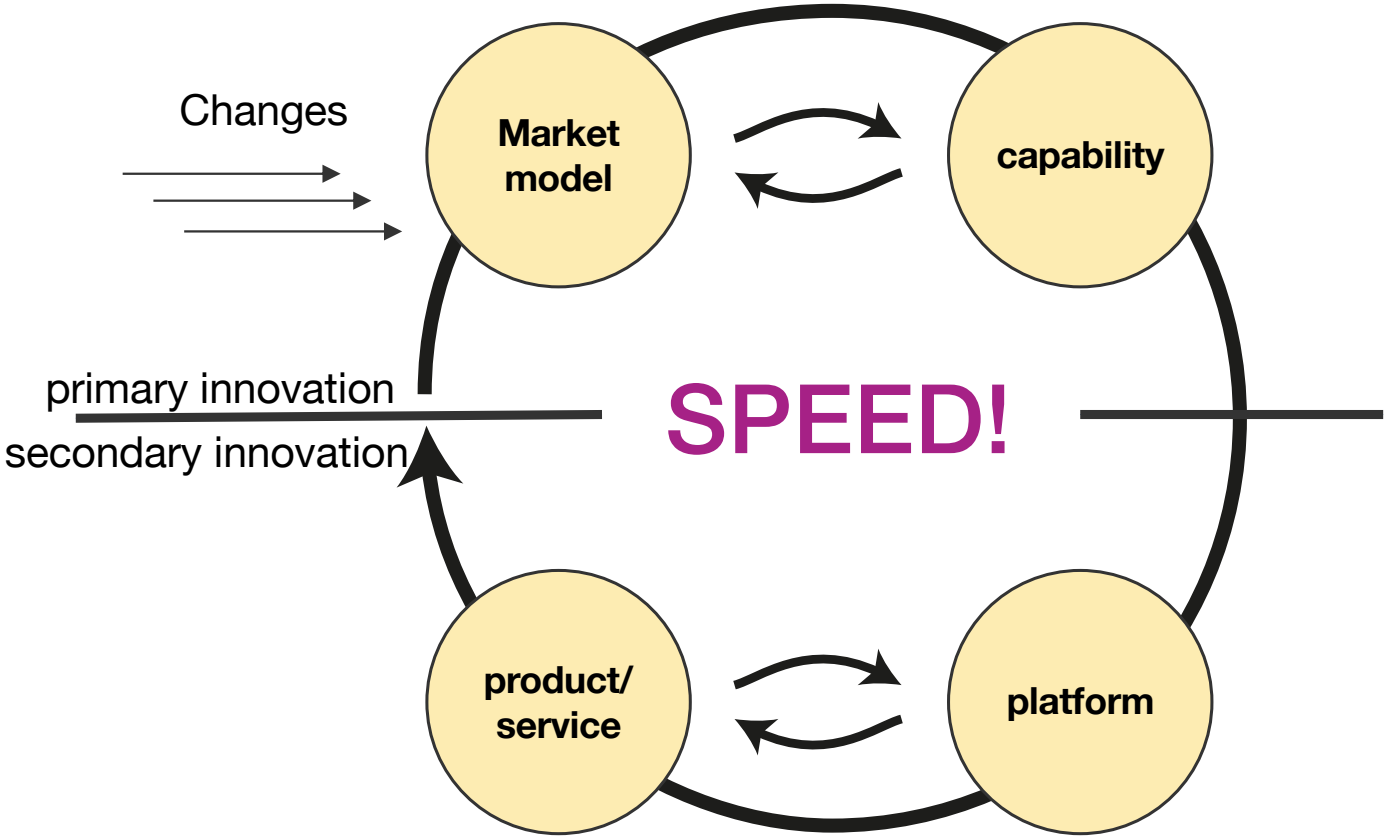
The Agile Manifesto

Reacting to changes instead of following a plan

Business agility



Business agility



Business Agility - Quality Attributes

”Enables or inhibits quality”

Composability

Deployability – a property of a system that indicates that it can be deployed – that is, allocated to an operating environment and run – within a predictable time and with an acceptable amount of work

Testability – how easily, efficiently and effectively an application can be tested

Availability – The likelihood that a system is not broken or repaired when it needs to be used.

Scalability – the ability of a system to handle a growing amount of work.

API-centric design

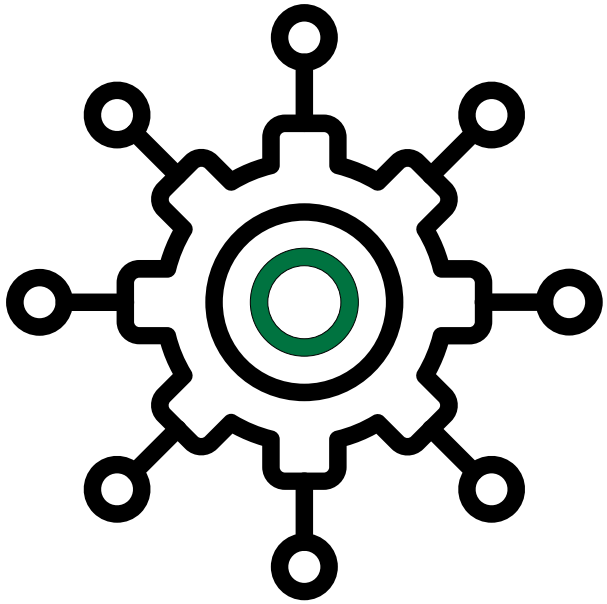
From an API

for communication

and integration

of a system

that is distributed



To multiple APIs

for reuse

as services

in multiple systems

often over the web

APIs are key elements of modern software architectures

APIs and Composability

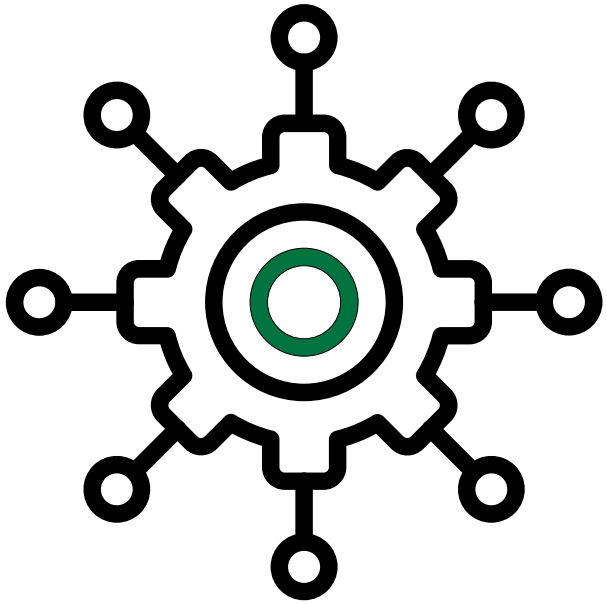
APIs enable composability

Public API:'s

Offers services

or extensions to platforms

Business APIs

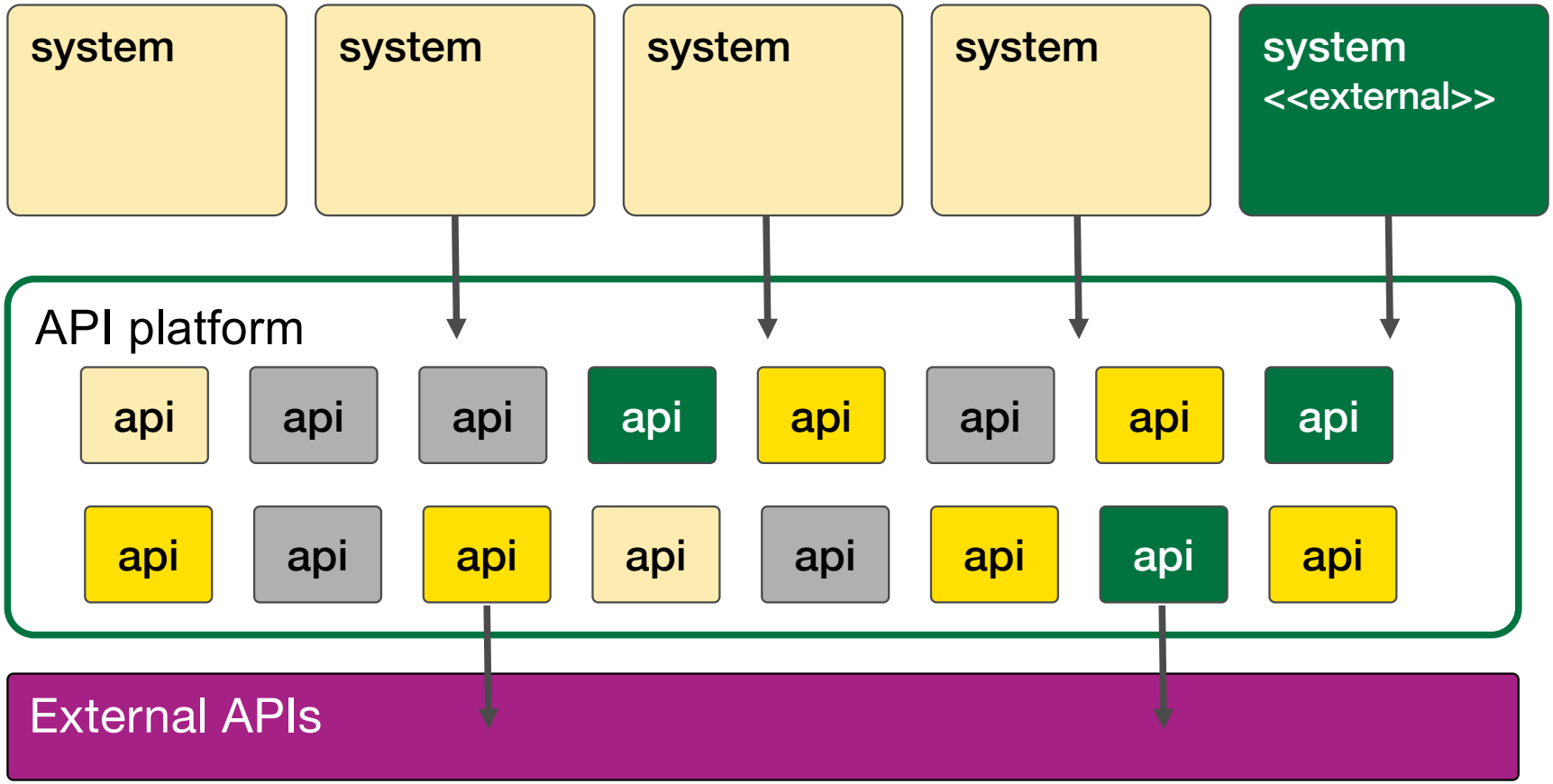


Example

Service

Platform

API-based platforms



Different API types

Remote Procedure Call, RPC – Clients call functions on a server.

Remote Method Invocation, RMI – RPC with objects!

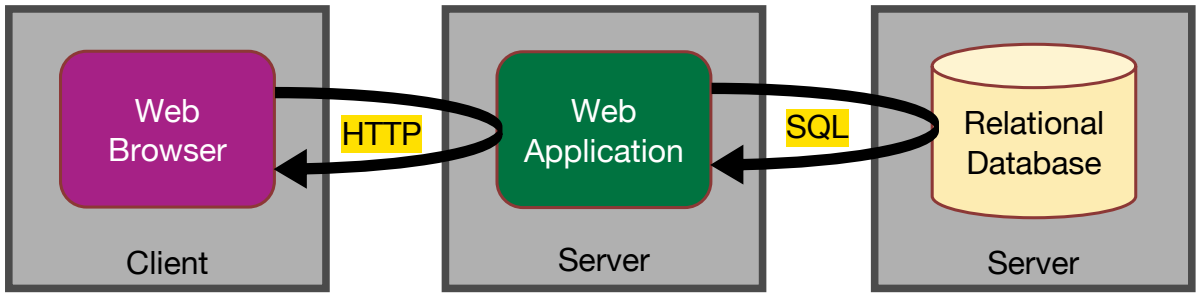
SOAP API	A protocol that uses XML to format messages	PayPal SOAP API, Salesforce API
REST API	Built on HTTP methods, stateless, returns typical JSON or XML.	GitHub API, Spotify API
GraphQL API	Clients ask "questions" on graph structures.	GitHub GraphQL API, Shopify API
WebSocket API	Enables full-duplex communication in real time.	Binance WebSocket API, Slack API

What is REST?

An architecture style for distributed "hyper-media systems" like Roy Fielding is first described in 2000.

a *Style* is a set of Patterns and Constraints:

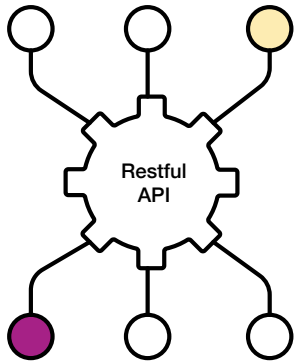
1. Client – Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code-On-Demand



REST?

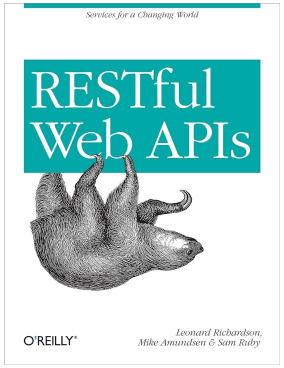
The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.

R. Fielding



Restfulness

Richardson's Maturity Model



Level	Key Concept	Description
Level 0	The Swamp of POX	Uses HTTP as transport only. No resources, only one endpoint for all actions.
Level 1	Resources	Resources (nouns) are introduced, but HTTP methods are still misused.
Level 2	HTTP Verbs (Methods)	Proper use of HTTP methods (GET, POST, PUT, DELETE) and status codes.
Level 3	Hypermedia Controls (HATEOAS)	Hypermedia links guide clients on what to do next, allowing dynamic exploration.

APIs that use HTTP

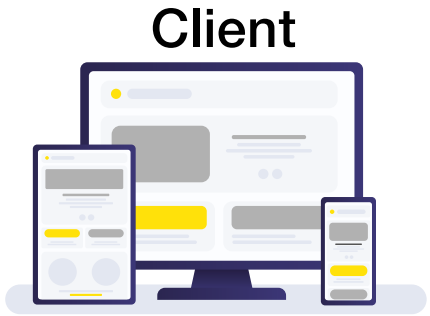
Uses URIs to discover resources.

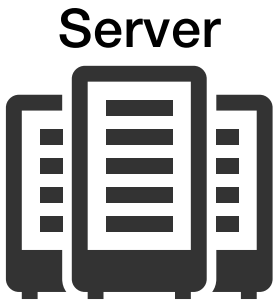
Using HTTP Methods to Specify Action:

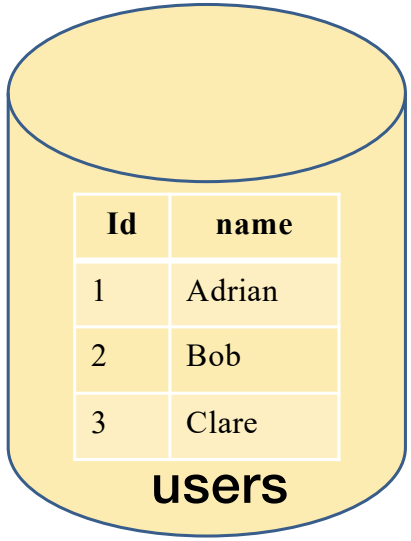
- Create: POST
- Retrieve: GET
- Update: PUT
- Delete: DELETE

Uses HTTP status code to indicate outages (e.g., errors).

How does REST ...?







State change.

```
{"id": 2, ”name": ”Bobby"}
```

```
GET /users/2
...
```

```
{"id": 2, ”name": ”Bob"}
```

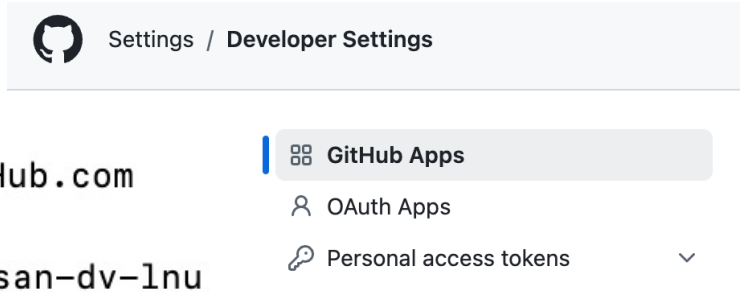
```
PUT /users/2
{"id": 2, ”name": ”Bobby"}
```

REST API - Example

Github

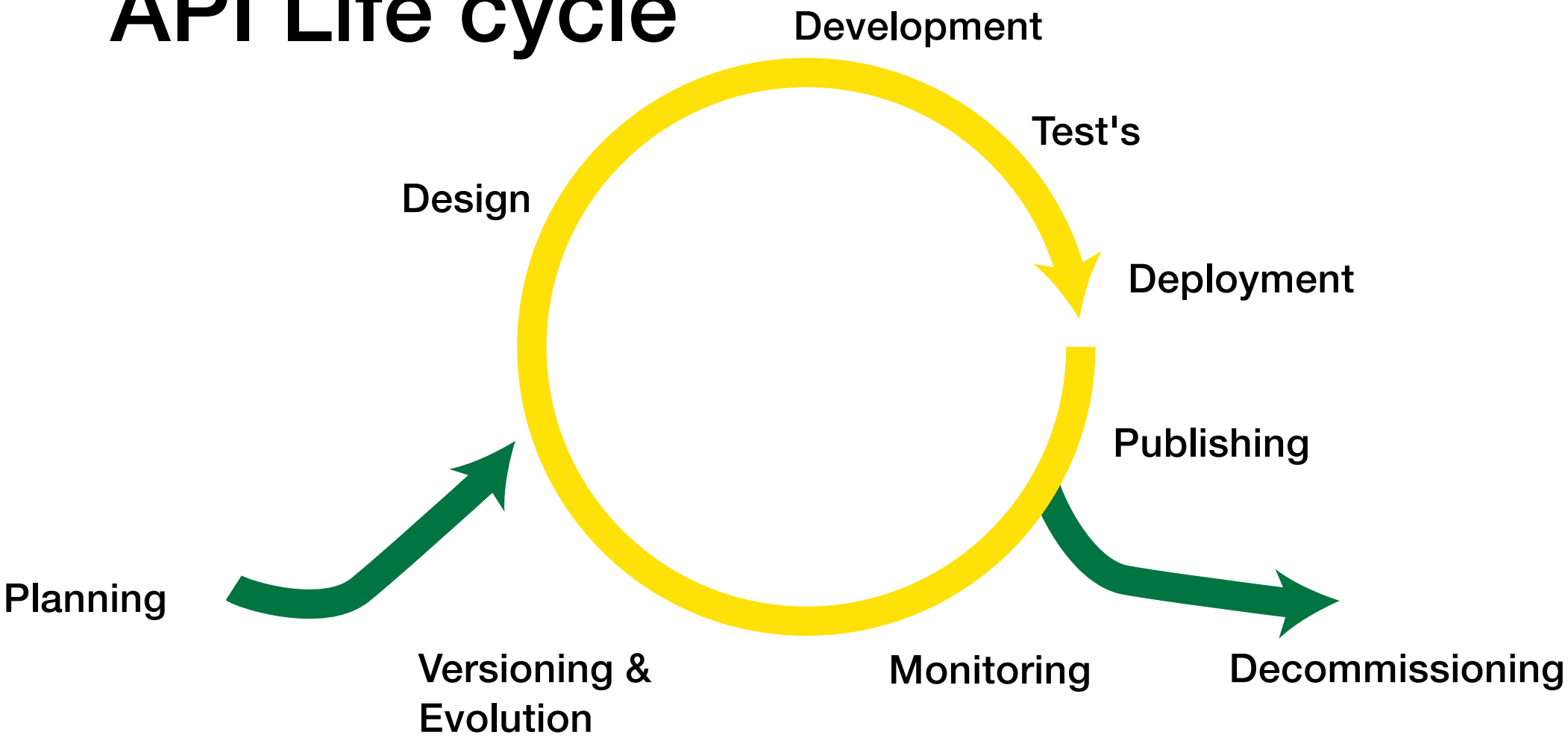
```
janmsi@MB-01009 ~ % curl https://api.GitHub.com

~ % curl https://api.github.com/users/jesan-dv-lnu
```



Step 1: Log in to your GitHub account.
Step 2: Find *Personal access tokens.*
Step 3: Create a new token**.**
Step 4: Confirm with your password.
Step 5: Add a description
Step 6: Select all the "scopes"
Step 7: ***Finally, tap on*** *generate new token*.

API Life cycle



Design of an API platform.

In addition to ensuring that an API delivers the intended functionality, other aspects need to be considered in the design.

Purpose and sufficient granularity

User perspective and reuse. For example, APIs to be used in different systems must be designed for reuse.

Follow established design principles for the current API type

For example, only APIs that are at level 3 can be considered RESTful.

Support API evolution

"Breaking & non-breaking changes". API versioning is necessary to handle (upcoming) changes to the contract.

Error handling

Security

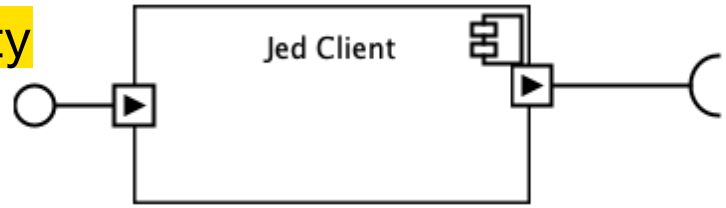
Specifying an API

API specification consists of a number of architecturally significant decisions

”Design by contract”

Reuse

Functionality



Security

Granularity

Error handling

API evolution

OpenAPI

A standard for http based APIs

Makes it easier for developers to describe and document APIs in a structured and uniform way.

Facilitates the development process as tools can automate and create clear documentation for APIs.

The structure provides a "standardized way" which increases the likelihood that the API is easy to understand, use and maintain, both internally and externally.

Open API Example

Parts in the specification:

Info – Metadata about the API - such as name, description, and version.

Paths – Defines the various endpoints (routes) that the API offers, including HTTP methods (GET, POST, etc.) and response codes.

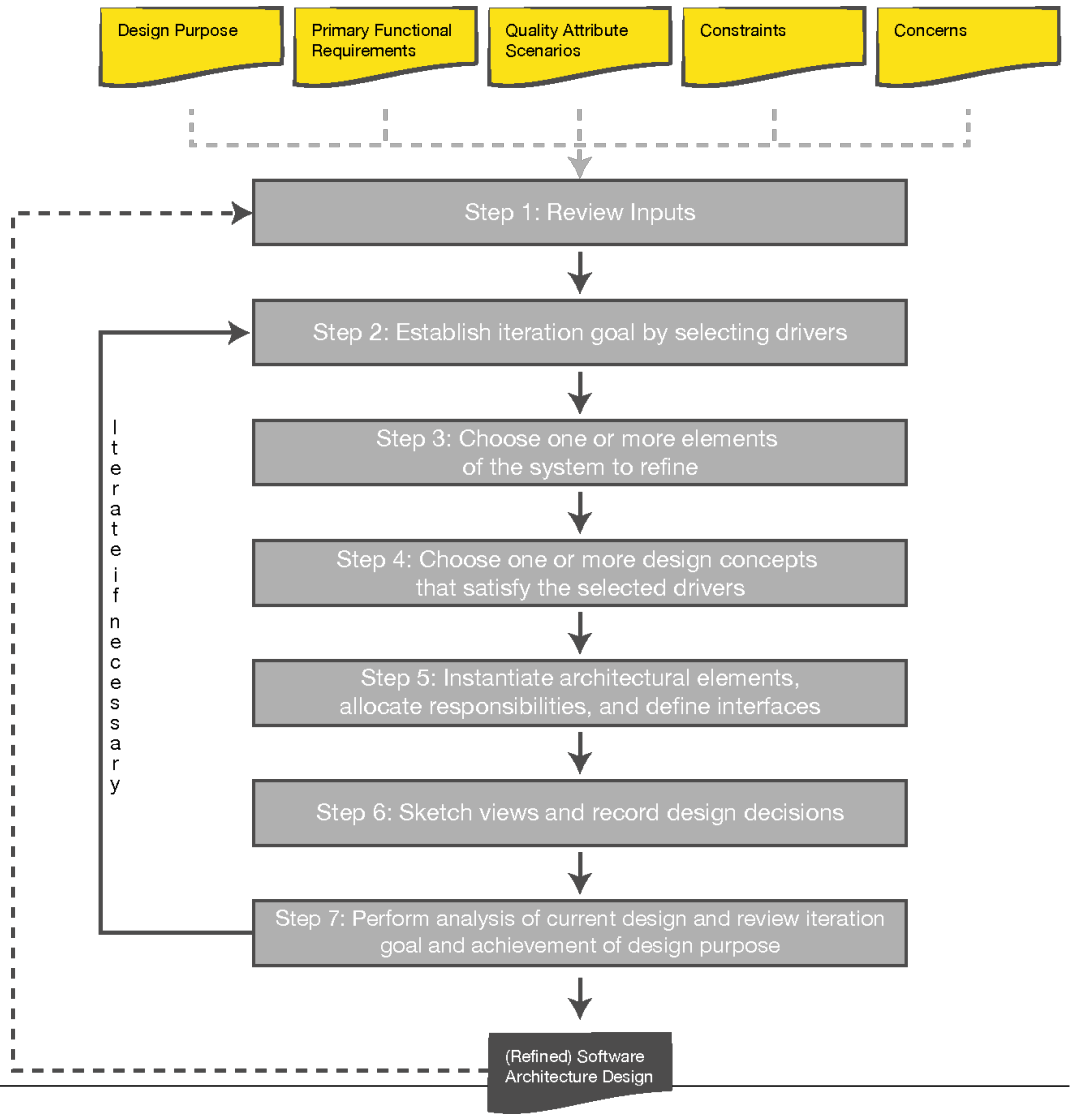
Responses – Describes responses that the API can return, including status codes and data types.

Request Body – Defines what data can be sent to the API, for example for POST or PUT methods.

Open API Exempel

```yaml
openapi: 3.0.0
info:
  title: Simple API
  description: Ett enkelt exempel på en OpenAPI-specifikation
  version: 1.0.0
paths:
  /users:
    get:
      summary: Hämta alla användare
      responses:
        '200':
          description: En lista över användare
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
    post:
      summary: Skapa en ny användare
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                name:
                  type: string
                  example: John Doe
      responses:
        '201':
          description: Användare skapad
```

APIs & ADD


Design Purpose
Primary Functional Requirements
Quality Attribute Scenarios
Constraints
Concerns
Step 1: Review Inputs
Step 2: Establish iteration goal by selecting drivers
Step 3: Choose one or more elements of the system to refine
Step 4: Choose one or more design concepts that satisfy the selected drivers
Step 5: Instantiate architectural elements, allocate responsibilities, and define interfaces
Step 6: Sketch views and record design decisions
Step 7: Perform analysis of current design and review iteration goal and achievement of design purpose
Iterate if necessary
(Refined) Software Architecture Design

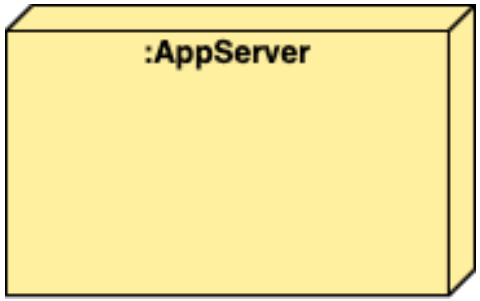
Deployment

Deployment

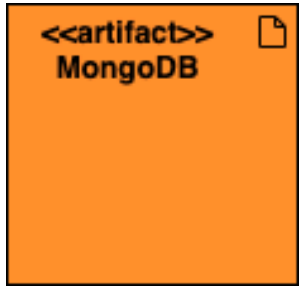
Deployability refers to a property of software indicating that it may be deployed—allocated to an environment for execution—with predictable time and effort.

Deployments in UML

Elements



deployment target/node



artifact

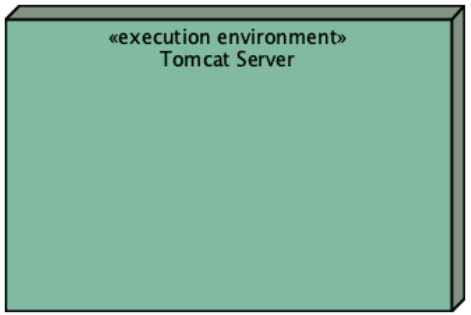
dependency

Communication path

Deployment Target

The physical or virtual environment where software is deployed (e.g., a server, cloud instance, or device

A device, a physical node



An environment, e.g, a container

Deployment Artifact

An artifact is a classifier that represents some physical entity, a piece of information that is used or is produced by a software development process, or by deployment and operation of a system

Some real life examples of UML artifacts are:

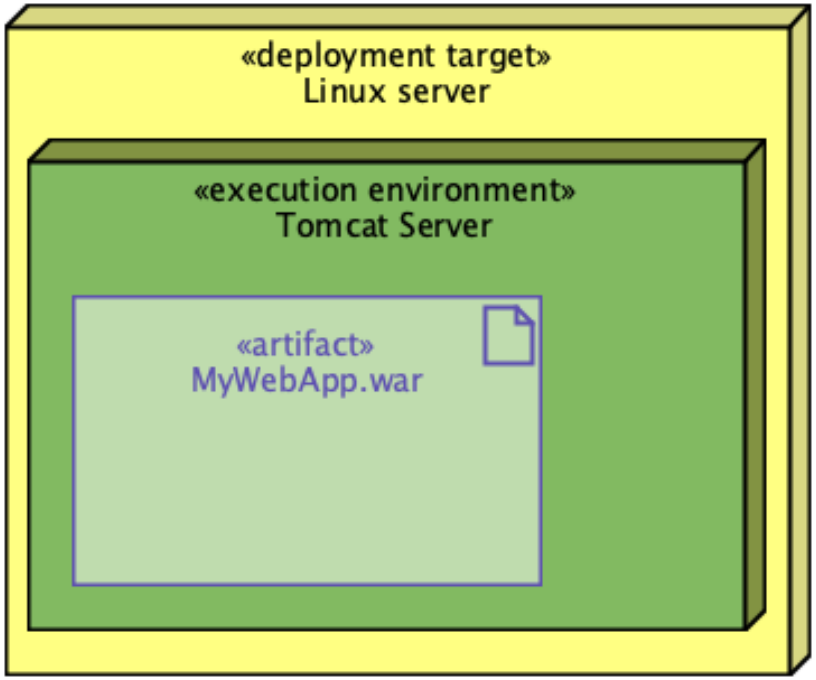
- text document
- source file
- script
- binary executable file
- archive file
- database table

Deploy Artifacts on Nodes

Linux Server is a node where the application is deployed.

Tomcat Server is an execution environment inside the Linux server.

MyApp.war is an artifact deployed within Tomcat.

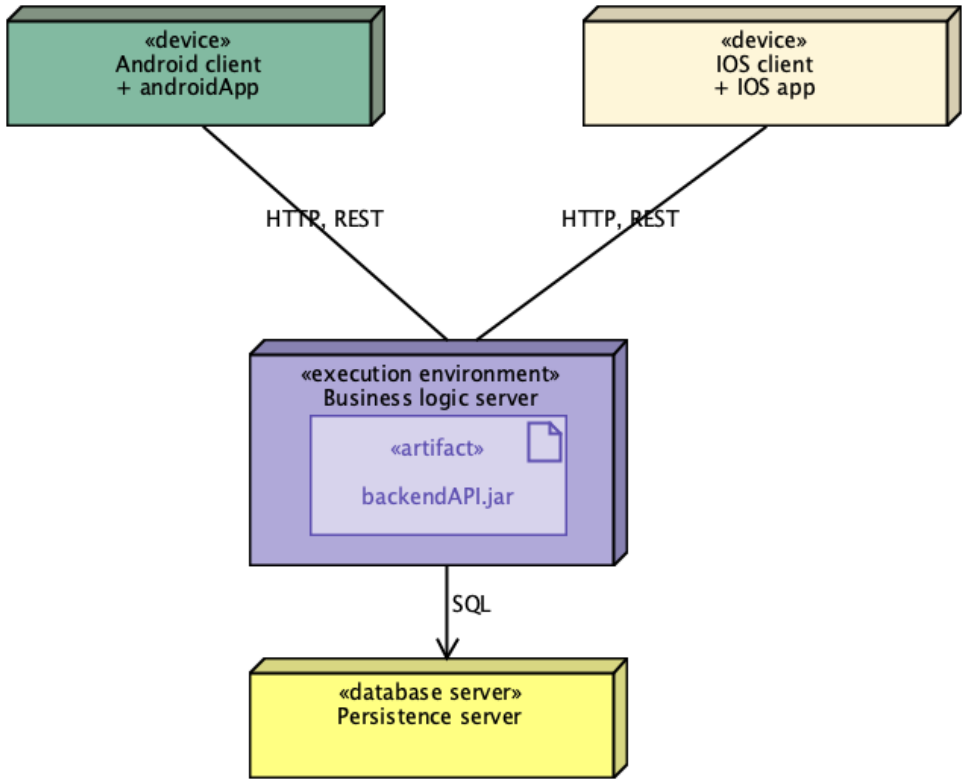


System Architecture
Deployments

Manifestation diagrams

Contents of artifacts

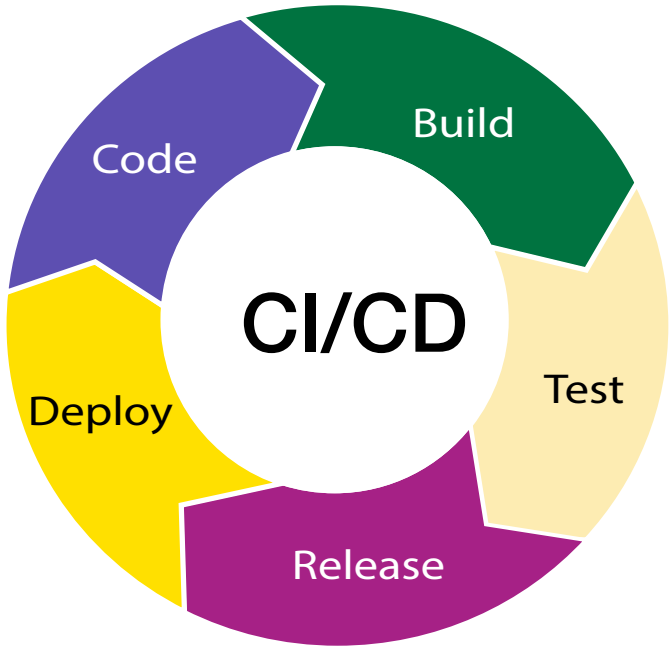
```
myapp.jar
├── META-INF/
│   ├── MANIFEST.MF
├── com/
│   ├── example/
│   │   ├── Main.class
│   │   ├── Helper.class
├── resources/
│   ├── config.properties
│   ├── logo.png
├── lib/
│   ├── external-lib.jar
```



Deployment

Deployability refers to a property of software indicating that it may be deployed—allocated to an environment for execution—with predictable time and effort.

If deployment is automatic
is it referred to as continuous deployment

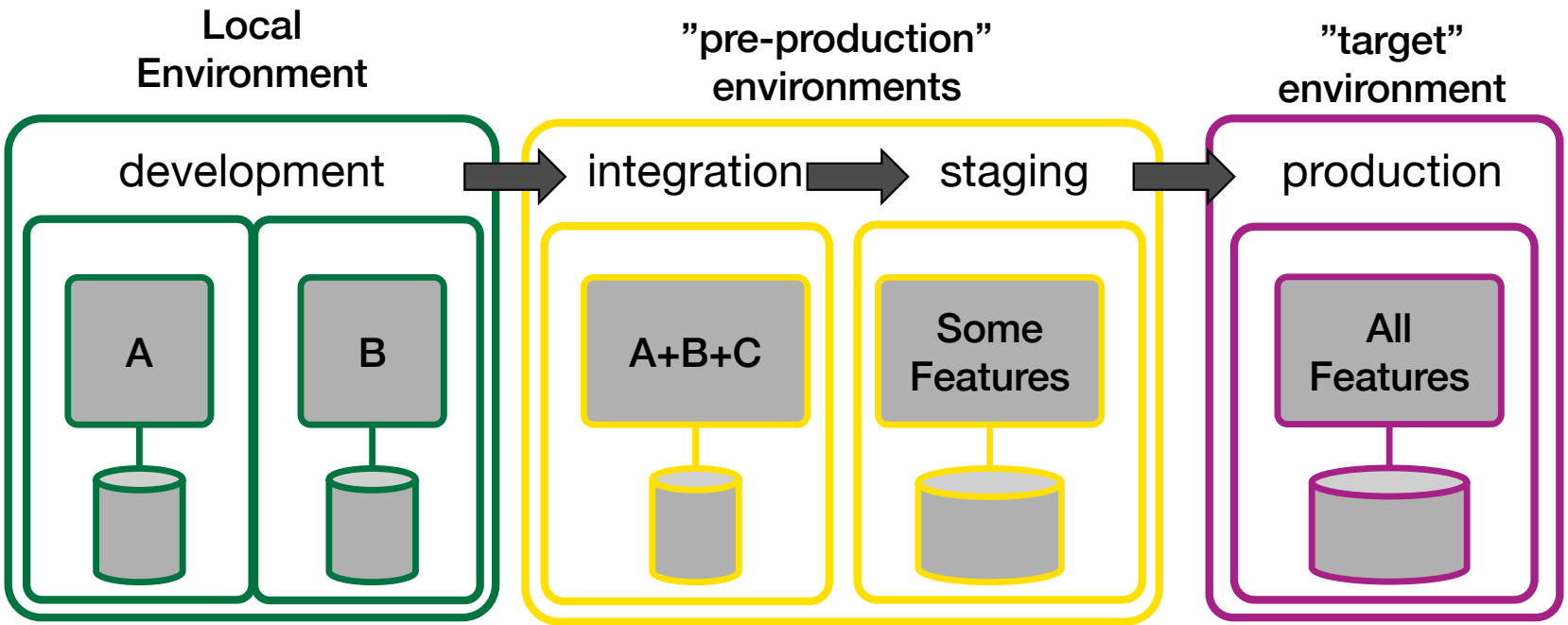


The Architecture Question!

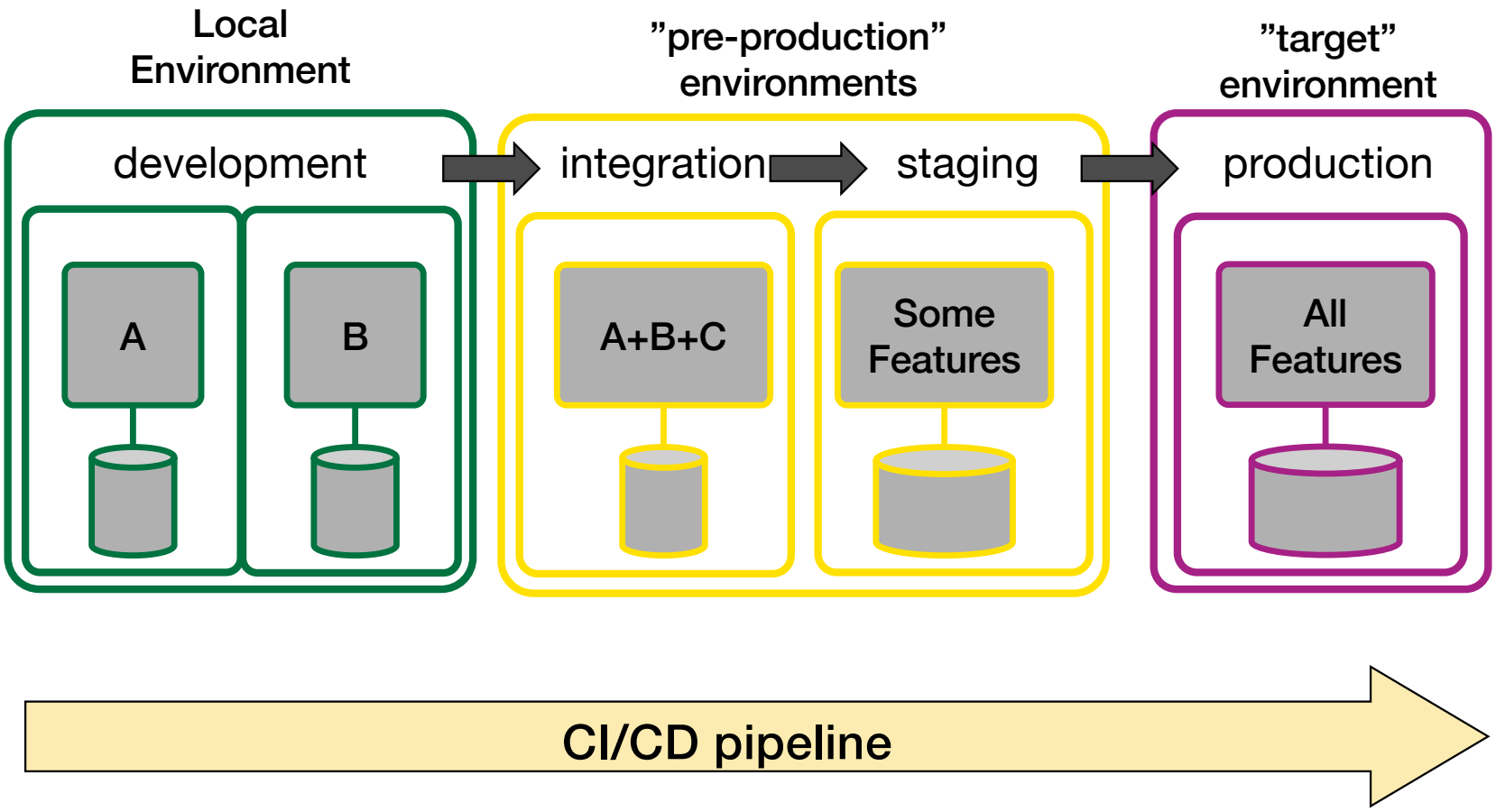
APIs and composability are necessary but not enough!

How can we create an architecture that makes it easier for developers to publish services—their APIs—so that their customers can use them?

Environment



Environment



Modeling CI/CD in UML

Modeling a **CI/CD pipeline** using a **UML Deployment Diagram** involves representing the physical architecture of the system, including nodes (hardware or software components), artifacts, and their relationships.

Nodes**:** Represent servers or execution environments

Artifacts**:** Represent deployment units like Docker images, build artifacts, or source code.

Communication Paths**:** Represent network connections between nodes.

Execution Environments**:** Represent runtime environments like containers or virtual machines.

Stereotypes**:** Label elements with meaningful tags like <<build server>>, <<artifact repository>>, <<deployment server>>, etc.

Steps to Model

Identify the Nodes (CI/CD Components)

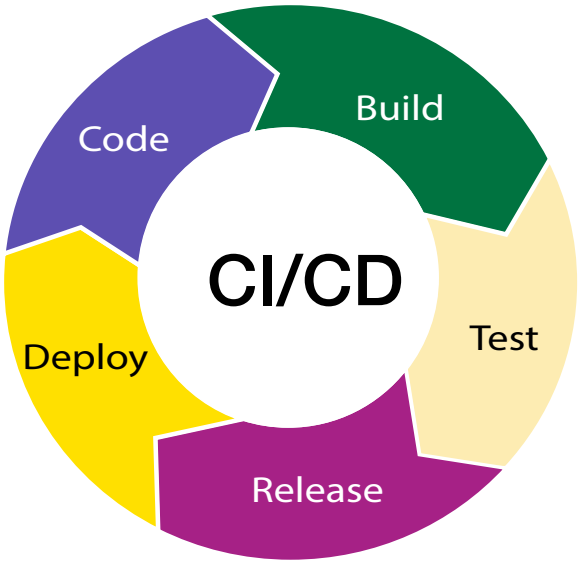
Developer Machine, Version Control System CI Server, Artifact Repository, Testing Server, Staging Server, Production Server

Define the Artifacts

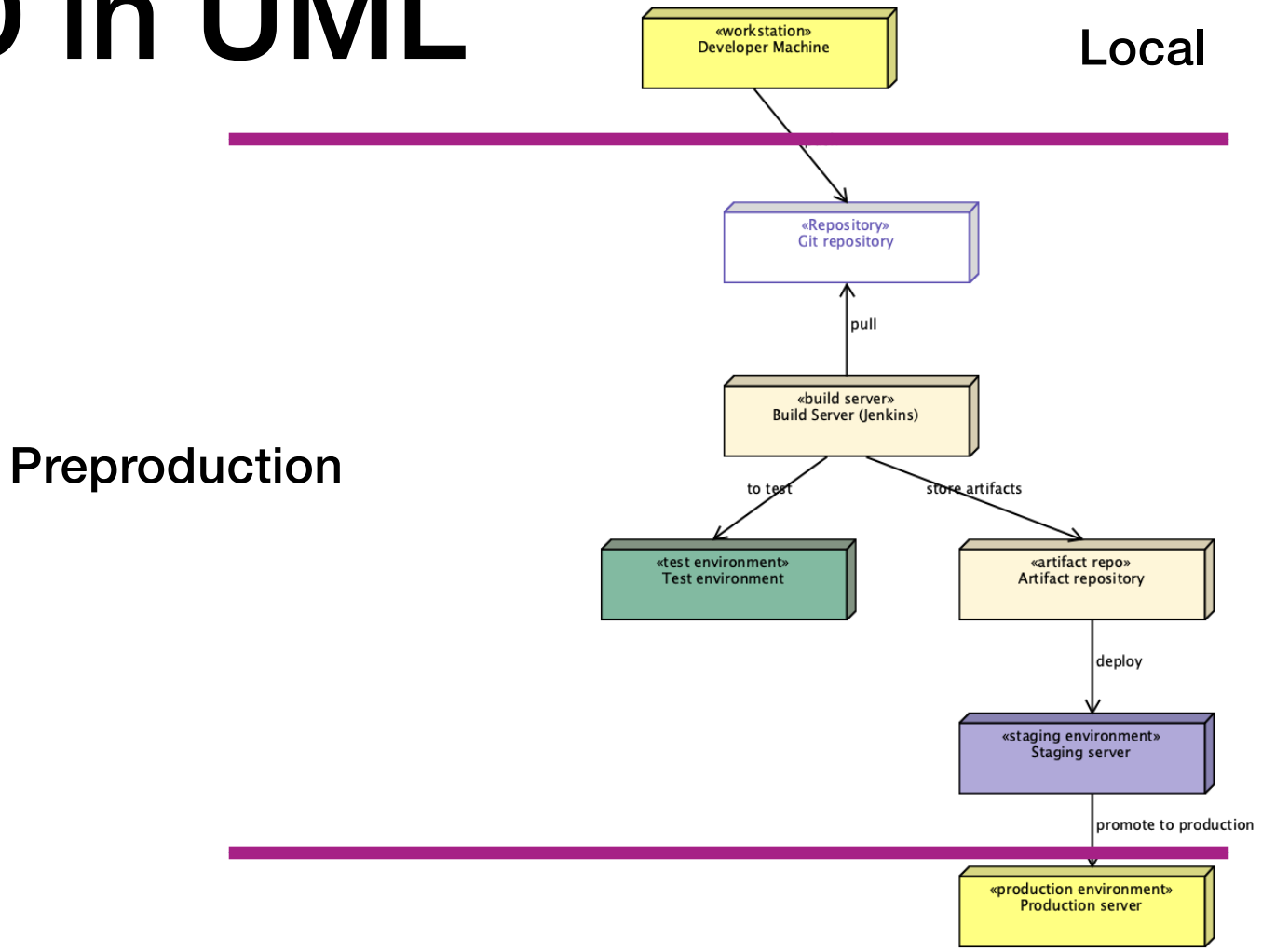
Source code
Build artifacts
Configuration files

Define the Communication Paths

Push/Pull (git)



CI/CD in UML


«workstation»
Developer Machine
Local
«Repository»
Git repository
pull
«build server»
Build Server (Jenkins)
Preproduction
to test
store artifacts
«test environment»
Test environment
«artifact repo»
Artifact repository
deploy
«staging environment»
Staging server
promote to production
«production environment»
Production server

General scenario

A requirement template!

Portion of Scenario	Description	Possible Values
Source	The trigger for the deployment	End user, developer, system administrator, operations personnel, component marketplace, product owner.
Stimulus	What causes the trigger	A new element is available to be deployed. This is typically a request to replace a software element with a new version (e.g., fix a defect, apply a security patch, upgrade to the latest release of a component or framework, upgrade to the latest version of an internally produced element). A new element is approved for incorporation. An existing element/set of elements needs to be rolled back.
Artifacts	What is to be changed	Specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates. Thus the artifact might be a single software element, multiple software elements, or the entire system.
Environment	Where the artifacts are being deployed	Integration environment. Staging environment. Production environment.
Response	What should happen	Full deployment. Subset deployed to a specified portion of: users, virtual machines (VMs), containers, servers, platforms. Monitor the new components. Roll back a previous deployment.
Response measure	How effective is the deployment, in terms of cost, time, or process effectiveness	Cost in terms of: ▪ Number, size, complexity of affected artifacts ▪ Average/worst-case effort ▪ Elapsed time ▪ Money (direct outlay or opportunity cost) ▪ New defects introduced Extent to which this deployment/rollback affects other functions or quality attributes. Number of failed deployments. Repeatability of the process. Traceability of the process.

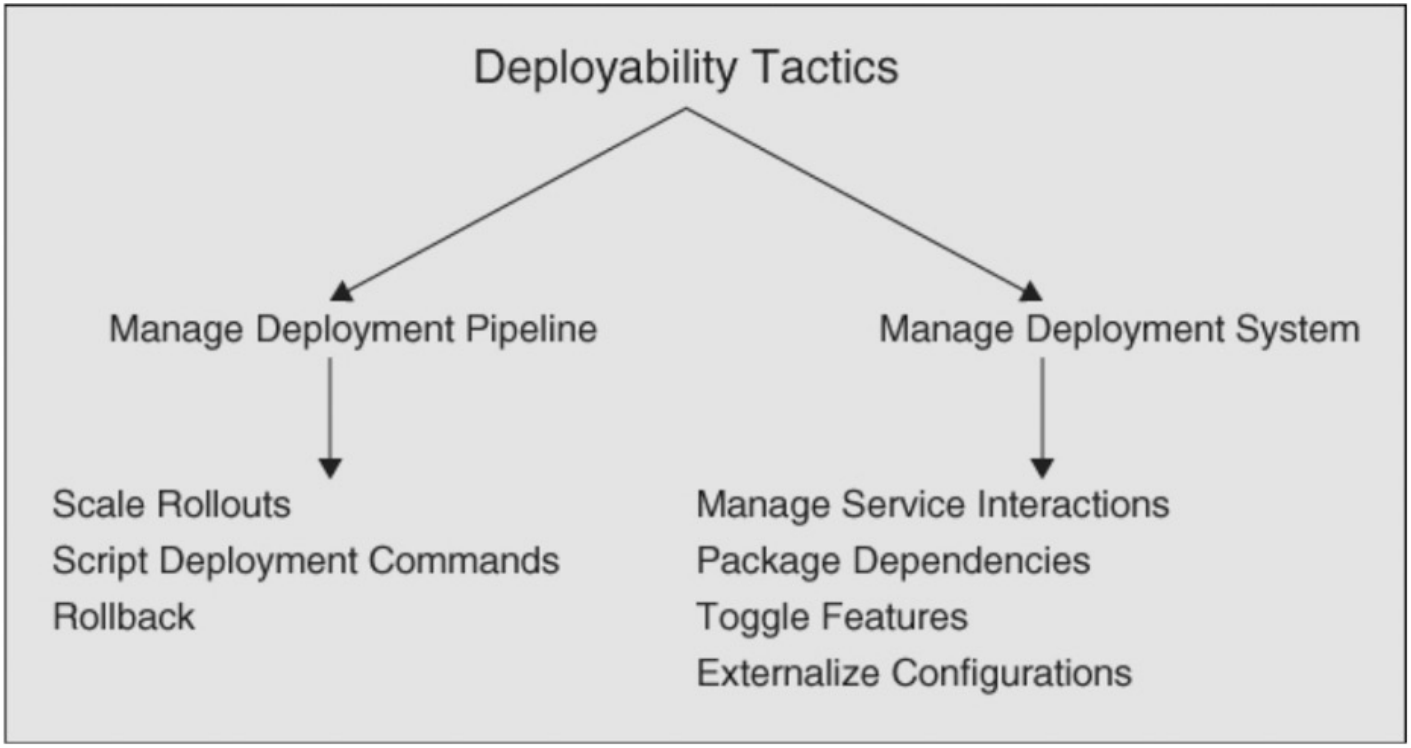
Design for Deployability

Granularity. A deployment can be the entire system or parts of a system. If the architecture creates the conditions for more fine-grained granularity in the deployment, some risks can be mitigated.

Configurable & Controllable. The architecture should allow for deployment at different resolutions, monitoring the parts being deployed, and rollback deployments that are not working.

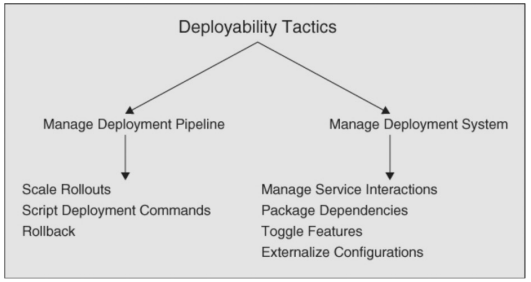
Efficient. The architecture should support rapid deployment and rollback with reasonable efforts.

Tactics


Deployability Tactics
Manage Deployment Pipeline
Manage Deployment System
Scale Rollouts
Script Deployment Commands
Rollback
Manage Service Interactions
Package Dependencies
Toggle Features
Externalize Configurations

Tactics

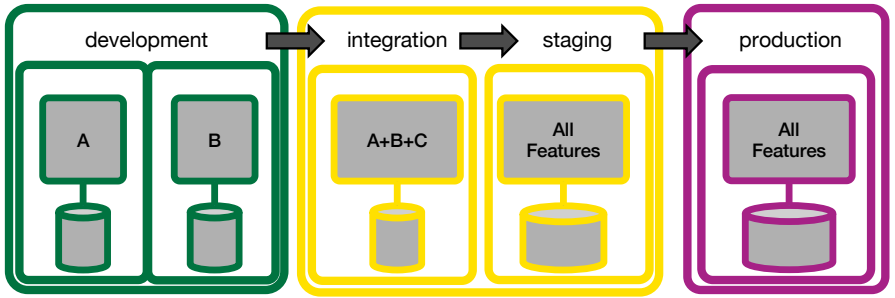
Manage deployment pipeline



Design decisions regarding the deployment infrastructure.

Scale rollouts Deploy to subsets of users. Limits the effects of any errors and facilitates monitoring and rollback.

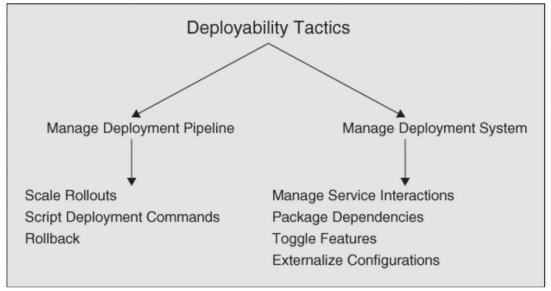
Roll back. If a distribution has defects, it needs to be "restored" to its previous state. Because deployments can include multiple updates to multiple services, the "recovery mechanism" is complex and should be automated.



Script deployment commands.
The steps to be performed during a deployment should be "scripted".

Tactics

Manage deployment System



Design decisions regarding the deployment infrastructure.

Package dependencies. Elements are packaged as "self-contained", i.e. kept together with their "internal dependencies". Cohesion + Coupling!

Manage service interactions. Deployment of services with external dependencies must be managed separately to avoid incompatibilities.

Feature Toggle. A "switch" used for new features, so that a feature can be automatically disabled at runtime, without the need for a new deployment. Reduce the consequences of incorrect features.

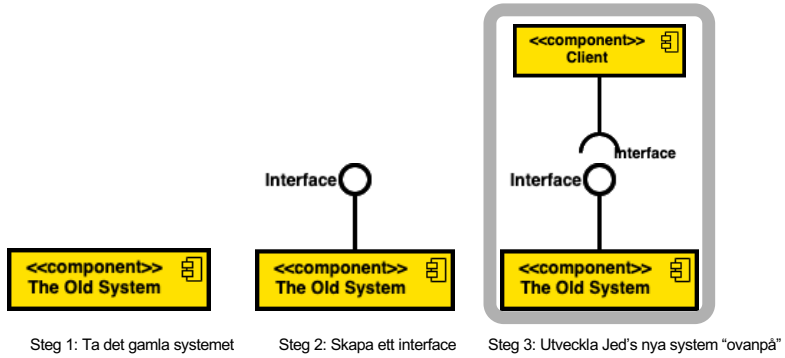
Externalize configurations. The system should not have any "hardcoded" configurations. It limits the ability to move it from one environment to another (e.g., from integration to production).

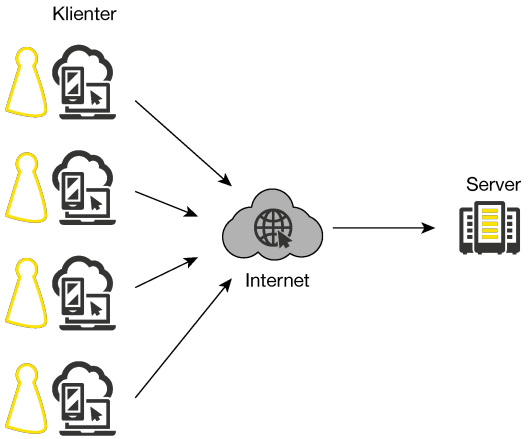
Patterns and Deployment

Microservices

Monoliths and modular monoliths

N-tier deployments



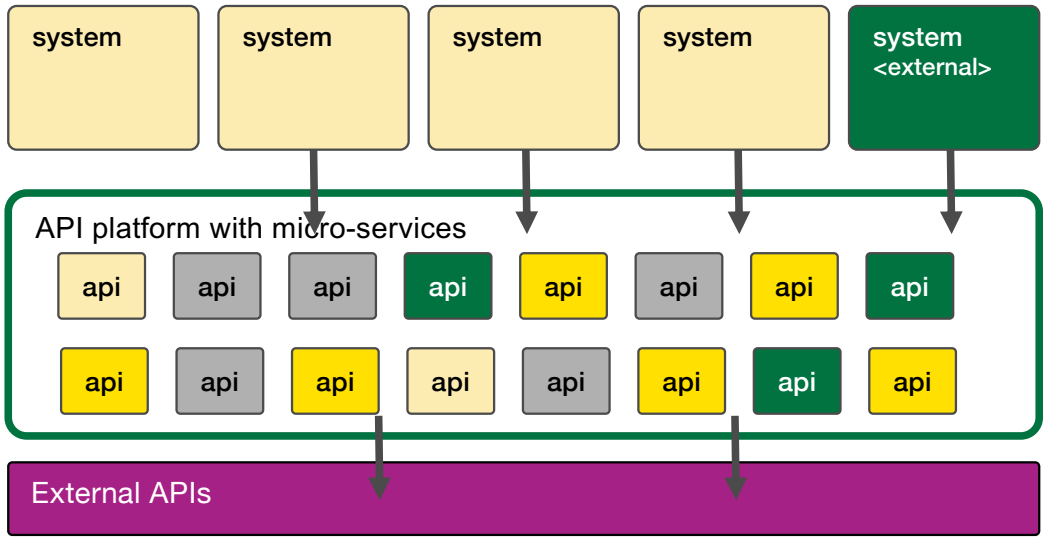


From Applications to Systems

Microservices are integrated to create a system

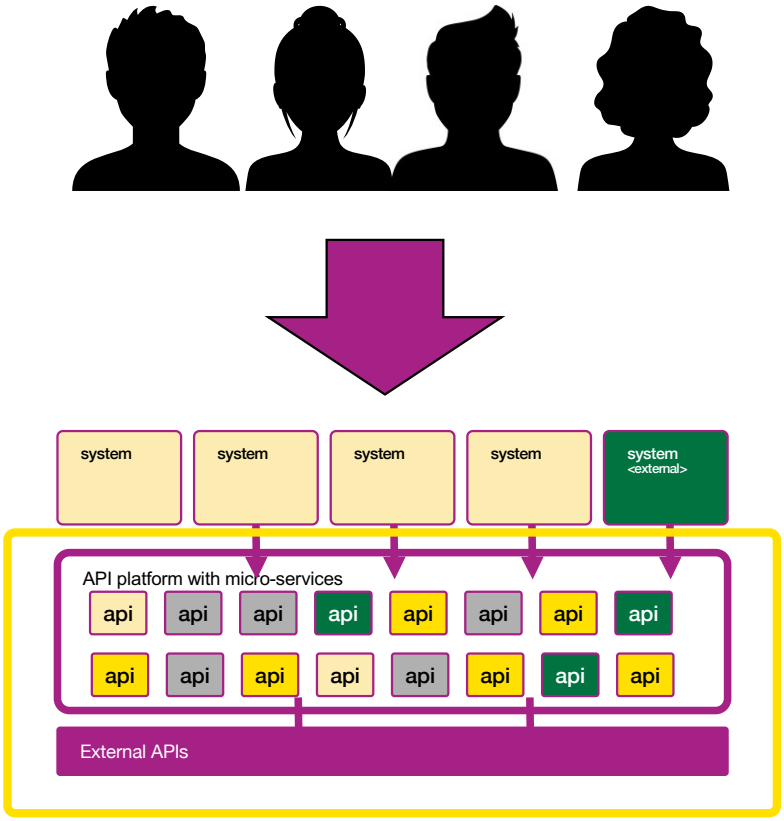
Microservices are separate independent components

Integrates through platforms or through an API gateway



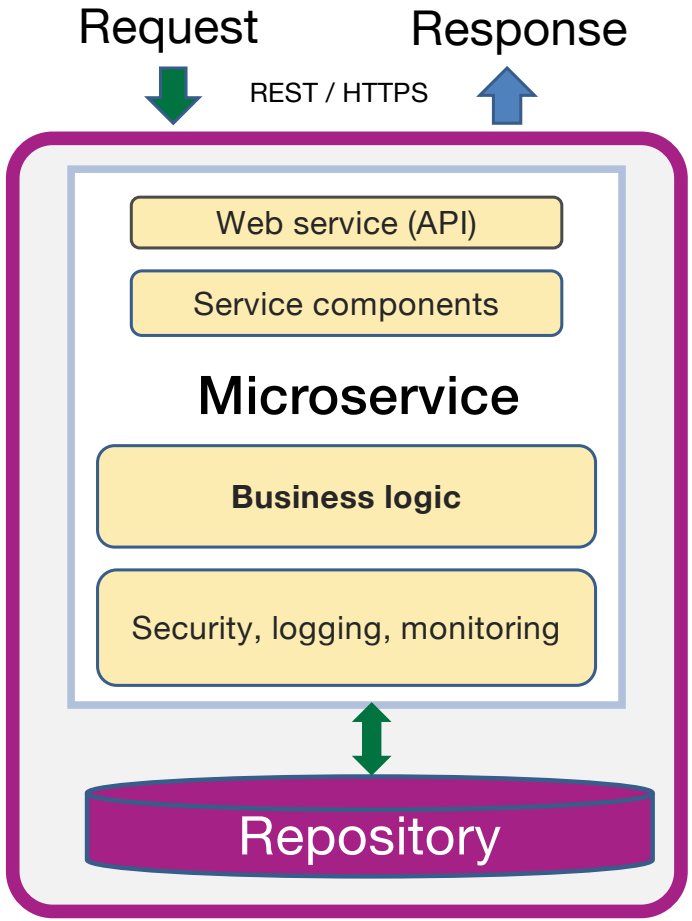
An Internal Architecture

- Not obvious to users if the system is based on microservices
- A decentralized architecture does not mean a "fragmented" user experience. The system creates a cohesive experience
- required for performance, high availability, extreme loads



Microservices

A Conceptual model



Create a Microservice

Small ”unit” with functionality

"Self-contained" and independent "technology stack"
Separate data warehouses

Synchronization with other services as needed via a persistence layer
Invoked via API with Request/Response
Commonly: REST, HTTP, JSON

"Self-contained”

- Internal functions that are not exposed externally
- Developers are "free" to choose technical components

Putting together a system

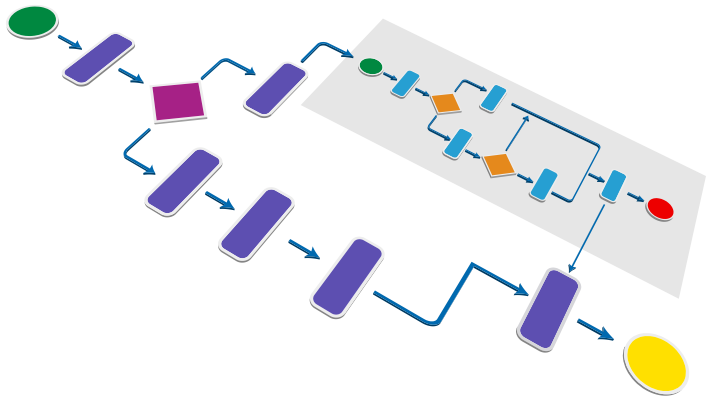
Each microservice performs a small limited task – is not a standalone system!

Orchestrate services

Multiple microservices are required to perform a complex task

Security

Load balancing



API Gateway

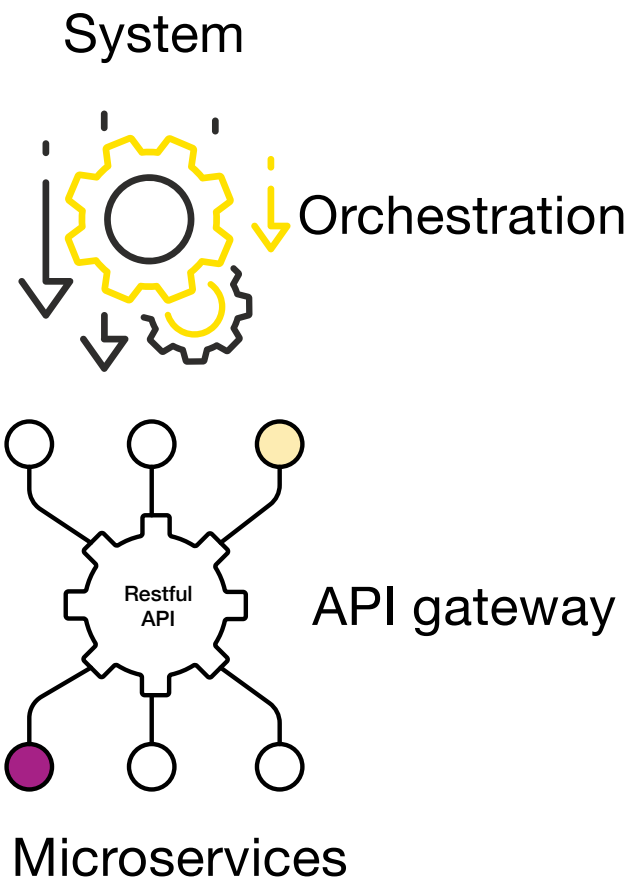
Requests to a microservice are usually not made directly!

A layer that manages access to microservices is needed to enable a flexible deployment.

An API gateway is an entry point for requests.

Developers don't need to know the "physical address" of each service (nothing should be hardcoded).

Manage authentication, protocol conversions, communication, routing, load balancing, etc.



Other patterns

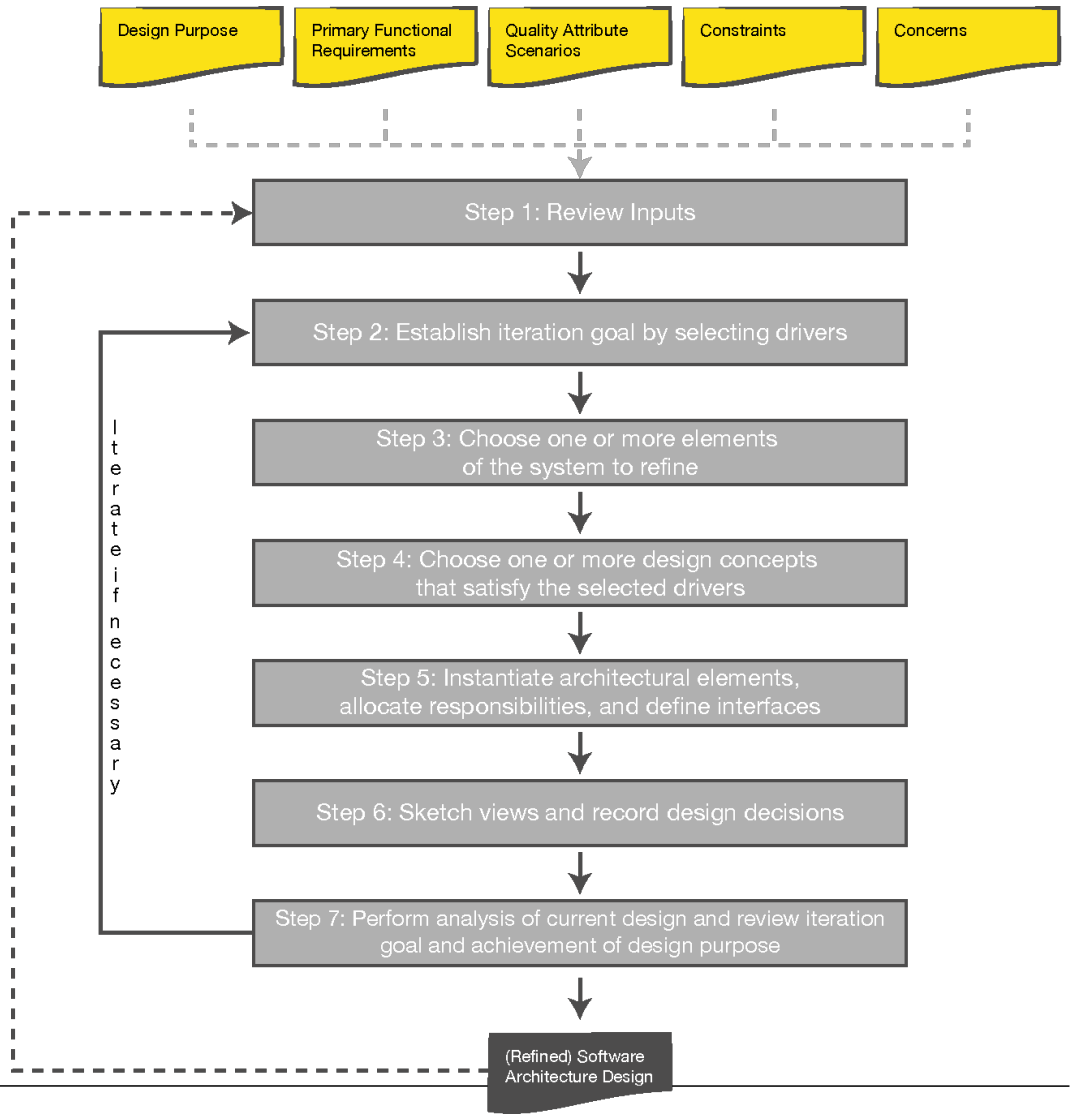
Blue – Green - Blue-green deployment creates n new instances of service A, the "green" instances. Once n instances are deployed, the information changes to point to the new version.

Rolling Upgrade – A rolling upgrade replaces instances of Service A with instances of the new version one by one.

Canary testing – Before a new version of a service is rolled out, it is tested in full operation in the production environment, but only for a small part of the users.

A/B testing

ADD & Deployability



Today’s lecture

APIs

”composability”

Business agility

Open API

CD/CI

environments

”deployability”

Patterns & tactics

“Composability,
Pieces fit, systems align,
Deploy with a breeze.”

— ChatGPT 4o

2DV604 Software Architecture

Lnu.se